

Introduction to Binary Search and Complexity Analysis with Python

Part 1 of "Data Structures and Algorithms in Python"

[Data Structures and Algorithms in Python](#) is beginner-friendly introduction to common data structures (linked lists, stacks, queues, graphs) and algorithms (search, sorting, recursion, dynamic programming) in Python, designed to help you prepare for coding interviews and assessments. Check out the full series here:

1. [Binary Search and Complexity Analysis](#)
2. [Python Classes and Linked Lists](#)
3. Arrays, Stacks, Queues and Strings (coming soon)
4. Binary Search Trees and Hash Tables (coming soon)
5. Insertion Sort, Merge Sort and Divide-and-Conquer (coming soon)
6. Quicksort, Partitions and Average-case Complexity (coming soon)
7. Recursion, Backtracking and Dynamic Programming (coming soon)
8. Knapsack, Subsequence and Matrix Problems (coming soon)
9. Graphs, Breadth-First Search and Depth-First Search (coming soon)
10. Shortest Paths, Spanning Trees & Topological Sorting (coming soon)
11. Disjoint Sets and the Union Find Algorithm (coming soon)
12. Interview Questions, Tips & Practical Advice (coming soon)

Earn a verified certificate of accomplishment for this course by signing up here: <http://pythonsa.com> .

Ask questions, get help & participate in discussions on the community forum: <https://jovian.ai/forum/c/data-structures-and-algorithms-in-python/78>

Prerequisites

This course assumes very little background in programming and mathematics, and you can learn the required concepts here:

- Basic programming with Python ([variables](#), [data types](#), [loops](#), [functions](#) etc.)
- Some high school mathematics ([polynomials](#), [vectors](#), [matrices](#) and [probability](#))
- No prior knowledge of data structures or algorithms is required

We'll cover any additional mathematical and theoretical concepts we need as we go along.

How to Run the Code

The best way to learn the material is to execute the code and experiment with it yourself. This tutorial is an executable [Jupyter notebook](#). You can *run* this tutorial and experiment with the code examples in a couple of ways: *using free online resources* (recommended) or *on your computer*.

Option 1: Running using free online resources (1-click, recommended)

The easiest way to start executing the code is to click the **Run** button at the top of this page and select **Run on Binder**. You can also select "Run on Colab" or "Run on Kaggle", but you'll need to create an account on [Google Colab](#) or [Kaggle](#) to use these platforms.

Option 2: Running on your computer locally

To run the code on your computer locally, you'll need to set up [Python](#), download the notebook and install the required libraries. We recommend using the [Conda](#) distribution of Python. Click the **Run** button at the top of this page, select the **Run Locally** option, and follow the instructions.

Jupyter Notebooks: This notebook is made of *cells*. Each cell can contain code written in Python or explanations in plain English. You can execute code cells and view the results instantly within the notebook. Jupyter is a powerful platform for experimentation and analysis. Don't be afraid to mess around with the code & break things - you'll learn a lot by encountering and fixing errors. You can use the "Kernel > Restart & Clear Output" menu option to clear all outputs and start again from the top.

Try executing the cells below:

```
# Import a library module
import math
```

```
# Use a function from the library
math.sqrt(49)
```

7.0

Problem

This course takes a coding-focused approach towards learning. In each notebook, we'll focus on solving one problem, and learn the techniques, algorithms, and data structures to devise an *efficient* solution. We will then generalize the technique and apply it to other problems.

In this notebook, we focus on solving the following problem:

QUESTION 1: Alice has some cards with numbers written on them. She arranges the cards in decreasing order, and lays them out face down in a sequence on a table. She challenges Bob to pick out the card containing a given number by turning over as few cards as possible. Write a function to help Bob locate the card.



This may seem like a simple problem, especially if you're familiar with the concept of *binary search*, but the strategy and technique we learning here will be widely applicable, and we'll soon use it to solve harder problems.

Why You Should Learn Data Structures and Algorithms

Whether you're pursuing a career in software development or data science, it's almost certain that you'll be asked to solve programming problems like *reversing a linked list* or *balancing a binary tree* in a technical interview or coding assessment.

It's well known, however, that you will almost never face these problems in your job as a software developer. So it's reasonable to wonder why such problems are asked in interviews and coding assessments. Solving programming problems demonstrates the following traits:

1. You can **think about a problem systematically** and solve it systematically step-by-step.
2. You can **envision different inputs, outputs, and edge cases** for programs you write.
3. You can **communicate your ideas clearly** to co-workers and incorporate their suggestions.
4. Most importantly, you can **convert your thoughts and ideas into working code** that's also readable.

It's not your knowledge of specific data structures or algorithms that's tested in an interview, but your approach towards the problem. You may fail to solve the problem and still clear the interview or vice versa. In this course, you will learn the skills to both solve problems and clear interviews successfully.

The Method

Upon reading the problem, you may get some ideas on how to solve it and your first instinct might be to start writing code. This is not the optimal strategy and you may end up spending a longer time trying to solve the problem due to coding errors, or may not be able to solve it at all.

Here's a systematic strategy we'll apply for solving problems:

1. State the problem clearly. Identify the input & output formats.
2. Come up with some example inputs & outputs. Try to cover all edge cases.
3. Come up with a correct solution for the problem. State it in plain English.
4. Implement the solution and test it using example inputs. Fix bugs, if any.
5. Analyze the algorithm's complexity and identify inefficiencies, if any.
6. Apply the right technique to overcome the inefficiency. Repeat steps 3 to 6.

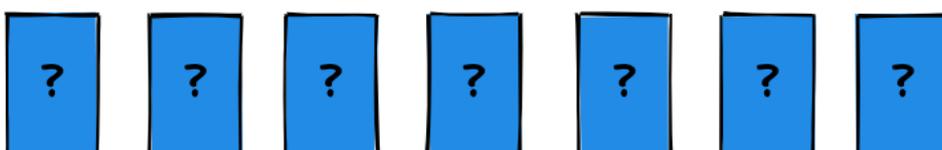
"Applying the right technique" is where the knowledge of common data structures and algorithms comes in handy.

Use this template for solving problems by applying this method: <https://jovian.ai/aakashns/python-problem-solving-template>

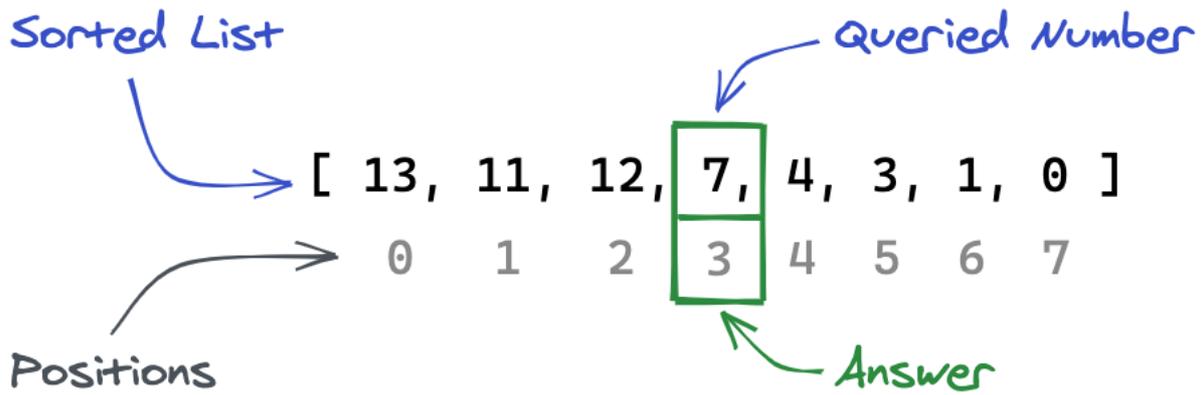
Solution

1. State the problem clearly. Identify the input & output formats.

You will often encounter detailed word problems in coding challenges and interviews. The first step is to state the problem clearly and precisely in abstract terms.



In this case, for instance, we can represent the sequence of cards as a list of numbers. Turning over a specific card is equivalent to accessing the value of the number at the corresponding position in the list.



The problem can now be stated as follows:

Problem

We need to write a program to find the position of a given number in a list of numbers arranged in decreasing order. We also need to minimize the number of times we access elements from the list.

Input

1. cards: A list of numbers sorted in decreasing order. E.g. [13, 11, 10, 7, 4, 3, 1, 0]
2. query: A number, whose position in the array is to be determined. E.g. 7

Output

3. position: The position of query in the list cards. E.g. 3 in the above case (counting from 0)

Based on the above, we can now create the signature of our function:

```
def locate_card(cards, query):  
    pass
```

Tips:

- Name your function appropriately and think carefully about the signature
- Discuss the problem with the interviewer if you are unsure how to frame it in abstract terms
- Use descriptive variable names, otherwise you may forget what a variable represents

2. Come up with some example inputs & outputs. Try to cover all edge cases.

Before we start implementing our function, it would be useful to come up with some example inputs and outputs which we can use later to test out problem. We'll refer to them as *test cases*.

Here's the test case described in the example above.

```
cards = [13, 11, 10, 7, 4, 3, 1, 0]
query = 7
output = 3
```

We can test our function by passing the inputs into function and comparing the result with the expected output.

```
result = locate_card(cards, query)
print(result)
```

None

```
result == output
```

False

Obviously, the two result does not match the output as we have not yet implemented the function.

We'll represent our test cases as dictionaries to make it easier to test them once we write implement our function. For example, the above test case can be represented as follows:

```
test = {
    'input': {
        'cards': [13, 11, 10, 7, 4, 3, 1, 0],
        'query': 7
    },
    'output': 3
}
```

The function can now be tested as follows.

```
locate_card(**test['input']) == test['output']
```

False

Our function should be able to handle any set of valid inputs we pass into it. Here's a list of some possible variations we might encounter:

1. The number query occurs somewhere in the middle of the list cards.
2. query is the first element in cards.
3. query is the last element in cards.
4. The list cards contains just one element, which is query.
5. The list cards does not contain number query.
6. The list cards is empty.
7. The list cards contains repeating numbers.
8. The number query occurs at more than one position in cards.
9. (can you think of any more variations?)

Edge Cases: It's likely that you didn't think of all of the above cases when you read the problem for the first time. Some of these (like the empty array or `query` not occurring in `cards`) are called *edge cases*, as they represent rare or extreme examples.

While edge cases may not occur frequently, your programs should be able to handle all edge cases, otherwise they may fail in unexpected ways. Let's create some more test cases for the variations listed above. We'll store all our test cases in an list for easier testing.

```
tests = []
```

```
# query occurs in the middle
tests.append(test)

tests.append({
  'input': {
    'cards': [13, 11, 10, 7, 4, 3, 1, 0],
    'query': 1
  },
  'output': 6
})
```

```
# query is the first element
tests.append({
  'input': {
    'cards': [4, 2, 1, -1],
    'query': 4
  },
  'output': 0
})
```

```
# query is the last element
tests.append({
  'input': {
    'cards': [3, -1, -9, -127],
    'query': -127
  },
  'output': 3
})
```

```
# cards contains just one element, query
tests.append({
  'input': {
    'cards': [6],
    'query': 6
  },
  'output': 0
})
```

The problem statement does not specify what to do if the list `cards` does not contain the number `query` .

1. Read the problem statement again, carefully.
2. Look through the examples provided with the problem.
3. Ask the interviewer/platform for a clarification.
4. Make a reasonable assumption, state it and move forward.

We will assume that our function will return `-1` in case `cards` does not contain `query` .

```
# cards does not contain query
tests.append({
  'input': {
    'cards': [9, 7, 5, 2, -9],
    'query': 4
  },
  'output': -1
})
```

```
# cards is empty
tests.append({
  'input': {
    'cards': [],
    'query': 7
  },
  'output': -1
})
```

```
# numbers can repeat in cards
tests.append({
  'input': {
    'cards': [8, 8, 6, 6, 6, 6, 6, 3, 2, 2, 2, 0, 0, 0],
    'query': 3
  },
  'output': 7
})
```

In the case where `query` occurs multiple times in `cards` , we'll expect our function to return the first occurrence of `query` .

While it may also be acceptable for the function to return any position where `query` occurs within the list, it would be slightly more difficult to test the function, as the output is non-deterministic.

```
# query occurs multiple times
tests.append({
  'input': {
    'cards': [8, 8, 6, 6, 6, 6, 6, 6, 3, 2, 2, 2, 0, 0, 0],
    'query': 6
  },
  },
```

```
'output': 2
})
```

Let's look at the full set of test cases we have created so far.

```
tests
```

```
[{'input': {'cards': [13, 11, 10, 7, 4, 3, 1, 0], 'query': 7}, 'output': 3},
 {'input': {'cards': [13, 11, 10, 7, 4, 3, 1, 0], 'query': 1}, 'output': 6},
 {'input': {'cards': [4, 2, 1, -1], 'query': 4}, 'output': 0},
 {'input': {'cards': [3, -1, -9, -127], 'query': -127}, 'output': 3},
 {'input': {'cards': [6], 'query': 6}, 'output': 0},
 {'input': {'cards': [9, 7, 5, 2, -9], 'query': 4}, 'output': -1},
 {'input': {'cards': [], 'query': 7}, 'output': -1},
 {'input': {'cards': [8, 8, 6, 6, 6, 6, 6, 3, 2, 2, 2, 0, 0, 0], 'query': 3},
  'output': 7},
 {'input': {'cards': [8, 8, 6, 6, 6, 6, 6, 6, 3, 2, 2, 2, 0, 0, 0],
  'query': 6},
  'output': 2}]
```

Great, now we have a fairly exhaustive set of test cases to evaluate our function.

Creating test cases beforehand allows you to identify different variations and edge cases in advance so that can make sure to handle them while writing code. Sometimes, you may start out confused, but the solution will reveal itself as you try to come up with interesting test cases.

Tip: Don't stress it if you can't come up with an exhaustive list of test cases though. You can come back to this section and add more test cases as you discover them. Coming up with good test cases is a skill that takes practice.

3. Come up with a correct solution for the problem. State it in plain English.

Our first goal should always be to come up with a *correct* solution to the problem, which may necessarily be the most *efficient* solution. The simplest or most obvious solution to a problem, which generally involves checking all possible answers is called the *brute force* solution.

In this problem, coming up with a correct solution is quite easy: Bob can simply turn over cards in order one by one, till he find a card with the given number on it. Here's how we might implement it:

1. Create a variable `position` with the value 0.
2. Check whether the number at index `position` in `card` equals `query`.
3. If it does, `position` is the answer and can be returned from the function
4. If not, increment the value of `position` by 1, and repeat steps 2 to 5 till we reach the last position.
5. If the number was not found, return -1.

Linear Search Algorithm: Congratulations, we've just written our first *algorithm*! An algorithm is simply a list of statements which can be converted into code and executed by a computer on different sets of inputs. This particular algorithm is called linear search, since it involves searching through a list in a linear fashion i.e. element after element.

Tip: Always try to express (speak or write) the algorithm in your own words before you start coding. It can be as brief or detailed as you require it to be. Writing is a great tool for thinking clearly. It's likely that you will find some parts of the solution difficult to express, which suggests that you are probably unable to think about it clearly. The more clearly you are able to express your thoughts, the easier it will be for you to turn into code.

4. Implement the solution and test it using example inputs. Fix bugs, if any.

Phew! We are finally ready to implement our solution. All the work we've done so far will definitely come in handy, as we now exactly what we want our function to do, and we have an easy way of testing it on a variety of inputs.

Here's a first attempt at implementing the function.

```
def locate_card(cards, query):  
    # Create a variable position with the value 0  
    position = 0  
  
    # Set up a loop for repetition  
    while True:  
  
        # Check if element at the current position matches the query  
        if cards[position] == query:  
  
            # Answer found! Return and exit..  
            return position  
  
        # Increment the position  
        position += 1  
  
        # Check if we have reached the end of the array  
        if position == len(cards):  
  
            # Number not found, return -1  
            return -1
```

Let's test out the function with the first test case

```
test
```

```
{'input': {'cards': [13, 11, 10, 7, 4, 3, 1, 0], 'query': 7}, 'output': 3}
```

```
result = locate_card(test['input']['cards'], test['input']['query'])  
result
```

```
3
```

```
result == output
```

```
True
```

Yay! The result matches the output.

To help you test your functions easily the `jovian` Python library provides a helper function `evaluate_test_case`. Apart from checking whether the function produces the expected result, it also displays the input, expected output, actual output from the function, and the execution time of the function.

```
!pip install jovian --upgrade --quiet
```

```
from jovian.pythondsa import evaluate_test_case
```

```
evaluate_test_case(locate_card, test)
```

Input:

```
{'cards': [13, 11, 10, 7, 4, 3, 1, 0], 'query': 7}
```

Expected Output:

```
3
```

Actual Output:

```
3
```

Execution Time:

```
0.004 ms
```

Test Result:

```
PASSED
```

```
(3, True, 0.004)
```

While it may seem like we have a working solution based on the above test, we can't be sure about it until we test the function with all the test cases.

We can use the `evaluate_test_cases` (plural) function from the `jovian` library to test our function on all the test cases with a single line of code.

```
from jovian.pythondsa import evaluate_test_cases
```

```
evaluate_test_cases(locate_card, tests)
```

TEST CASE #0

Input:

```
{'cards': [13, 11, 10, 7, 4, 3, 1, 0], 'query': 7}
```

Expected Output:

3

Actual Output:

3

Execution Time:

0.004 ms

Test Result:

PASSED

TEST CASE #1

Input:

```
{'cards': [13, 11, 10, 7, 4, 3, 1, 0], 'query': 1}
```

Expected Output:

6

Actual Output:

6

Execution Time:

0.003 ms

Test Result:

PASSED

TEST CASE #2

Input:

```
{'cards': [4, 2, 1, -1], 'query': 4}
```

Expected Output:

0

Actual Output:

0

Execution Time:

0.001 ms

Test Result:

PASSED

TEST CASE #3

Input:

```
{'cards': [3, -1, -9, -127], 'query': -127}
```

Expected Output:

3

Actual Output:

3

Execution Time:

0.004 ms

Test Result:

PASSED

TEST CASE #4

Input:

```
{'cards': [6], 'query': 6}
```

Expected Output:

0

Actual Output:

0

Execution Time:

0.005 ms

Test Result:

PASSED

TEST CASE #5

Input:

```
{'cards': [9, 7, 5, 2, -9], 'query': 4}
```

Expected Output:

-1

Actual Output:

-1

Execution Time:

0.004 ms

Test Result:

PASSED

TEST CASE #6

```
-----  
IndexError                                Traceback (most recent call last)  
<ipython-input-27-2111dddaa84c> in <module>  
----> 1 evaluate_test_cases(locate_card, tests)  
  
~/miniconda3/envs/pythonsa/lib/python3.6/site-packages/jovian/pythonsa/__init__.py in  
evaluate_test_cases(function, test_cases, error_only)  
    69         if not error_only:  
    70             print("\n\033[1mTEST CASE #{}\033[0m".format(i))  
----> 71         result = evaluate_test_case(function, test_case, display=False)  
    72         results.append(result)  
    73         if error_only and not result[1]:  
  
~/miniconda3/envs/pythonsa/lib/python3.6/site-packages/jovian/pythonsa/__init__.py in  
evaluate_test_case(function, test_case, display)  
    50  
    51     start = timer()  
----> 52     actual_output = function(**inputs)  
    53     end = timer()  
    54  
  
<ipython-input-19-9ed30c367c36> in locate_card(cards, query)
```

```

7
8     # Check if element at the current position matches the query
----> 9     if cards[position] == query:
10
11         # Answer found! Return and exit..

```

IndexError: list index out of range

Oh no! Looks like our function encountered an error in the sixth test case. The error message suggests that we're trying to access an index outside the range of valid indices in the list. Looks like the list `cards` is empty in this case, and may be the root of the problem.

Let's add some `print` statements within our function to print the inputs and the value of the `position` variable in each loop.

```

def locate_card(cards, query):
    position = 0

    print('cards:', cards)
    print('query:', query)

    while True:
        print('position:', position)

        if cards[position] == query:
            return position

        position += 1
        if position == len(cards):
            return -1

```

```

cards6 = tests[6]['input']['cards']
query6 = tests[6]['input']['query']

locate_card(cards6, query6)

```

```

cards: []
query: 7
position: 0

```

IndexError Traceback (most recent call last)

```

<ipython-input-29-5f727cb3c2c3> in <module>
      2 query6 = tests[6]['input']['query']
      3
----> 4 locate_card(cards6, query6)

<ipython-input-28-aabbbc74d5cf> in locate_card(cards, query)
      8     print('position:', position)
      9
----> 10     if cards[position] == query:

```

```
11         return position
12
```

IndexError: list index out of range

Clearly, since `cards` is empty, it's not possible to access the element at index 0. To fix this, we can check whether we've reached the end of the array before trying to access an element from it. In fact, this can be terminating condition for the `while` loop itself.

```
def locate_card(cards, query):
    position = 0
    while position < len(cards):
        if cards[position] == query:
            return position
        position += 1
    return -1
```

Let's test the failing case again.

```
tests[6]
```

```
{'input': {'cards': [], 'query': 7}, 'output': -1}
```

```
locate_card(cards6, query6)
```

```
-1
```

The result now matches the expected output. Do you now see the benefit of listing test cases beforehand? Without a good set of test cases, we may never have discovered this error in our function.

Let's verify that all the other test cases pass too.

```
evaluate_test_cases(locate_card, tests)
```

TEST CASE #0

Input:

```
{'cards': [13, 11, 10, 7, 4, 3, 1, 0], 'query': 7}
```

Expected Output:

```
3
```

Actual Output:

```
3
```

Execution Time:

0.004 ms

Test Result:

PASSED

TEST CASE #1

Input:

{'cards': [13, 11, 10, 7, 4, 3, 1, 0], 'query': 1}

Expected Output:

6

Actual Output:

6

Execution Time:

0.006 ms

Test Result:

PASSED

TEST CASE #2

Input:

{'cards': [4, 2, 1, -1], 'query': 4}

Expected Output:

0

Actual Output:

0

Execution Time:

0.002 ms

Test Result:

PASSED

TEST CASE #3

Input:

```
{'cards': [3, -1, -9, -127], 'query': -127}
```

Expected Output:

3

Actual Output:

3

Execution Time:

0.003 ms

Test Result:

PASSED

TEST CASE #4

Input:

```
{'cards': [6], 'query': 6}
```

Expected Output:

0

Actual Output:

0

Execution Time:

0.002 ms

Test Result:

PASSED

TEST CASE #5

Input:

```
{'cards': [9, 7, 5, 2, -9], 'query': 4}
```

Expected Output:

-1

Actual Output:

-1

Execution Time:

0.003 ms

Test Result:

PASSED

TEST CASE #6

Input:

```
{'cards': [], 'query': 7}
```

Expected Output:

-1

Actual Output:

-1

Execution Time:

0.004 ms

Test Result:

PASSED

TEST CASE #7

Input:

```
{'cards': [8, 8, 6, 6, 6, 6, 6, 3, 2, 2, 2, 0, 0, 0], 'query': 3}
```

Expected Output:

7

Actual Output:

7

Execution Time:

0.004 ms

Test Result:

PASSED

TEST CASE #8

Input:

```
{'cards': [8, 8, 6, 6, 6, 6, 6, 6, 6, 3, 2, 2, 2, 0, 0, 0], 'query': 6}
```

Expected Output:

2

Actual Output:

2

Execution Time:

0.002 ms

Test Result:

PASSED

SUMMARY

TOTAL: 9, PASSED: 9, FAILED: 0

```
[(3, True, 0.004),  
(6, True, 0.006),  
(0, True, 0.002),  
(3, True, 0.003),  
(0, True, 0.002),  
(-1, True, 0.003),  
(-1, True, 0.004),  
(7, True, 0.004),  
(2, True, 0.002)]
```

Our code passes all the test cases. Of course, there might be some other edge cases we haven't thought of which may cause the function to fail. Can you think of any?

Tip: In a real interview or coding assessment, you can skip the step of implementing and testing the brute force solution in the interest of time. It's generally quite easy to figure out the complexity of the brute force solution from the plain English description.

5. Analyze the algorithm's complexity and identify inefficiencies, if any.

Recall this statement from original question: "Alice challenges Bob to pick out the card containing a given number by turning over as few cards as possible." We restated this requirement as: "Minimize the number of times we access elements from the list cards "



Before we can minimize the number, we need a way to measure it. Since we access a list element once in every iteration, for a list of size N we access the elements from the list up to N times. Thus, Bob may need to overturn up to N cards in the worst case, to find the required card.

Suppose he is only allowed to overturn 1 card per minute, it may take him 30 minutes to find the required card if 30 cards are laid out on the table. Is this the best he can do? Is a way for Bob to arrive at the answer by turning over just 5 cards, instead of 30?

The field of study concerned with finding the amount of time, space or other resources required to complete the execution of computer programs is called *the analysis of algorithms*. And the process of figuring out the best algorithm to solve a given problem is called *algorithm design and optimization*.

Complexity and Big O Notation

Complexity of an algorithm is a measure of the amount of time and/or space required by an algorithm for an input of a given size e.g. N . Unless otherwise stated, the term *complexity* always refers to the worst-case complexity (i.e. the highest possible time/space taken by the program/algorithm to process an input).

In the case of linear search:

1. The *time complexity* of the algorithm is cN for some fixed constant c that depends on the number of operations we perform in each iteration and the time taken to execute a statement. Time complexity is sometimes also called the *running time* of the algorithm.
2. The *space complexity* is some constant c' (independent of N), since we just need a single variable position to iterate through the array, and it occupies a constant space in the computer's memory (RAM).

Big O Notation: Worst-case complexity is often expressed using the Big O notation. In the Big O, we drop fixed constants and lower powers of variables to capture the trend of relationship between the size of the input and the complexity of the algorithm i.e. if the complexity of the algorithm is $cN^3 + dN^2 + eN + f$, in the Big O notation it is expressed as $O(N^3)$

Thus, the time complexity of linear search is $O(N)$ and its space complexity is $O(1)$.

Save and upload your work to Jovian

Whether you're running this Jupyter notebook online or on your computer, it's essential to save your work from time to time. You can continue working on a saved notebook later or share it with friends and colleagues to let them execute your code. [Jovian](#) offers an easy way of saving and sharing your Jupyter notebooks online.

```
!pip install jovian --upgrade --quiet
```

```
import jovian
```

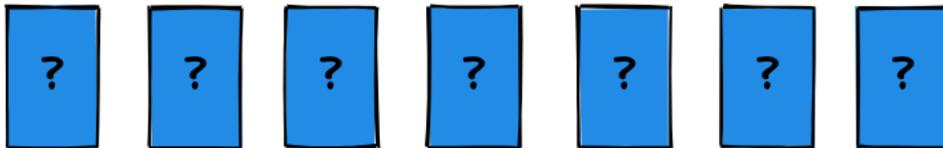
```
jovian.commit(project='python-binary-search', environment=None)
```

```
[jovian] Attempting to save notebook..
```

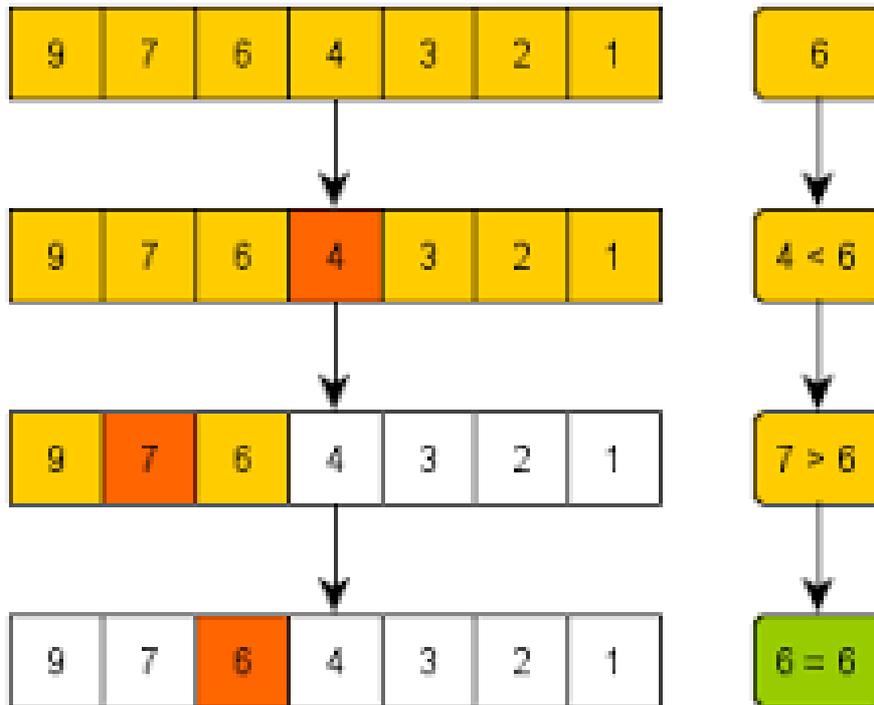
6. Apply the right technique to overcome the inefficiency. Repeat steps 3 to 6.

At the moment, we're simply going over cards one by one, and not even utilizing the fact that they're sorted. This is called a *brute force* approach.

It would be great if Bob could somehow guess the card at the first attempt, but with all the cards turned over it's simply impossible to guess the right card.



The next best idea would be to pick a random card, and use the fact that the list is sorted, to determine whether the target card lies to the left or right of it. In fact, if we pick the middle card, we can reduce the number of additional cards to be tested to half the size of the list. Then, we can simply repeat the process with each half. This technique is called binary search. Here's a visual explanation of the technique:



7. Come up with a correct solution for the problem. State it in plain English.

Here's how binary search can be applied to our problem:

1. Find the middle element of the list.
2. If it matches queried number, return the middle position as the answer.
3. If it is less than the queried number, then search the first half of the list
4. If it is greater than the queried number, then search the second half of the list
5. If no more elements remain, return -1.

```
jovian.commit()
```

```
[jovian] Attempting to save notebook..
```

```
[jovian] Updating notebook "aakashns/python-binary-search" on https://jovian.ai/
```

```
[jovian] Uploading notebook..
```

```
[jovian] Capturing environment..
```

```
[jovian] Committed successfully! https://jovian.ai/aakashns/python-binary-search
```

```
'https://jovian.ai/aakashns/python-binary-search'
```

8. Implement the solution and test it using example inputs. Fix bugs, if any.

Here's an implementation of binary search for solving our problem. We also print the relevant variables in each iteration of the while loop.

```
def locate_card(cards, query):
    lo, hi = 0, len(cards) - 1
```

```
while lo <= hi:
    mid = (lo + hi) // 2
    mid_number = cards[mid]

    print("lo:", lo, ", hi:", hi, ", mid:", mid, ", mid_number:", mid_number)

    if mid_number == query:
        return mid
    elif mid_number < query:
        hi = mid - 1
    elif mid_number > query:
        lo = mid + 1

return -1
```

Let's test it out using the test cases.

```
evaluate_test_cases(locate_card, tests)
```

TEST CASE #0

lo: 0 , hi: 7 , mid: 3 , mid_number: 7

Input:

```
{'cards': [13, 11, 10, 7, 4, 3, 1, 0], 'query': 7}
```

Expected Output:

3

Actual Output:

3

Execution Time:

0.884 ms

Test Result:

PASSED

TEST CASE #1

lo: 0 , hi: 7 , mid: 3 , mid_number: 7

lo: 4 , hi: 7 , mid: 5 , mid_number: 3

lo: 6 , hi: 7 , mid: 6 , mid_number: 1

Input:

```
{'cards': [13, 11, 10, 7, 4, 3, 1, 0], 'query': 1}
```

Expected Output:

6

Actual Output:

6

Execution Time:

0.541 ms

Test Result:

PASSED

TEST CASE #2

lo: 0 , hi: 3 , mid: 1 , mid_number: 2

lo: 0 , hi: 0 , mid: 0 , mid_number: 4

Input:

```
{'cards': [4, 2, 1, -1], 'query': 4}
```

Expected Output:

0

Actual Output:

0

Execution Time:

0.489 ms

Test Result:

PASSED

TEST CASE #3

lo: 0 , hi: 3 , mid: 1 , mid_number: -1

lo: 2 , hi: 3 , mid: 2 , mid_number: -9

lo: 3 , hi: 3 , mid: 3 , mid_number: -127

Input:

```
{'cards': [3, -1, -9, -127], 'query': -127}
```

Expected Output:

3

Actual Output:

3

Execution Time:

0.727 ms

Test Result:

PASSED

TEST CASE #4

lo: 0 , hi: 0 , mid: 0 , mid_number: 6

Input:

```
{'cards': [6], 'query': 6}
```

Expected Output:

0

Actual Output:

0

Execution Time:

0.12 ms

Test Result:

PASSED

TEST CASE #5

lo: 0 , hi: 4 , mid: 2 , mid_number: 5

lo: 3 , hi: 4 , mid: 3 , mid_number: 2

Input:

```
{'cards': [9, 7, 5, 2, -9], 'query': 4}
```

Expected Output:

-1

Actual Output:

-1

Execution Time:

0.382 ms

Test Result:

PASSED

TEST CASE #6

Input:

{'cards': [], 'query': 7}

Expected Output:

-1

Actual Output:

-1

Execution Time:

0.002 ms

Test Result:

PASSED

TEST CASE #7

lo: 0 , hi: 13 , mid: 6 , mid_number: 6

lo: 7 , hi: 13 , mid: 10 , mid_number: 2

lo: 7 , hi: 9 , mid: 8 , mid_number: 2

lo: 7 , hi: 7 , mid: 7 , mid_number: 3

Input:

{'cards': [8, 8, 6, 6, 6, 6, 6, 3, 2, 2, 2, 0, 0, 0], 'query': 3}

Expected Output:

7

Actual Output:

7

Execution Time:

0.626 ms

Test Result:

PASSED

TEST CASE #8

lo: 0 , hi: 14 , mid: 7 , mid_number: 6

Input:

{'cards': [8, 8, 6, 6, 6, 6, 6, 6, 3, 2, 2, 2, 0, 0, 0], 'query': 6}

Expected Output:

2

Actual Output:

7

Execution Time:

0.106 ms

Test Result:

FAILED

SUMMARY

TOTAL: 9, PASSED: 8, FAILED: 1

[(3, True, 0.884),
(6, True, 0.541),
(0, True, 0.489),
(3, True, 0.727),
(0, True, 0.12),
(-1, True, 0.382),
(-1, True, 0.002),

```
(7, True, 0.626),  
(7, False, 0.106)]
```

Looks like it passed 8 out of 9 tests! Let's look at the failed test.

```
evaluate_test_case(locate_card, tests[8])
```

Input:

```
{'cards': [8, 8, 6, 6, 6, 6, 6, 6, 3, 2, 2, 2, 0, 0, 0], 'query': 6}
```

Expected Output:

```
2
```

```
lo: 0 , hi: 14 , mid: 7 , mid_number: 6
```

Actual Output:

```
7
```

Execution Time:

```
0.341 ms
```

Test Result:

```
FAILED
```

```
(7, False, 0.341)
```

Seems like our function returned the position 7 . Let's check what lies at this position in the input list.

```
cards8 = tests[8]['input']['cards']  
query8 = tests[8]['input']['query']
```

```
cards8[7]
```

```
6
```

Seems like we did locate a 6 in the array, it's just that it wasn't the first 6. As you can guess, this is because in binary search, we don't go over indices in a linear order.

So how do we fix it?

When we find that `cards[mid]` is equal to `query` , we need to check whether it is the first occurrence of `query` in the list i.e the number that comes before it.

```
[8, 8, 6, 6, 6, 6, 6, 6, 3, 2, 2, 2, 0, 0, 0]
```

To make it easier, we'll define a helper function called `test_location` , which will take the list `cards` , the `query` and `mid` as inputs.

```

def test_location(cards, query, mid):
    mid_number = cards[mid]
    print("mid:", mid, ", mid_number:", mid_number)
    if mid_number == query:
        if mid-1 >= 0 and cards[mid-1] == query:
            return 'left'
        else:
            return 'found'
    elif mid_number < query:
        return 'left'
    else:
        return 'right'

def locate_card(cards, query):
    lo, hi = 0, len(cards) - 1

    while lo <= hi:
        print("lo:", lo, ", hi:", hi)
        mid = (lo + hi) // 2
        result = test_location(cards, query, mid)

        if result == 'found':
            return mid
        elif result == 'left':
            hi = mid - 1
        elif result == 'right':
            lo = mid + 1
    return -1

```

```

evaluate_test_case(locate_card, tests[8])

```

Input:

```
{'cards': [8, 8, 6, 6, 6, 6, 6, 6, 3, 2, 2, 2, 0, 0, 0], 'query': 6}
```

Expected Output:

2

lo: 0 , hi: 14

mid: 7 , mid_number: 6

lo: 0 , hi: 6

mid: 3 , mid_number: 6

lo: 0 , hi: 2

mid: 1 , mid_number: 8

lo: 2 , hi: 2

mid: 2 , mid_number: 6

Actual Output:

2

Execution Time:

0.969 ms

Test Result:

PASSED

(2, True, 0.969)

```
evaluate_test_cases(locate_card, tests)
```

TEST CASE #0

lo: 0 , hi: 7

mid: 3 , mid_number: 7

Input:

{'cards': [13, 11, 10, 7, 4, 3, 1, 0], 'query': 7}

Expected Output:

3

Actual Output:

3

Execution Time:

0.224 ms

Test Result:

PASSED

TEST CASE #1

lo: 0 , hi: 7

mid: 3 , mid_number: 7

lo: 4 , hi: 7

mid: 5 , mid_number: 3

lo: 6 , hi: 7

mid: 6 , mid_number: 1

Input:

```
{'cards': [13, 11, 10, 7, 4, 3, 1, 0], 'query': 1}
```

Expected Output:

6

Actual Output:

6

Execution Time:

0.581 ms

Test Result:

PASSED

TEST CASE #2

lo: 0 , hi: 3

mid: 1 , mid_number: 2

lo: 0 , hi: 0

mid: 0 , mid_number: 4

Input:

```
{'cards': [4, 2, 1, -1], 'query': 4}
```

Expected Output:

0

Actual Output:

0

Execution Time:

0.313 ms

Test Result:

PASSED

TEST CASE #3

lo: 0 , hi: 3

mid: 1 , mid_number: -1

lo: 2 , hi: 3
mid: 2 , mid_number: -9
lo: 3 , hi: 3
mid: 3 , mid_number: -127

Input:

```
{'cards': [3, -1, -9, -127], 'query': -127}
```

Expected Output:

3

Actual Output:

3

Execution Time:

47.893 ms

Test Result:

PASSED

TEST CASE #4

lo: 0 , hi: 0
mid: 0 , mid_number: 6

Input:

```
{'cards': [6], 'query': 6}
```

Expected Output:

0

Actual Output:

0

Execution Time:

0.13 ms

Test Result:

PASSED

TEST CASE #5

lo: 0 , hi: 4

mid: 2 , mid_number: 5

lo: 3 , hi: 4

mid: 3 , mid_number: 2

Input:

```
{'cards': [9, 7, 5, 2, -9], 'query': 4}
```

Expected Output:

-1

Actual Output:

-1

Execution Time:

0.319 ms

Test Result:

PASSED

TEST CASE #6

Input:

```
{'cards': [], 'query': 7}
```

Expected Output:

-1

Actual Output:

-1

Execution Time:

0.002 ms

Test Result:

PASSED

TEST CASE #7

lo: 0 , hi: 13
mid: 6 , mid_number: 6
lo: 7 , hi: 13
mid: 10 , mid_number: 2
lo: 7 , hi: 9
mid: 8 , mid_number: 2
lo: 7 , hi: 7
mid: 7 , mid_number: 3

Input:

```
{'cards': [8, 8, 6, 6, 6, 6, 6, 3, 2, 2, 2, 0, 0, 0], 'query': 3}
```

Expected Output:

7

Actual Output:

7

Execution Time:

0.684 ms

Test Result:

PASSED

TEST CASE #8

lo: 0 , hi: 14
mid: 7 , mid_number: 6
lo: 0 , hi: 6
mid: 3 , mid_number: 6
lo: 0 , hi: 2
mid: 1 , mid_number: 8
lo: 2 , hi: 2
mid: 2 , mid_number: 6

Input:

```
{'cards': [8, 8, 6, 6, 6, 6, 6, 6, 3, 2, 2, 2, 0, 0, 0], 'query': 6}
```

Expected Output:

2

Actual Output:

2

Execution Time:

0.539 ms

Test Result:

PASSED

SUMMARY

TOTAL: 9, PASSED: 9, FAILED: 0

```
[(3, True, 0.224),  
(6, True, 0.581),  
(0, True, 0.313),  
(3, True, 47.893),  
(0, True, 0.13),  
(-1, True, 0.319),  
(-1, True, 0.002),  
(7, True, 0.684),  
(2, True, 0.539)]
```

In fact, once we have written out the algorithm, we may want to add a few more test cases:

1. The number lies in first half of the array.
2. The number lies in the second half of the array.

Here is the final code for the algorithm (without the print statements):

```
def test_location(cards, query, mid):  
    if cards[mid] == query:  
        if mid-1 >= 0 and cards[mid-1] == query:  
            return 'left'  
        else:  
            return 'found'  
    elif cards[mid] < query:  
        return 'left'  
    else:  
        return 'right'  
  
def locate_card(cards, query):  
    lo, hi = 0, len(cards) - 1  
    while lo <= hi:  
        mid = (lo + hi) // 2  
        result = test_location(cards, query, mid)  
        if result == 'found':
```

```
        return mid
    elif result == 'left':
        hi = mid - 1
    elif result == 'right':
        lo = mid + 1
return -1
```

Try creating a few more test cases to test the algorithm more extensively.

Let's save our work before continuing.

```
jovian.commit()
```

```
[jovian] Attempting to save notebook..
```

```
[jovian] Updating notebook "aakashns/python-binary-search" on https://jovian.ai/
```

```
[jovian] Uploading notebook..
```

```
[jovian] Capturing environment..
```

```
[jovian] Committed successfully! https://jovian.ai/aakashns/python-binary-search
```

```
'https://jovian.ai/aakashns/python-binary-search'
```

9. Analyze the algorithm's complexity and identify inefficiencies, if any.

Once again, let's try to count the number of iterations in the algorithm. If we start out with an array of N elements, then each time the size of the array reduces to half for the next iteration, until we are left with just 1 element.

Initial length - N

Iteration 1 - $N/2$

Iteration 2 - $N/4$ i.e. $N/2^2$

Iteration 3 - $N/8$ i.e. $N/2^3$

...

Iteration k - $N/2^k$

Since the final length of the array is 1, we can find the

$$N/2^k = 1$$

Rearranging the terms, we get

$$N = 2^k$$

Taking the logarithm

$$k = \log N$$

Where \log refers to log to the base 2. Therefore, our algorithm has the time complexity $O(\log N)$. This fact is often stated as: binary search *runs* in logarithmic time. You can verify that the space complexity of binary search is $O(1)$.

Binary Search vs. Linear Search

```
def locate_card_linear(cards, query):
    position = 0
    while position < len(cards):
        if cards[position] == query:
            return position
        position += 1
    return -1
```

```
large_test = {
    'input': {
        'cards': list(range(10000000, 0, -1)),
        'query': 2
    },
    'output': 9999998
}
```

```
result, passed, runtime = evaluate_test_case(locate_card_linear, large_test, display=False)
print("Result: {}\nPassed: {}\nExecution Time: {} ms".format(result, passed, runtime))
```

Result: 9999998

Passed: True

Execution Time: 1103.3 ms

```
result, passed, runtime = evaluate_test_case(locate_card, large_test, display=False)
print("Result: {}\nPassed: {}\nExecution Time: {} ms".format(result, passed, runtime))
```

Result: 9999998

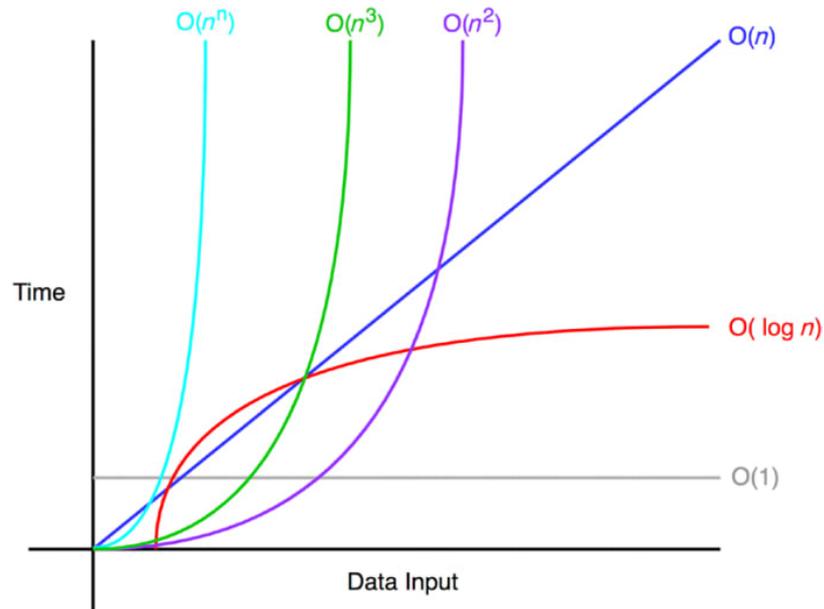
Passed: True

Execution Time: 0.019 ms

The binary search version is over 55,000 times faster than the linear search version.

Furthermore, as the size of the input grows larger, the difference only gets bigger. For a list 10 times the size, linear search would run for 10 times longer, whereas binary search would only require 3 additional operations! (can you verify this?) That's the real difference between the complexities $O(N)$ and $O(\log N)$.

Another way to look at it is that binary search runs $c * N / \log N$ times faster than linear search, for some fixed constant c . Since $\log N$ grows very slowly compared to N , the difference gets larger with the size of the input. Here's a graph showing how the comparing common functions for running time of algorithms ([source](#)):



Do you see now why we ignore constants and lower order terms while expressing the complexity using the Big O notation?

Generic Binary Search

Here is the general strategy behind binary search, which is applicable to a variety of problems:

1. Come up with a condition to determine whether the answer lies before, after or at a given position
2. Retrieve the midpoint and the middle element of the list.
3. If it is the answer, return the middle position as the answer.
4. If answer lies before it, repeat the search with the first half of the list
5. If the answer lies after it, repeat the search with the second half of the list.

Here is the generic algorithm for binary search, implemented in Python:

```
def binary_search(lo, hi, condition):
    """TODO - add docs"""
    while lo <= hi:
        mid = (lo + hi) // 2
        result = condition(mid)
        if result == 'found':
            return mid
        elif result == 'left':
            hi = mid - 1
        else:
            lo = mid + 1
    return -1
```

The worst-case complexity or running time of binary search is $O(\log N)$, provided the complexity of the condition used to determine whether the answer lies before, after or at a given position is $O(1)$.

Note that `binary_search` accepts a function `condition` as an argument. Python allows passing functions as arguments to other functions, unlike C++ and Java.

We can now rewrite the `locate_card` function more succinctly using the `binary_search` function.

```
def locate_card(cards, query):  
  
    def condition(mid):  
        if cards[mid] == query:  
            if mid > 0 and cards[mid-1] == query:  
                return 'left'  
            else:  
                return 'found'  
        elif cards[mid] < query:  
            return 'left'  
        else:  
            return 'right'  
  
    return binary_search(0, len(cards) - 1, condition)
```

Note here that we have defined a function within a function, another handy feature in Python. And the inner function can access the variables within the outer function.

```
evaluate_test_cases(locate_card, tests)
```

TEST CASE #0

Input:

```
{'cards': [13, 11, 10, 7, 4, 3, 1, 0], 'query': 7}
```

Expected Output:

3

Actual Output:

3

Execution Time:

0.006 ms

Test Result:

PASSED

TEST CASE #1

Input:

```
{'cards': [13, 11, 10, 7, 4, 3, 1, 0], 'query': 1}
```

Expected Output:

6

Actual Output:

6

Execution Time:

0.006 ms

Test Result:

PASSED

TEST CASE #2

Input:

```
{'cards': [4, 2, 1, -1], 'query': 4}
```

Expected Output:

0

Actual Output:

0

Execution Time:

0.005 ms

Test Result:

PASSED

TEST CASE #3

Input:

```
{'cards': [3, -1, -9, -127], 'query': -127}
```

Expected Output:

3

Actual Output:

3

Execution Time:

0.008 ms

Test Result:

PASSED

TEST CASE #4

Input:

{'cards': [6], 'query': 6}

Expected Output:

0

Actual Output:

0

Execution Time:

0.003 ms

Test Result:

PASSED

TEST CASE #5

Input:

{'cards': [9, 7, 5, 2, -9], 'query': 4}

Expected Output:

-1

Actual Output:

-1

Execution Time:

0.004 ms

Test Result:

PASSED

TEST CASE #6

Input:

```
{'cards': [], 'query': 7}
```

Expected Output:

-1

Actual Output:

-1

Execution Time:

0.005 ms

Test Result:

PASSED

TEST CASE #7

Input:

```
{'cards': [8, 8, 6, 6, 6, 6, 6, 3, 2, 2, 2, 0, 0, 0], 'query': 3}
```

Expected Output:

7

Actual Output:

7

Execution Time:

0.006 ms

Test Result:

PASSED

TEST CASE #8

Input:

```
{'cards': [8, 8, 6, 6, 6, 6, 6, 6, 3, 2, 2, 2, 0, 0, 0], 'query': 6}
```

Expected Output:

2

Actual Output:

2

Execution Time:

0.005 ms

Test Result:

PASSED

SUMMARY

TOTAL: 9, PASSED: 9, FAILED: 0

```
[(3, True, 0.006),  
(6, True, 0.006),  
(0, True, 0.005),  
(3, True, 0.008),  
(0, True, 0.003),  
(-1, True, 0.004),  
(-1, True, 0.005),  
(7, True, 0.006),  
(2, True, 0.005)]
```

The `binary_search` function can now be used to solve other problems too. It is a tested piece of logic.

Question: Given an array of integers `nums` sorted in ascending order, find the starting and ending position of a given number.

This differs from the problem in only two significant ways:

1. The numbers are sorted in increasing order.
2. We are looking for both the increasing order and the decreasing order.

Here's the full code for solving the question, obtained by making minor modifications to our previous function:

```
def first_position(nums, target):  
    def condition(mid):
```

```

    if nums[mid] == target:
        if mid > 0 and nums[mid-1] == target:
            return 'left'
        return 'found'
    elif nums[mid] < target:
        return 'right'
    else:
        return 'left'
return binary_search(0, len(nums)-1, condition)

def last_position(nums, target):
    def condition(mid):
        if nums[mid] == target:
            if mid < len(nums)-1 and nums[mid+1] == target:
                return 'right'
            return 'found'
        elif nums[mid] < target:
            return 'right'
        else:
            return 'left'
    return binary_search(0, len(nums)-1, condition)

def first_and_last_position(nums, target):
    return first_position(nums, target), last_position(nums, target)

```

We can test our solution by making a submission here: <https://leetcode.com/problems/find-first-and-last-position-of-element-in-sorted-array/>

The Method - Revisited

Here's a systematic strategy we've applied for solving the problem:

1. State the problem clearly. Identify the input & output formats.
2. Come up with some example inputs & outputs. Try to cover all edge cases.
3. Come up with a correct solution for the problem. State it in plain English.
4. Implement the solution and test it using example inputs. Fix bugs, if any.
5. Analyze the algorithm's complexity and identify inefficiencies, if any.
6. Apply the right technique to overcome the inefficiency. Repeat steps 3 to 6.

Use this template for solving problems using this method: <https://jovian.ai/aakashns/python-problem-solving-template>

This seemingly obvious strategy will help you solve almost any programming problem you will face in an interview or coding assessment.

The objective of this course is to rewire your brain to think using this method, by applying it over and over to different types of problems. This is a course about thinking about problems systematically and turning those thoughts into code.

Problems for Practice

Here are some resources to learn more and find problems to practice.

- Assignment on Binary Search: <https://jovian.ai/aakashns/python-binary-search-assignment>
- Binary Search Problems on LeetCode: <https://leetcode.com/problems/binary-search/>
- Binary Search Problems on GeeksForGeeks: <https://www.geeksforgeeks.org/binary-search/>
- Binary Search Problems on Codeforces: <https://codeforces.com/problemset?tags=binary+search>

Use this template for solving problems: <https://jovian.ai/aakashns/python-problem-solving-template>

Start a discussion on the forum: <https://jovian.ai/forum/c/data-structures-and-algorithms-in-python/lesson-1-binary-search-linked-lists-and-complex/81>

Try to solve at least 5-10 problems over the week to master binary search.

```
jovian.commit()
```

```
[jovian] Attempting to save notebook..
```

```
[jovian] Updating notebook "aakashns/python-binary-search" on https://jovian.ai/
```

```
[jovian] Uploading notebook..
```

```
[jovian] Capturing environment..
```

```
[jovian] Committed successfully! https://jovian.ai/aakashns/python-binary-search
```

```
'https://jovian.ai/aakashns/python-binary-search'
```

Assignment 1 - Binary Search Practice

This assignment is a part of the course ["Data Structures and Algorithms in Python"](#).

In this assignment, you'll get to practice some of the concepts and skills covered in the following notebooks:

1. [Binary Search and Complexity Analysis](#)
2. [Solving Programming Problems Systematically](#)

As you go through this notebook, you will find a ??? in certain places. To complete this assignment, you must replace all the ??? with appropriate values, expressions or statements to ensure that the notebook runs properly end-to-end.

Some things to keep in mind:

- Make sure to run all the code cells, otherwise you may get errors like `NameError` for undefined variables.
- Do not change variable names, delete cells or disturb other existing code. It may cause problems during evaluation.
- In some cases, you may need to add some code cells or new statements before or after the line of code containing the ???.
- Since you'll be using a temporary online service for code execution, save your work by running `jovian.commit` at regular intervals.
- Questions marked **(Optional)** will not be considered for evaluation, and can be skipped. They are for your learning.

You can make submissions on this page: <https://jovian.ai/learn/data-structures-and-algorithms-in-python/assignment/assignment-1-binary-search-practice>

If you are stuck, you can ask for help on the community forum: <https://jovian.ai/forum/c/data-structures-and-algorithms-in-python/assignment-1/87>. You can get help with errors or ask for hints, but **please don't ask for OR share the full working answer code** on the forum.

How to run the code and save your work

The recommended way to run this notebook is to click the "Run" button at the top of this page, and select "Run on Binder". This will run the notebook on mybinder.org, a free online service for running Jupyter notebooks.

This tutorial is an executable [Jupyter notebook](#). You can *run* this tutorial and experiment with the code examples in a couple of ways: *using free online resources* (recommended) or *on your computer*.

Option 1: Running using free online resources (1-click, recommended)

The easiest way to start executing the code is to click the **Run** button at the top of this page and select **Run on Binder**. You can also select "Run on Colab" or "Run on Kaggle", but you'll need to create an account on [Google Colab](#) or [Kaggle](#) to use these platforms.

Option 2: Running on your computer locally

To run the code on your computer locally, you'll need to set up [Python](#), download the notebook and install the required libraries. We recommend using the [Conda](#) distribution of Python. Click the **Run** button at the top of this

page, select the **Run Locally** option, and follow the instructions.

Saving your work

Before starting the assignment, let's save a snapshot of the assignment to your [Jovian](#) profile, so that you can access it later, and continue your work.

```
!pip install jovian --upgrade --quiet
```

```
import jovian
```

```
project='python-binary-search-assignment'
```

```
jovian.commit(project=project, privacy='secret', environment=None)
```

```
[jovian] Updating notebook "evanmarie/python-binary-search-assignment" on  
https://jovian.ai
```

```
[jovian] Committed successfully! https://jovian.ai/evanmarie/python-binary-search-assignment
```

```
'https://jovian.ai/evanmarie/python-binary-search-assignment'
```

You may be asked to [provide an API Key](#) to upload your notebook. The privacy of your assignment notebook is set to "Secret", so that you can the evaluators can access it, but it will not shown on your public profile to other users.

To continue working on a saved assignment, just visit [your profile](#) and run the saved notebook again.

Problem - Rotated Lists

We'll solve the following problem step-by-step:

You are given list of numbers, obtained by rotating a sorted list an unknown number of times. Write a function to determine the minimum number of times the original sorted list was rotated to obtain the given list. Your function should have the worst-case complexity of $O(\log N)$, where N is the length of the list. You can assume that all the numbers in the list are unique.

Example: The list `[5, 6, 9, 0, 2, 3, 4]` was obtained by rotating the sorted list `[0, 2, 3, 4, 5, 6, 9]` 3 times.

We define "rotating a list" as removing the last element of the list and adding it before the first element. E.g. rotating the list `[3, 2, 4, 1]` produces `[1, 3, 2, 4]`.

"Sorted list" refers to a list where the elements are arranged in the increasing order e.g. `[1, 3, 5, 7]`.

The Method

Here's the systematic strategy we'll apply for solving problems:

1. State the problem clearly. Identify the input & output formats.

2. Come up with some example inputs & outputs. Try to cover all edge cases.
3. Come up with a correct solution for the problem. State it in plain English.
4. Implement the solution and test it using example inputs. Fix bugs, if any.
5. Analyze the algorithm's complexity and identify inefficiencies, if any.
6. Apply the right technique to overcome the inefficiency. Repeat steps 3 to 6.

This approach is explained in detail in [Lesson 1](#) of the course. Let's apply this approach step-by-step.

Solution

1. State the problem clearly. Identify the input & output formats.

While this problem is stated clearly enough, it's always useful to try and express in your own words, in a way that makes it most clear for you. It's perfectly OK if your description overlaps with the original problem statement to a large extent.

Q: Express the problem in your own words below (to edit this cell, double click on it).

Problem

- `nums` is a list of integers, which at one point were a nicely organized, sorted ascending list of integers. * However, `nums` has had its sorted, ascending order skewed by having been rotated an unknown number of times. * The rotation was performed by removing the final digit from the end of the list and placing at the beginning of the list. * Determine how many times the list had this rotation performed on it to arrive at the current state, i.e. the state it is presented for this problem.

Q: The function you write will take one input called `nums` . What does it represent? Give an example.

Input

1. `nums`: the list presented for the problem that was once a sorted list in ascending order, now having been rotated, last digit moved to `nums[0]` an unknown number of rotations. EX: if `nums` was once `[1, 2, 3, 4, 5, 6]`, the problem may be presented with it as `[4, 5, 6, 1, 2, 3]`, in which case, `nums` has gone through 3 rotations, so that 6, then 5, then 4 were rotated from the end of `nums` to `nums[0]` position.

Q: The function you write will return a single output called `rotations` . What does it represent? Give an example.

Output

3. `rotations`: the number of times `nums` had a rotation performed upon it, i.e. however many digits come before the lowest digit in the list. Therefore if `nums` was `[4, 5, 6, 1, 2, 3]`, it is clear that there are 3 digits which come before the lowest digit in the list. And since `nums` was once in sorted, ascending order with the lowest digit coming first, it has undergone 3 rotations to arrive at its current state.

Based on the above, we can now create a signature of our function:

```
def count_rotations(nums): # I was silly and thought I was supposed to do the binary search

    # If the list has not been rotated at all.
    if len(nums) == 0 or nums[0] <= nums[-1]:
        return 0

    # Pivot refers to the first number in nums, to be compared to find the lowest
    # integer in the list, which will be the start of the original, sorted nums
    pivot = nums[0]

    # Start of list to be searched is nums[0] at the beginning of the function
    def find_origin(mini_nums, start = 0):
        # Specify the middle of the portion of the list to be searched.
        # mini_nums is the list or sub-list currently being searched.
        mid = len(mini_nums) // 2

        # If the mid of mini_nums is less than the integer at the start
        # of the portion of the list currently being searched.
        if mini_nums[mid] < pivot:
            # If the midpoint = 0, there is only 1 integer in the list, return it.
            # It would thereby be zero, and there would have been no rotations.
            if mid == 0:
                return start

            # If the integer to the right of mid is less than the pivot, return
            # the index of mid + start, which = the number of rotations
            if mini_nums[mid-1] > pivot:
                return mid+start

            else:
                # If we have not yet found our rotation count and mid is less than
                # pivot, the lowest is to the left, so repeat algorithm on left half
                return find_origin(mini_nums[:mid], start = start)

        # The search has led all the way to the beginning of the list where only the
        # integer to the right of mid can be the origin. Return the number of rotations
        if mid == 0:
            return start + 1

        # If the midpoint is greater than the pivot, the low point lies to the right
        # half of the list, repeat the algorithm there.
        return find_origin(mini_nums[mid+1:], start = start+mid+1)

    # This is the beginning...even though it is at the end. Alice in Wonderland much?
    return find_origin(nums)
```

After each, step remember to save your notebook

```
jovian.commit(project=project)
```

```
[jovian] Updating notebook "evanmarie/python-binary-search-assignment" on  
https://jovian.ai
```

```
[jovian] Committed successfully! https://jovian.ai/evanmarie/python-binary-search-assignment
```

```
'https://jovian.ai/evanmarie/python-binary-search-assignment'
```

2. Come up with some example inputs & outputs. Try to cover all edge cases.

Our function should be able to handle any set of valid inputs we pass into it. Here's a list of some possible variations we might encounter:

1. A list of size 10 rotated 3 times.
2. A list of size 8 rotated 5 times.
3. A list that wasn't rotated at all.
4. A list that was rotated just once.
5. A list that was rotated $n-1$ times, where n is the size of the list.
6. A list that was rotated n times (do you get back the original list here?)
7. An empty list.
8. A list containing just one element.
9. (can you think of any more?)

We'll express our test cases as dictionaries, to test them easily. Each dictionary will contain 2 keys: `input` (a dictionary itself containing one key for each argument to the function and `output` (the expected result from the function). Here's an example.

```
test = {  
    'input': {  
        'nums': [19, 25, 29, 3, 5, 6, 7, 9, 11, 14]  
    },  
    'output': 3  
}
```

We can test the function by passing the input to it directly or by using the `evaluate_test_case` function from `jovian`.

```
nums0 = test['input']['nums']  
output0 = test['input']['output']  
result0 = count_rotations(nums0)  
  
result0, result0 == output0
```

```
(3, False)
```

```
from jovian.pythondsa import evaluate_test_case
```

```
evaluate_test_case(count_rotations, test)
```

Input:

```
{'nums': [19, 25, 29, 3, 5, 6, 7, 9, 11, 14]}
```

Expected Output:

3

Actual Output:

3

Execution Time:

0.008 ms

Test Result:

PASSED

(3, True, 0.008)

Let's create one test case for each of the scenarios listed above. We'll store our test cases in an array called tests .

Q: Create proper test cases for each of the scenarios listed above.

```
test0 = test
```

```
# A list of size 8 rotated 5 times.
test1 = {
    'input': {
        'nums': [4, 5, 6, 7, 8, 1, 2, 3]
    },
    'output': 5
}
```

```
# A list that wasn't rotated at all.
test2 = {
    'input': {
        'nums': [1, 2, 3, 4, 5, 6, 7, 8]
    },
    'output': 0
}
```

A list that was rotated just once. A list that was rotated n-1 times, where n is the size of the list. A list that was rotated n times (do you get back the original list here?) An empty list. A list containing just one element.

```
# A list that was rotated just once.
test3 = {
  'input': {
    'nums': [10, 4, 5, 6, 7, 8, 9]
  },
  'output': 1
}
```

```
# A list that was rotated n-1 times, where n is the size of the list.
test4 = {
  'input': {
    'nums': [2, 3, 4, 5, 6, 7, 8, 1]
  },
  'output': 7
}
```

```
# A list that was rotated n times, where n is the size of the list
test5 = {
  'input': {
    'nums': [1, 2, 3, 4, 5, 6, 7, 8]
  },
  'output': 0
}
```

HINT: Read the question carefully to determine the correct output for the above test case.

```
# An empty list.
test6 = {
  'input': {
    'nums': []
  },
  'output': 0
}
```

```
# A list containing just one element.
test7 = {
  'input': {
    'nums': [3]
  },
  'output': 0
}
```

```
tests = [test0, test1, test2, test3, test3, test5, test6, test7]
```

Q (Optional): Include any further test cases below, for other interesting scenarios you can think of.

```
# A list containing just two elements.
test8 = {
    'input': {
        'nums': [2, 1]
    },
    'output': 1
}
```

Evaluate your function against all the test cases together using the `evaluate_test_cases` (plural) function from `jovian`.

```
from jovian.pythondsa import evaluate_test_cases
```

```
evaluate_test_cases(count_rotations, tests)
```

TEST CASE #0

Input:

```
{'nums': [19, 25, 29, 3, 5, 6, 7, 9, 11, 14]}
```

Expected Output:

3

Actual Output:

3

Execution Time:

0.008 ms

Test Result:

PASSED

TEST CASE #1

Input:

```
{'nums': [4, 5, 6, 7, 8, 1, 2, 3]}
```

Expected Output:

5

Actual Output:

5

Execution Time:

0.006 ms

Test Result:

PASSED

TEST CASE #2

Input:

{'nums': [1, 2, 3, 4, 5, 6, 7, 8]}

Expected Output:

0

Actual Output:

0

Execution Time:

0.002 ms

Test Result:

PASSED

TEST CASE #3

Input:

{'nums': [10, 4, 5, 6, 7, 8, 9]}

Expected Output:

1

Actual Output:

1

Execution Time:

0.004 ms

Test Result:

PASSED

TEST CASE #4

Input:

```
{'nums': [10, 4, 5, 6, 7, 8, 9]}
```

Expected Output:

1

Actual Output:

1

Execution Time:

0.014 ms

Test Result:

PASSED

TEST CASE #5

Input:

```
{'nums': [1, 2, 3, 4, 5, 6, 7, 8]}
```

Expected Output:

0

Actual Output:

0

Execution Time:

0.002 ms

Test Result:

PASSED

TEST CASE #6

Input:

```
{'nums': []}
```

Expected Output:

0

Actual Output:

0

Execution Time:

0.002 ms

Test Result:

PASSED

TEST CASE #7

Input:

```
{'nums': [3]}
```

Expected Output:

0

Actual Output:

0

Execution Time:

0.002 ms

Test Result:

PASSED

SUMMARY

TOTAL: 8, PASSED: 8, FAILED: 0

```
[(3, True, 0.008),
 (5, True, 0.006),
 (0, True, 0.002),
 (1, True, 0.004),
 (1, True, 0.014),
 (0, True, 0.002),
 (0, True, 0.002),
 (0, True, 0.002)]
```

Verify that all the test cases were evaluated. We expect them all to fail, since we haven't implemented the function yet.

Let's save our work before continuing.

```
jovian.commit(project=project)
```

```
[jovian] Updating notebook "evanmarie/python-binary-search-assignment" on
https://jovian.ai
```

```
[jovian] Committed successfully! https://jovian.ai/evanmarie/python-binary-search-assignment
```

```
'https://jovian.ai/evanmarie/python-binary-search-assignment'
```

3. Come up with a correct solution for the problem. State it in plain English.

Our first goal should always be to come up with a *correct* solution to the problem, which may not necessarily be the most *efficient* solution. Try to think of a solution before you read further.

Coming up with the correct solution is quite easy, and it's based on this insight: If a list of sorted numbers is rotated k times, then the smallest number in the list ends up at position k (counting from 0). Further, it is the only number in the list which is smaller than the number before it. Thus, we simply need to **check for each number in the list whether it is smaller than the number that comes before it** (if there is a number before it). Then, our answer i.e. the number of rotations is simply the position of this number is . If we cannot find such a number, then the list wasn't rotated at all.

Example: In the list `[19, 25, 29, 3, 5, 6, 7, 9, 11, 14]`, the number `3` is the only number smaller than its predecessor. It occurs at the position `3` (counting from `0`), hence the array was rotated `3` times.

We can use the *linear search* algorithm as a first attempt to solve this problem i.e. we can perform the check for every position one by one. But first, try describing the above solution in your own words, that make it clear to you.

Q (Optional): Describe the linear search solution explained above problem in your own words.

1. Starting with the second integer in the list, compare it to the first. If the first is smaller than the second, return the first integer's index number.
2. If the first integer is not smaller than the second, move on to the next integer in the list until an integer is found that is smaller than the one before it.
3. When one is found that is smaller than all the numbers before it, that is the number that marks the beginning of the list.
4. Return the index number of the smallest number in the list. That index is equal to the number of times the list was rotated.

(add more steps if required)

Let's save and upload our work before continuing.

```
import jovian
```

```
jovian.commit(project=project)
```

[jovian] Updating notebook "evanmarie/python-binary-search-assignment" on <https://jovian.ai>

[jovian] Committed successfully! <https://jovian.ai/evanmarie/python-binary-search-assignment>

'<https://jovian.ai/evanmarie/python-binary-search-assignment>'

4. Implement the solution and test it using example inputs. Fix bugs, if any.

Q: Implement the solution described in step 3.

```
def count_rotations_linear(nums):
    position = 0          # What is the initial value of position?

    if len(nums) == 0 or len(nums) == 1:
        return 0
    if nums[0] < nums[-1]:
        return 0

    while position <= len(nums): # When should the loop be terminated?

        # Success criteria: check whether the number at the current position is smaller
        if position > 0 and nums[position] < nums[position - 1]: # How to perform the check
            return position

        # Move to the next position
        position += 1

    return None          # What if none of the positions passed the check
```

Let's test out the function with the first test case.

```
linear_search_result = evaluate_test_case(count_rotations_linear, test)
```

Input:

```
{'nums': [19, 25, 29, 3, 5, 6, 7, 9, 11, 14]}
```

Expected Output:

3

Actual Output:

3

Execution Time:

0.004 ms

Test Result:

PASSED

Make sure your function passes the test. Fix bugs, if any.

Let's test it out with all the test cases.

```
linear_search_results = evaluate_test_cases(count_rotations_linear, tests)
```

TEST CASE #0

Input:

```
{'nums': [19, 25, 29, 3, 5, 6, 7, 9, 11, 14]}
```

Expected Output:

3

Actual Output:

3

Execution Time:

0.005 ms

Test Result:

PASSED

TEST CASE #1

Input:

```
{'nums': [4, 5, 6, 7, 8, 1, 2, 3]}
```

Expected Output:

5

Actual Output:

5

Execution Time:

0.005 ms

Test Result:

PASSED

TEST CASE #2

Input:

{'nums': [1, 2, 3, 4, 5, 6, 7, 8]}

Expected Output:

0

Actual Output:

0

Execution Time:

0.002 ms

Test Result:

PASSED

TEST CASE #3

Input:

{'nums': [10, 4, 5, 6, 7, 8, 9]}

Expected Output:

1

Actual Output:

1

Execution Time:

0.003 ms

Test Result:

PASSED

TEST CASE #4

Input:

```
{'nums': [10, 4, 5, 6, 7, 8, 9]}
```

Expected Output:

1

Actual Output:

1

Execution Time:

0.002 ms

Test Result:

PASSED

TEST CASE #5

Input:

```
{'nums': [1, 2, 3, 4, 5, 6, 7, 8]}
```

Expected Output:

0

Actual Output:

0

Execution Time:

0.002 ms

Test Result:

PASSED

TEST CASE #6

Input:

```
{'nums': []}
```

Expected Output:

0

Actual Output:

0

Execution Time:

0.001 ms

Test Result:

PASSED

TEST CASE #7

Input:

```
{'nums': [3]}
```

Expected Output:

0

Actual Output:

0

Execution Time:

0.002 ms

Test Result:

PASSED

SUMMARY

TOTAL: 8, PASSED: 8, FAILED: 0

Once again, make sure all the tests pass. Fix errors and bugs, if any.

NOTE: During evaluation, your submission will be tested against a much larger set of test cases (not listed here). Make sure to test your solution thoroughly.

If you are stuck, you can ask for help on the community forum: <https://jovian.ai/forum/c/data-structures-and-algorithms-in-python/assignment-1/87>. You can get help with errors or ask for hints, but **please don't ask for OR share the full working answer code** on the forum.

5. Analyze the algorithm's complexity and identify inefficiencies, if any.

Count the maximum number of iterations it may take for the algorithm to return the result.

Q: What is the worst-case complexity (running time) of the algorithm expressed in the Big O Notation? Assume that the size of the list is N (uppercase).

```
# linear_search_complexity = O[N]
```

6. Apply the right technique to overcome the inefficiency. Repeat steps 3 to 6.

As you might have guessed, we can apply *Binary Search* to solve this problem. The key question we need to answer in binary search is: Given the middle element, how to decide if it is the answer (smallest number), or whether the answer lies to the left or right of it.

If the middle element is smaller than its predecessor, then it is the answer. However, if it isn't, this check is not sufficient to determine whether the answer lies to the left or the right of it. Consider the following examples.

[7, 8, 1, 3, 4, 5, 6] (answer lies to the left of the middle element)

[1, 2, 3, 4, 5, -1, 0] (answer lies to the right of the middle element)

Here's a check that will help us determine if the answer lies to the left or the right: *If the middle element of the list is smaller than the last element of the range, then the answer lies to the left of it. Otherwise, the answer lies to the right.*

Do you see why this strategy works?

7. Come up with a correct solution for the problem. State it in plain English.

Before we implement the solution, it's useful to describe it in a way that makes most sense to you. In a coding interview, you will almost certainly be asked to describe your approach before you start writing code.

Q (Optional): Describe the binary search solution explained above problem in your own words.

1. Jump to the middle integer in the list. Is it smaller than the one before? If yes, return its index. The goal has been found..
2. If no, compare the middle integer to the last integer of the list. If the mid point is greater than the last number in the list, the answer lies in the right portion of the list.
3. If the middle integer is NOT greater than the integer at the end of the list, the answer lies in the left half of the list.
4. Jump to the middle integer of the side of the list determined to be the one that contains the answer, and continue the binary search in a loop until the answer is discovered, breaking the list down by half each time.

(add more steps if required)

Let's save and upload our work before continuing.

```
jovian.commit(project=project)
```

```
[jovian] Updating notebook "evanmarie/python-binary-search-assignment" on  
https://jovian.ai
```

```
[jovian] Committed successfully! https://jovian.ai/evanmarie/python-binary-search-assignment
```

```
'https://jovian.ai/evanmarie/python-binary-search-assignment'
```

8. Implement the solution and test it using example inputs. Fix bugs, if any.

Q: Implement the binary search solution described in step 7.

If you are stuck, you can ask for help on the community forum: <https://jovian.ai/forum/c/data-structures-and-algorithms-in-python/assignment-1/87>. You can get help with errors or ask for hints, but **please don't ask for OR share the full working answer code** on the forum.

```
def count_rotations_binary(nums):  
  
    if len(nums) == 0 or len(nums) == 1:  
        return 0  
    if nums[0] < nums[-1]:  
        return 0  
  
    lo = 0  
    hi = len(nums) - 1  
  
    while lo <= hi:  
        mid = (lo + hi) // 2  
        mid_number = nums[mid]  
  
        if mid > 0 and mid_number < nums[mid-1]:  
            # The middle position is the answer  
            return mid  
  
        elif nums[mid] < nums[hi]:  
            # Answer lies in the left half  
            hi = mid - 1  
  
        else:  
            # Answer lies in the right half  
            lo = mid + 1  
  
    return 0
```

Let's test out the function with the first test case.

```
binary_search_result = evaluate_test_case(count_rotations_binary, test)
```

Input:

```
{'nums': [19, 25, 29, 3, 5, 6, 7, 9, 11, 14]}
```

Expected Output:

3

Actual Output:

3

Execution Time:

0.005 ms

Test Result:

PASSED

Make sure your function passes the test. Fix bugs, if any.

Let's test it out with all the test cases.

```
binary_search_results = evaluate_test_cases(count_rotations_binary, tests)
```

TEST CASE #0

Input:

```
{'nums': [19, 25, 29, 3, 5, 6, 7, 9, 11, 14]}
```

Expected Output:

3

Actual Output:

3

Execution Time:

0.005 ms

Test Result:

PASSED

TEST CASE #1

Input:

```
{'nums': [4, 5, 6, 7, 8, 1, 2, 3]}
```

Expected Output:

5

Actual Output:

5

Execution Time:

0.003 ms

Test Result:

PASSED

TEST CASE #2

Input:

```
{'nums': [1, 2, 3, 4, 5, 6, 7, 8]}
```

Expected Output:

0

Actual Output:

0

Execution Time:

0.002 ms

Test Result:

PASSED

TEST CASE #3

Input:

```
{'nums': [10, 4, 5, 6, 7, 8, 9]}
```

Expected Output:

1

Actual Output:

1

Execution Time:

0.003 ms

Test Result:

PASSED

TEST CASE #4

Input:

```
{'nums': [10, 4, 5, 6, 7, 8, 9]}
```

Expected Output:

1

Actual Output:

1

Execution Time:

0.002 ms

Test Result:

PASSED

TEST CASE #5

Input:

```
{'nums': [1, 2, 3, 4, 5, 6, 7, 8]}
```

Expected Output:

0

Actual Output:

0

Execution Time:

0.002 ms

Test Result:

PASSED

TEST CASE #6

Input:

{'nums': []}

Expected Output:

0

Actual Output:

0

Execution Time:

0.001 ms

Test Result:

PASSED

TEST CASE #7

Input:

{'nums': [3]}

Expected Output:

0

Actual Output:

0

Execution Time:

0.002 ms

Test Result:

PASSED

SUMMARY

TOTAL: 8, PASSED: 8, FAILED: 0

Once again, make sure all the tests pass. Fix errors and bugs, if any.

NOTE: During evaluation, your submission will be tested against a much larger set of test cases (not listed here). Make sure to test your solution thoroughly.

If you are stuck, you can ask for help on the community forum: <https://jovian.ai/forum/c/data-structures-and-algorithms-in-python/assignment-1/87>. You can get help with errors or ask for hints, but **please don't ask for OR share the full working answer code** on the forum.

Let's save our work before continuing.

```
jovian.commit(project=project)
```

[jovian] Updating notebook "evanmarie/python-binary-search-assignment" on <https://jovian.ai>

[jovian] Committed successfully! <https://jovian.ai/evanmarie/python-binary-search-assignment>

```
'https://jovian.ai/evanmarie/python-binary-search-assignment'
```

9. Analyze the algorithm's complexity and identify inefficiencies, if any.

Q: What is the worst-case complexity (running time) of the algorithm expressed in the Big O Notation? Assume that the size of the list is N (uppercase).

Hint: Count the maximum number of iterations it may take for the algorithm to return the result.

```
# binary_search_complexity = O[log N]
```

Is binary search the optimal solution to the problem? How can you prove it? Discuss in the forums.

Let's save our work before continuing.

```
import jovian
```

```
jovian.commit()
```

[jovian] Updating notebook "evanmarie/python-binary-search-assignment" on <https://jovian.ai>

```
[jovian] Committed successfully! https://jovian.ai/evanmarie/python-binary-search-assignment
```

```
'https://jovian.ai/evanmarie/python-binary-search-assignment'
```

Make a Submission

To make a submission, visit the [assignment page](#) and submit the link to your notebook.

You can also make a submission by executing the following statement:

```
jovian.submit(assignment="pythondsa-assignment1")
```

```
[jovian] Updating notebook "evanmarie/python-binary-search-assignment" on https://jovian.ai
```

```
[jovian] Committed successfully! https://jovian.ai/evanmarie/python-binary-search-assignment
```

```
[jovian] Submitting assignment..
```

```
[jovian] Verify your submission at https://jovian.ai/learn/data-structures-and-algorithms-in-python/assignment/assignment-1-binary-search-practice
```

You can view your previous submissions under the "Submission History" section of the [assignment page](#). Only your last submission will be considered for evaluation.

Bonus Questions

The questions in this section are optional, and will not affect your grade. Discuss the bonus questions here: <https://jovian.ai/forum/t/optional-bonus-questions-discussion-assignment-1/15486>

You can also copy over the bonus questions to a new notebook to share your solution on the forum without sharing your assignment notebook. Duplicate this template: <https://jovian.ai/aakashns/python-problem-solving-template>

Optional Bonus 1: Using the Generic Binary Search Algorithm

The `jovian` library provides a generic implementation of the binary search algorithm.

```
from jovian.pythondsa import binary_search
```

You can view its source code using the `??` command in Jupyter or on [the Github repository](#) for the `jovian` library.

```
??binary_search
```

Q (Optional): Implement the `count_rotations` function using the generic `binary_search` function.

Hint: You'll need to define the condition which returns "found", "left" or "right" by performing the appropriate check on the middle position in the range.

```

def condition(mid, hi, lo, nums):
    mid_number = nums[mid]
    if mid > 0 and mid_number < nums[mid-1]:
        return 'found'
    elif nums[mid] < nums[hi]:
        return 'left'
    else:
        return 'right'
    return count_rotations_generic(0, len(nums) - 1, condition)

def count_rotations_generic(nums):

    if len(nums) == 0 or len(nums) == 1:
        return 0
    if nums[0] < nums[-1]:
        return 0

    lo = 0
    hi = len(nums) - 1

    while lo <= hi:
        mid = (lo + hi) // 2
        result = condition(mid, lo, hi, nums)

        if result == 'found':
            return mid
        elif result == 'left':
            hi = mid - 1
        else:
            lo = mid + 1
    return 0

```

```
evaluate_test_case(count_rotations_generic, test)
```

Input:

```
{'nums': [19, 25, 29, 3, 5, 6, 7, 9, 11, 14]}
```

Expected Output:

3

Actual Output:

3

Execution Time:

0.007 ms

Test Result:

PASSED

(3, True, 0.007)

```
evaluate_test_cases(count_rotations_generic, tests)
```

TEST CASE #0

Input:

```
{'nums': [19, 25, 29, 3, 5, 6, 7, 9, 11, 14]}
```

Expected Output:

3

Actual Output:

3

Execution Time:

0.007 ms

Test Result:

PASSED

TEST CASE #1

Input:

```
{'nums': [4, 5, 6, 7, 8, 1, 2, 3]}
```

Expected Output:

5

Actual Output:

5

Execution Time:

0.004 ms

Test Result:

PASSED

TEST CASE #2

Input:

```
{'nums': [1, 2, 3, 4, 5, 6, 7, 8]}
```

Expected Output:

0

Actual Output:

0

Execution Time:

0.002 ms

Test Result:

PASSED

TEST CASE #3

Input:

```
{'nums': [10, 4, 5, 6, 7, 8, 9]}
```

Expected Output:

1

Actual Output:

1

Execution Time:

0.003 ms

Test Result:

PASSED

TEST CASE #4

Input:

```
{'nums': [10, 4, 5, 6, 7, 8, 9]}
```

Expected Output:

1

Actual Output:

1

Execution Time:

0.003 ms

Test Result:

PASSED

TEST CASE #5

Input:

```
{'nums': [1, 2, 3, 4, 5, 6, 7, 8]}
```

Expected Output:

0

Actual Output:

0

Execution Time:

0.003 ms

Test Result:

PASSED

TEST CASE #6

Input:

```
{'nums': []}
```

Expected Output:

0

Actual Output:

0

Execution Time:

0.002 ms

Test Result:

PASSED

TEST CASE #7

Input:

{'nums': [3]}

Expected Output:

0

Actual Output:

0

Execution Time:

0.002 ms

Test Result:

PASSED

SUMMARY

TOTAL: 8, PASSED: 8, FAILED: 0

[(3, True, 0.007),
(5, True, 0.004),
(0, True, 0.002),
(1, True, 0.003),
(1, True, 0.003),
(0, True, 0.003),
(0, True, 0.002),
(0, True, 0.002)]

```
jovian.commit()
```

[jovian] Updating notebook "evanmarie/python-binary-search-assignment" on <https://jovian.ai>

```
-----  
ApiError                                Traceback (most recent call last)  
/tmp/ipykernel_1053/2763693686.py in <module>  
----> 1 jovian.commit()  
  
/opt/conda/lib/python3.9/site-packages/jovian/utils/commit.py in commit(message, files,  
outputs, environment, privacy, filename, project, new_project, git_commit, git_message,  
**kwargs)  
    197  
    198     # Create or update gist (with title and )  
--> 199     res = api.create_gist_simple(filename, project_id, privacy, project_title,  
message)  
    200     slug, owner, version, title = res['slug'], res['owner'], res['version'],  
res['title']  
    201     username = owner['username']  
  
/opt/conda/lib/python3.9/site-packages/jovian/utils/api.py in  
create_gist_simple(filename, gist_slug, privacy, title, version_title)  
    57         nb_file = (filename, f)  
    58         if gist_slug:  
---> 59             return upload_file(gist_slug=gist_slug, file=nb_file,  
version_title=version_title)  
    60         else:  
    61             data = {'visibility': privacy}  
  
/opt/conda/lib/python3.9/site-packages/jovian/utils/api.py in upload_file(gist_slug,  
file, folder, version, artifact, version_title)  
    98         log(warning, error=True)  
    99         return data  
--> 100     raise ApiError('File upload failed: ' + pretty(res))  
    101  
    102
```

ApiError: File upload failed: (HTTP 400) Uploaded notebook file seems to be corrupt.

Discuss your solution on the forum: <https://jovian.ai/forum/c/data-structures-and-algorithms-in-python/assignment-1/87>

Optional Bonus 2: Handling repeating numbers

So far we've assumed that the numbers in the list are unique. What if the numbers can repeat? E.g. [5, 6, 6, 9, 9, 9, 0, 0, 2, 3, 3, 3, 3, 4, 4]. Can you modify your solution to handle this special case?

Q (Optional): Create additional test cases where the list can contain repeating numbers

```
extended_tests = list(tests)
```

```
extended_tests.append({
    'input': {
        'nums': [5, 6, 6, 9, 9, 9, 0, 0, 2, 3, 3, 3, 3, 4, 4]
    },
    'output': 6
})
```

```
extended_tests.append({
    'input': {
        'nums': [5, 6, 6, 9, 9, 9, 0, 0, 2, 3, 3, 3, 3, 4, 5]
    },
    'output': 6
})
```

```
# add more test cases if required
```

```
extended_tests.append({
    'input': {
        'nums': [6, 6, 9, 9, 9, 0, 0, 2, 3, 3, 3, 3, 4, 5, 5]
    },
    'output': 5
})
```

Q (Optional): Modify your solution (if required) to handle the case where the list can contain repeating numbers.

```
def condition(mid, hi, lo, nums):
    mid_number = nums[mid]
    if mid > 0 and mid_number < nums[mid-1]:
        # The middle position is the answer
        return 'found', mid

    elif mid > 0 and mid_number < nums[hi] and mid_number == nums[mid-1]:
        repeated = mid
        while nums[repeated] == nums[repeated - 1]:
            repeated = repeated - 1

        if repeated > 0 and nums[repeated] < nums[repeated-1]:
            return 'found', repeated

        else:
            return 'left', repeated

    elif mid > 0 and mid_number < nums[hi]:
```

```

        return 'left', mid

    elif mid > 0 and nums[mid] >= nums[hi]:
        return 'right', None

    return None

def count_rotations_generic(nums):

    if len(nums) == 0 or len(nums) == 1:
        return 0
    if nums[0] < nums[-1]:
        return 0

    lo = 0
    hi = len(nums) - 1

    while lo <= hi:
        mid = (lo + hi) // 2
        result, index = condition(mid, lo, hi, nums)

        if result == 'found':
            return index
        elif result == 'left':
            hi = index - 1
        else:
            lo = mid + 1
    return 0

```

Test your function to make sure it works properly.

```
evaluate_test_cases(count_rotations_generic, extended_tests)
```

Discuss your solution on the forum: <https://jovian.ai/forum/c/data-structures-and-algorithms-in-python/assignment-1/87>

Optional Bonus 3: Searching in a Rotated List

Here's a slightly advanced extension to this problem:

You are given list of numbers, obtained by rotating a sorted list an unknown number of times. You are also given a target number. Write a function to find the position of the target number within the rotated list. You can assume that all the numbers in the list are unique.

Example: In the rotated sorted list [5, 6, 9, 0, 2, 3, 4], the target number 2 occurs at position 5.

Q (Optional): Create some test cases for the above problem.

```
bonus_test_01 = {  
  'input': {  
    'nums': [5, 6, 9, 0, 2, 3, 4],  
    'target': 9  
  },  
  
  'output': 2  
}
```

```
bonus_test_02 = {  
  'input': {  
    'nums': [7, 8, 9, 1, 2, 3, 4, 5, 6],  
    'target': 4  
  },  
  
  'output': 6  
}
```

```
bonus_test_03 = {  
  'input': {  
    'nums': [22, 24, 25, 28, 11, 13, 15, 17, 19, 21],  
    'target': 11  
  },  
  
  'output': 4  
}
```

```
bonus_test_04 = {  
  'input': {  
    'nums': [66, 67, 68, 69, 63, 64, 65, 66],  
    'target': 68  
  },  
  
  'output': 2  
}
```

```
# Has repeated numbers  
bonus_test_05 = {  
  'input': {  
    'nums': [66, 67, 67, 68, 69, 63, 64, 65, 66],  
    'target': 68  
  },  
  
  'output': 3  
}
```

```
# Has repeated numbers
bonus_test_06 = {
    'input': {
        'nums': [],
        'target' : None
    },

    'output': None
}
```

```
# Has repeated numbers
bonus_test_07 = {
    'input': {
        'nums': [23],
        'target' : 23,
    },

    'output': 0
}
```

```
tests_02 = [bonus_test_01, bonus_test_02, bonus_test_03, bonus_test_04, bonus_test_05,
```

Q (Optional): Implement a solution to the above problem using binary search.

HINT: One way to solve this problem is to identify two sorted subarrays within the given array (using the `count_rotations_binary` function defined above), then perform a binary search on each subarray to determine the position of the target element. Another way is to modify `count_rotations_binary` to solve the problem directly.

```
def split_nums(nums):
    cutoff = count_rotations_binary(nums)
    return nums[:cutoff], nums[cutoff:]

def find_element(nums, target):
    if len(nums) == 0:
        return None
    if len(nums) == 1:
        if nums[0] == target:
            return 0
        else:
            return None
    left, right = split_nums(nums)
    print(f"left: {left}, right: {right}")

    if left[0] <= target and target <= left[-1]:
        print(f"Searching left for {target}.")
        return binary_search_of_the_gods(left, target)
```

```
else:
    print(f"Searching right for {target}.")
    return (binary_search_of_the_gods(right, target) + len(left))
```

```
def binary_search_of_the_gods(nums, target):
    lo = 0
    hi = len(nums) - 1

    while lo <= hi:
        middle = (lo + hi) // 2
        mid_number = nums[middle]

        if mid_number == target:
            return middle
        elif mid_number < target:
            lo = middle + 1
        elif mid_number > target:
            hi = middle - 1

    return None
```

Test your solution using the cells below.

```
evaluate_test_cases(find_element, tests_02)
```

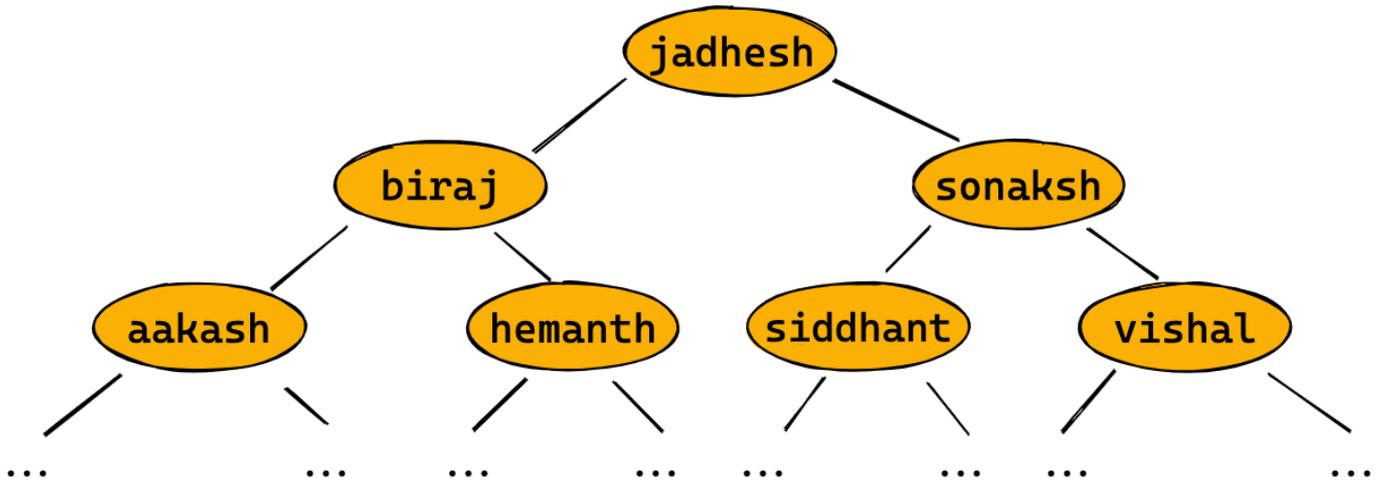
You can test your solution to the above problem here: <https://leetcode.com/problems/search-in-rotated-sorted-array/>

Discuss your approach on the forum: <https://jovian.ai/forum/c/data-structures-and-algorithms-in-python/assignment-1/87>

```
jovian.commit()
```

Binary Search Trees, Traversals and Balancing in Python

Part 2 of "Data Structures and Algorithms in Python"



[Data Structures and Algorithms in Python](#) is a beginner-friendly introduction to common data structures (linked lists, stacks, queues, graphs) and algorithms (search, sorting, recursion, dynamic programming) in Python, designed to help you prepare for coding interviews and assessments.

Earn a verified certificate of accomplishment for this course by signing up here: <http://pythonsa.com>.

Ask questions, get help & participate in discussions on the community forum: <https://jovian.ai/forum/c/data-structures-and-algorithms-in-python/78>

Prerequisites

This course assumes very little background in programming and mathematics, and you can learn the required concepts here:

- Basic programming with Python ([variables](#), [data types](#), [loops](#), [functions](#) etc.)
- Some high school mathematics ([polynomials](#), [vectors](#), [matrices](#) and [probability](#))
- No prior knowledge of data structures or algorithms is required

We'll cover any additional mathematical and theoretical concepts we need as we go along.

How to Run the Code

The best way to learn the material is to execute the code and experiment with it yourself. This tutorial is an executable [Jupyter notebook](#). You can *run* this tutorial and experiment with the code examples in a couple of ways: *using free online resources* (recommended) or *on your computer*.

Option 1: Running using free online resources (1-click, recommended)

The easiest way to start executing the code is to click the **Run** button at the top of this page and select **Run on Binder**. You can also select "Run on Colab" or "Run on Kaggle", but you'll need to create an account on [Google Colab](#) or [Kaggle](#) to use these platforms.

Option 2: Running on your computer locally

To run the code on your computer locally, you'll need to set up [Python](#), download the notebook and install the required libraries. We recommend using the [Conda](#) distribution of Python. Click the **Run** button at the top of this page, select the **Run Locally** option, and follow the instructions.

Problem

In this notebook, we'll focus on solving the following problem:

QUESTION 1: As a senior backend engineer at Jovian, you are tasked with developing a fast in-memory data structure to manage profile information (username, name and email) for 100 million users. It should allow the following operations to be performed efficiently:

1. **Insert** the profile information for a new user.
2. **Find** the profile information of a user, given their username
3. **Update** the profile information of a user, given their username
4. **List** all the users of the platform, sorted by username

You can assume that usernames are unique.

Along the way, we will also solve several other questions related to binary trees and binary search trees that are often asked in coding interviews and assessments.

The Method

Here's a systematic strategy we'll apply for solving problems:

1. State the problem clearly. Identify the input & output formats.
2. Come up with some example inputs & outputs. Try to cover all edge cases.
3. Come up with a correct solution for the problem. State it in plain English.
4. Implement the solution and test it using example inputs. Fix bugs, if any.
5. Analyze the algorithm's complexity and identify inefficiencies, if any.
6. Apply the right technique to overcome the inefficiency. Repeat steps 3 to 6.

1. State the problem clearly. Identify the input & output formats.

Problem

We need to create a data structure which can store 100 million records and perform insertion, search, update and list operations efficiently.

Input

The key inputs to our data structure are user profiles, which contain the username, name and email of a user.

A Python *class* would be a great way to represent the information for a user. A class is a blueprint for creating *objects*. Everything in Python is an *object* belonging to some *class*. Here's the simplest possible class in Python, with nothing in it:

```
class User:  
    pass
```

We can create or *instantiate* an object of the class by calling it like a function.

```
user1 = User()
```

We can verify that the object is of the class `User` .

```
user1
```

```
<__main__.User at 0x7faf012c3b00>
```

```
type(user1)
```

```
__main__.User
```

The object `user1` does not contain any useful information. Let's add a *constructor method* to the class to store some *attributes* or *properties*.

```
class User:  
    def __init__(self, username, name, email):  
        self.username = username  
        self.name = name  
        self.email = email  
        print('User created!')
```

We can now create an object with some properties.

```
user2 = User('john', 'John Doe', 'john@doe.com')
```

```
User created!
```

```
user2
```

```
<__main__.User at 0x7faf012c3e48>
```

Here's what's happening above (conceptually):

- Python creates an empty object of the type `user` and stores it in the variable `user2`
- Python then invokes the function `User.__init__` with the arguments `user2`, `"john"`, `"John Doe"` and `"john@doe.com"`
- As the `__init__` function is executed, the properties `username`, `name` and `email` are set on the object `user2`

We can access the properties of the object using the `.` notation.

```
user2.name
```

```
'John Doe'
```

```
user2.email, user2.username
```

```
('john@doe.com', 'john')
```

You can also define custom methods inside a class.

```
class User:
    def __init__(self, username, name, email):
        self.username = username
        self.name = name
        self.email = email

    def introduce_yourself(self, guest_name):
        print("Hi {}, I'm {}! Contact me at {} .".format(guest_name, self.name, self.email))
```

```
user3 = User('jane', 'Jane Doe', 'jane@doe.com')
```

```
user3.introduce_yourself('David')
```

```
Hi David, I'm Jane Doe! Contact me at jane@doe.com .
```

When we try to invoke the method `user3.introduce_yourself`, the object `user3` is automatically passed as the first argument `self`. Indeed, the following statement is equivalent to the above statement.

```
User.introduce_yourself(user3, 'David')
```

```
Hi David, I'm Jane Doe! Contact me at jane@doe.com .
```

Finally, we'll define a couple of helper methods to display user objects nicely within Jupyter.

```
class User:
    def __init__(self, username, name, email):
        self.username = username
        self.name = name
        self.email = email

    def __repr__(self):
        return "User(username='{}', name='{}', email='{}')".format(self.username, self.name, self.email)

    def __str__(self):
        return self.__repr__()
```

```
user4 = User('jane', 'Jane Doe', 'jane@doe.com')
```

```
user4
```

```
User(username='jane', name='Jane Doe', email='jane@doe.com')
```

Exercise: What is the purpose of defining the functions `__str__` and `__repr__` within a class? How are the two functions different? Illustrate with some examples using the empty cells below.

Learn more about classes in Python here: <https://jovian.ai/aakashns/python-classes-and-linked-lists>.

Output

We can also express our desired data structure as a Python class `UserDatabase` with four methods: `insert`, `find`, `update` and `list_all`.

```
class UserDatabase:
    def insert(self, user):
        pass

    def find(self, username):
        pass

    def update(self, user):
        pass

    def list_all(self):
        pass
```

It's good programming practice to list out the signatures of different class functions before we actually implement the class.

2. Come up with some example inputs & outputs.

Let's create some sample user profiles that we can use to test our functions once we implement them.

```
aakash = User('aakash', 'Aakash Rai', 'aakash@example.com')
biraj = User('biraj', 'Biraj Das', 'biraj@example.com')
hemanth = User('hemanth', 'Hemanth Jain', 'hemanth@example.com')
jadhesh = User('jadhesh', 'Jadhesh Verma', 'jadhesh@example.com')
siddhant = User('siddhant', 'Siddhant Sinha', 'siddhant@example.com')
sonaksh = User('sonaksh', 'Sonaksh Kumar', 'sonaksh@example.com')
vishal = User('vishal', 'Vishal Goel', 'vishal@example.com')
```

```
users = [aakash, biraj, hemanth, jadhesh, siddhant, sonaksh, vishal]
```

We can access different fields within a user profile object using the `.` (dot) notation.

```
biraj.username, biraj.email, biraj.name
```

```
('biraj', 'biraj@example.com', 'Biraj Das')
```

We can also view a string representation of the object, since defined the `__repr__` and `__str__` methods

```
print(aakash)
```

```
User(username='aakash', name='Aakash Rai', email='aakash@example.com')
```

```
users
```

```
[User(username='aakash', name='Aakash Rai', email='aakash@example.com'),  
User(username='biraj', name='Biraj Das', email='biraj@example.com'),  
User(username='hemanth', name='Hemanth Jain', email='hemanth@example.com'),  
User(username='jadhesh', name='Jadhesh Verma', email='jadhesh@example.com'),  
User(username='siddhant', name='Siddhant Sinha', email='siddhant@example.com'),  
User(username='sonaksh', name='Sonaksh Kumar', email='sonaksh@example.com'),  
User(username='vishal', name='Vishal Goel', email='vishal@example.com')]
```

Since we haven't implemented our data structure yet, it's not possible to list sample outputs. However you can try to come up with different scenarios to test future implementations

Exercise: List some scenarios for testing the class methods `insert`, `find`, `update` and `list_all`.

1. Insert:

- A. Inserting into an empty database of users
- B. Trying to insert a user with a username that already exists
- C. Inserting a user with a username that does not exist
- D. ???

2. Find:

- A. ???
- B. ???
- C. ???

3. Update:

- A. ???
- B. ???
- C. ???

4. List:

- A. ???
- B. ???

C. ???

3. Come up with a correct solution. State it in plain English.

Here's a simple and easy solution to the problem: we store the `User` objects in a list sorted by usernames.

The various functions can be implemented as follows:

1. **Insert:** Loop through the list and add the new user at a position that keeps the list sorted.
2. **Find:** Loop through the list and find the user object with the username matching the query.
3. **Update:** Loop through the list, find the user object matching the query and update the details
4. **List:** Return the list of user objects.

We can use the fact usernames, which are strings can be compared using the `<`, `>` and `==` operators in Python.

```
'biraj' < 'hemanth'
```

True

4. Implement the solution and test it using example inputs.

The code for implementing the above solution is also fairly straightforward.

```
class UserDatabase:
    def __init__(self):
        self.users = []

    def insert(self, user):
        i = 0
        while i < len(self.users):
            # Find the first username greater than the new user's username
            if self.users[i].username > user.username:
                break
            i += 1
        self.users.insert(i, user)

    def find(self, username):
        for user in self.users:
            if user.username == username:
                return user

    def update(self, user):
        target = self.find(user.username)
        target.name, target.email = user.name, user.email

    def list_all(self):
        return self.users
```

We can create a new database of users by *instantiating* an object of the `UserDatabase` class.

```
database = UserDatabase()
```

Let's insert some entries into the object.

```
database.insert(hemanth)
database.insert(aakash)
database.insert(siddhant)
```

We can now retrieve the data for a user, given their username.

```
user = database.find('siddhant')
user
```

```
User(username='siddhant', name='Siddhant Sinha', email='siddhant@example.com')
```

Let's try changing the information for a user

```
database.update(User(username='siddhant', name='Siddhant U', email='siddhantu@example.com'))
```

```
user = database.find('siddhant')
user
```

```
User(username='siddhant', name='Siddhant U', email='siddhantu@example.com')
```

Finally, we can retrieve a list of users in alphabetical order.

```
database.list_all()
```

```
[User(username='aakash', name='Aakash Rai', email='aakash@example.com'),
 User(username='hemanth', name='Hemanth Jain', email='hemanth@example.com'),
 User(username='siddhant', name='Siddhant U', email='siddhantu@example.com')]
```

Let's verify that a new user is inserted into the correct position.

```
database.insert(biraj)
```

```
database.list_all()
```

```
[User(username='aakash', name='Aakash Rai', email='aakash@example.com'),
 User(username='biraj', name='Biraj Das', email='biraj@example.com'),
 User(username='hemanth', name='Hemanth Jain', email='hemanth@example.com'),
 User(username='siddhant', name='Siddhant U', email='siddhantu@example.com')]
```

The user `biraj` was inserted just before `hemanth`, as expected.

Exercise: Use the empty cells below to test the various scenarios you listed in step 2 above.

5. Analyze the algorithm's complexity and identify inefficiencies

The operations `insert`, `find`, `update` involves iterating over a list of users, in the worst case, they may take up to N iterations to return a result, where N is the total number of users. `list_all` however, simply returns the existing internal list of users.

Thus, the time complexities of the various operations are:

1. Insert: $O(N)$
2. Find: $O(N)$
3. Update: $O(N)$
4. List: $O(1)$

Exercise: Verify that the space complexity of each operation is $O(1)$.

Is this good enough? To get a sense how long each function might take if there are 100 million users on the platform, we can simply run an `for` or `while` loop on 10 million numbers.

```
%%time
for i in range(100000000):
    j = i*i
```

CPU times: user 8.42 s, sys: 8.05 ms, total: 8.42 s

Wall time: 8.43 s

It takes almost 10 seconds to execute all the iterations in the above cell.

- A 10-second delay for fetching user profiles will lead to a suboptimal users experience and may cause many users to stop using the platform altogether.
- The 10-second processing time for each profile request will also significantly limit the number of users that can access the platform at a time or increase the cloud infrastructure costs for the company by millions of dollars.

As a senior backend engineer, you must come up with a more efficient data structure! Choosing the right data structure for the requirements at hand is an important skill. It's apparent that a sorted list of users might not be the best data structure to organize profile information for millions of users.

Save and upload your work to Jovian

Whether you're running this Jupyter notebook online or on your computer, it's essential to save your work from time to time. You can continue working on a saved notebook later or share it with friends and colleagues to let them execute your code. [Jovian](#) offers an easy way of saving and sharing your Jupyter notebooks online.

```
!pip install jovian --upgrade --quiet
```

```
import jovian
```

```
jovian.commit(project='python-binary-search-trees')
```

```
[jovian] Attempting to save notebook..
```

```
[jovian] Updating notebook "aakashns/python-binary-search-trees" on https://jovian.ai/
```

```
[jovian] Uploading notebook..
```

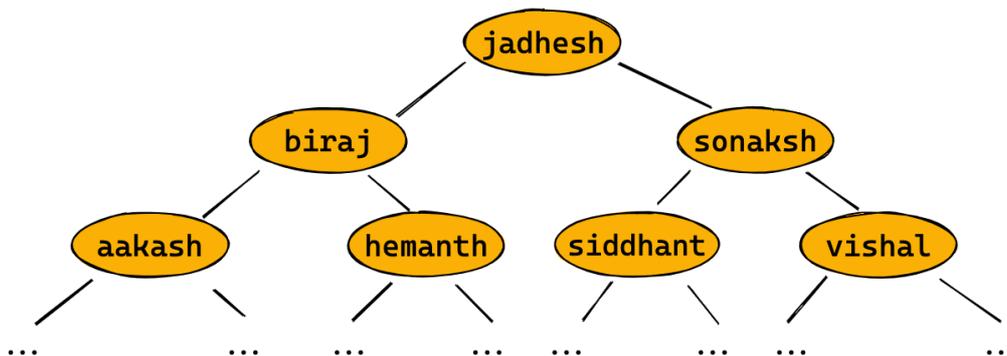
```
[jovian] Capturing environment..
```

```
[jovian] Committed successfully! https://jovian.ai/aakashns/python-binary-search-trees
```

```
'https://jovian.ai/aakashns/python-binary-search-trees'
```

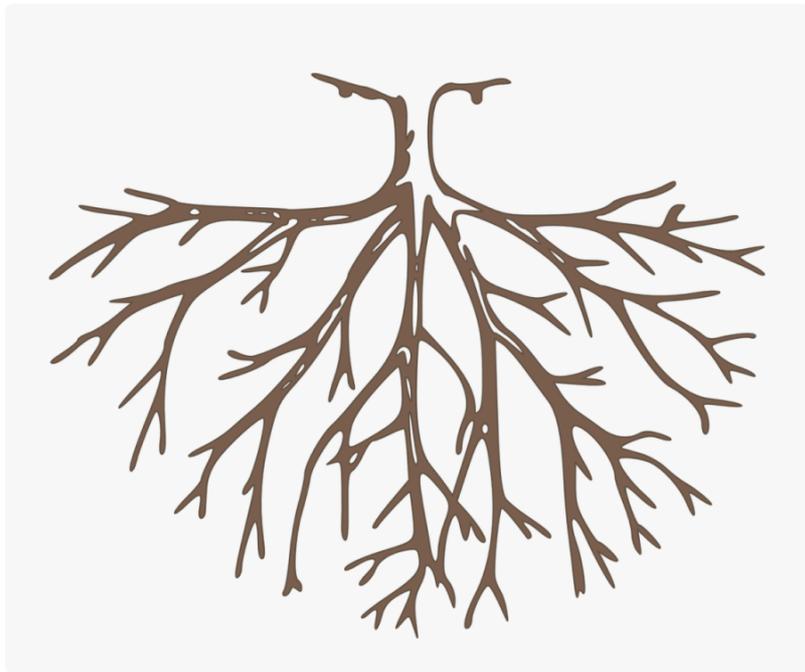
6. Apply the right technique to overcome the inefficiency

We can limit the number of iterations required for common operations like find, insert and update by organizing our data in the following structure, called a **binary tree**:

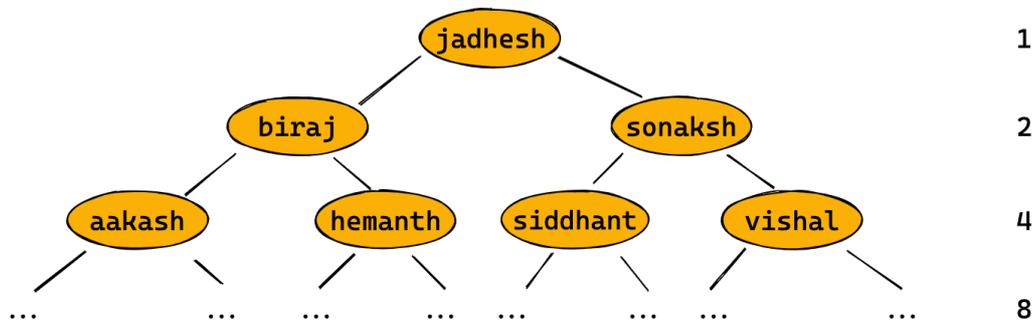


It's called a tree because it vaguely like an inverted tree trunk with branches.

- The word "binary" indicates that each "node" in the tree can have at most 2 children (left or right).
- Nodes can have 0, 1 or 2 children. Nodes that do not have any children are sometimes also called "leaves".
- The single node at the top is called the "root" node, and it typically where operations like search, insertion etc. begin.



Balanced Binary Search Trees



For our use case, we require the binary tree to have some additional properties:

1. **Keys and Values:** Each node of the tree stores a key (a username) and a value (a User object). Only keys are shown in the picture above for brevity. A binary tree where nodes have both a key and a value is often referred to as a **map** or **treemap** (because it maps keys to values).
2. **Binary Search Tree:** The *left subtree* of any node only contains nodes with keys that are lexicographically smaller than the node's key, and the *right subtree* of any node only contains nodes with keys that lexicographically larger than the node's key. A tree that satisfies this property is called a **binary search trees**, and it's easy to locate a specific key by traversing a single path down from the root note.
3. **Balanced Tree:** The tree is **balanced** i.e. it does not skew too heavily to one side or the other. The left and right subtrees of any node shouldn't differ in height/depth by more than 1 level.

Height of a Binary Tree

The number of levels in a tree is called its height. As you can tell from the picture above, each level of a tree contains twice as many nodes as the previous level.

For a tree of height k , here's a list of the number of nodes at each level:

Level 0: 1

Level 1: 2

Level 2: 4 i.e. 2^2

Level 3: 8 i.e. 2^3

...

Level k-1: $2^{(k-1)}$

If the total number of nodes in the tree is N , then it follows that

$$N = 1 + 2^1 + 2^2 + 2^3 + \dots + 2^{(k-1)}$$

We can simplify this equation by adding 1 on each side:

$$N + 1 = 1 + 1 + 2^1 + 2^2 + 2^3 + \dots + 2^{(k-1)}$$

$$N + 1 = 2^1 + 2^1 + 2^2 + 2^3 + \dots + 2^{(k-1)}$$

$$N + 1 = 2^2 + 2^2 + 2^3 + \dots + 2^{(k-1)}$$

$$N + 1 = 2^3 + 2^3 + \dots + 2^{(k-1)}$$

...

$$N + 1 = 2^{(k-1)} + 2^{(k-1)}$$

$$N + 1 = 2^k$$

$$k = \log(N + 1) \leq \log(N) + 1$$

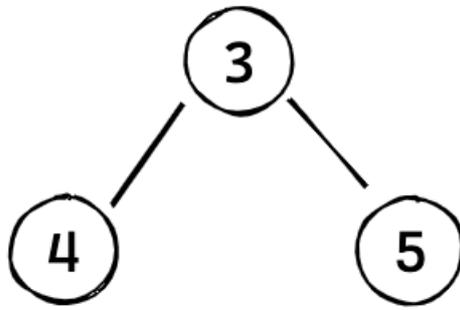
Thus, to store N records we require a balanced binary search tree (BST) of height no larger than $\log(N) + 1$. This is a very useful property, in combination with the fact that nodes are arranged in a way that makes it easy to find a specific key by following a single path down from the root.

As we'll see soon, the `insert`, `find` and `update` operations in a balanced BST have time complexity $O(\log N)$ since they all involve traversing a single path down from the root of the tree.

Binary Tree

QUESTION 2: Implement a binary tree using Python, and show its usage with some examples.

To begin, we'll create simple binary tree (without any of the additional properties) containing numbers as keys within nodes. Here's an example:



Here's a simple class representing a node within a binary tree.

```
class TreeNode:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
```

Let's create objects representing each node of the above tree

```
node0 = TreeNode(3)
node1 = TreeNode(4)
node2 = TreeNode(5)
```

Let's verify that `node0` is an object of the type `TreeNode` and has the property `key` set to `3`.

```
node0
```

```
<__main__.TreeNode at 0x7faf03417b00>
```

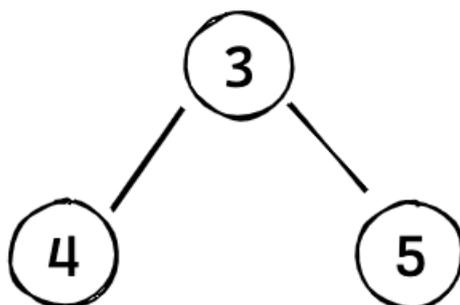
```
node0.key
```

```
3
```

We can *connect* the nodes by setting the `.left` and `.right` properties of the root node.

```
node0.left = node1
node0.right = node2
```

And we're done! We can create a new variable `tree` which simply points to the root node, and use it to access all the nodes within the tree.



```
tree = node0
```

```
tree.key
```

3

```
tree.left.key
```

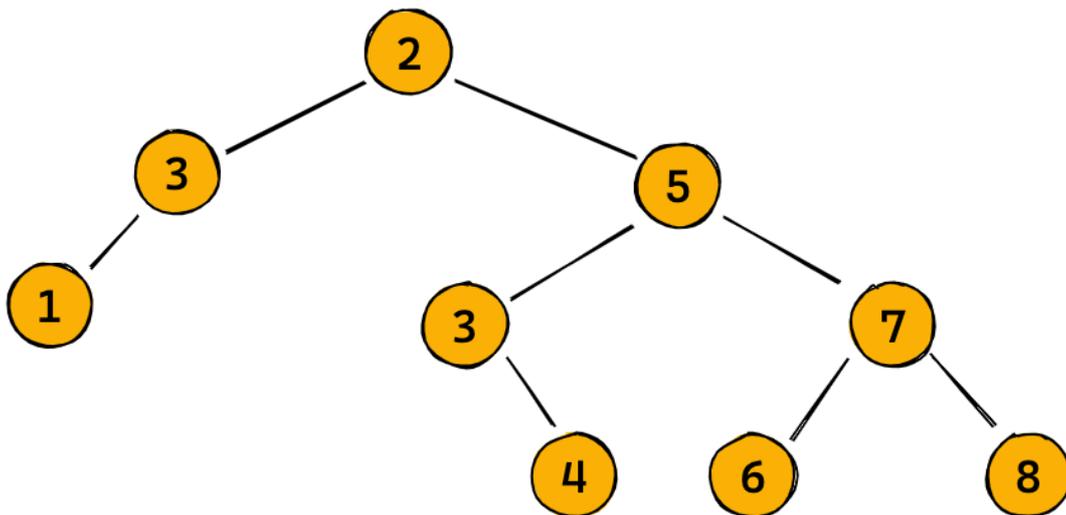
4

```
tree.right.key
```

5

Going forward, we'll use the term "tree" to refer to the root node. The term "node" can refer to any node in a tree, not necessarily the root.

Exercise: Create the following binary tree using the `TreeNode` class defined above.



It's a bit inconvenient to create a tree by manually connecting all the nodes. Let's write a helper function which can convert a tuple with the structure (left_subtree, key, right_subtree) (where left_subtree and right_subtree are themselves tuples) into binary tree.

Here's an tuple representing the tree shown above:

```
tree_tuple = ((1,3, None), 2, ((None, 3, 4), 5, (6, 7, 8)))
```

```
def parse_tuple(data):  
    # print(data)  
    if isinstance(data, tuple) and len(data) == 3:
```

```

node = TreeNode(data[1])
node.left = parse_tuple(data[0])
node.right = parse_tuple(data[2])
elif data is None:
    node = None
else:
    node = TreeNode(data)
return node

```

The `parse_tuple` creates a new root node when a tuple of size 3 as an the input. Interestingly, to create the left and right subtrees for the node, the `parse_tuple` function invokes itself. This technique is called *recursion*. The chain of *recursive* calls ends when `parse_tuple` encounters a number or `None` as input. We'll use recursion extensively throughout this tutorial.

Exercise: Add print statements inside `parse_tuple` to display the arguments for each call of the function. Does the sequence of recursive calls make sense to you?

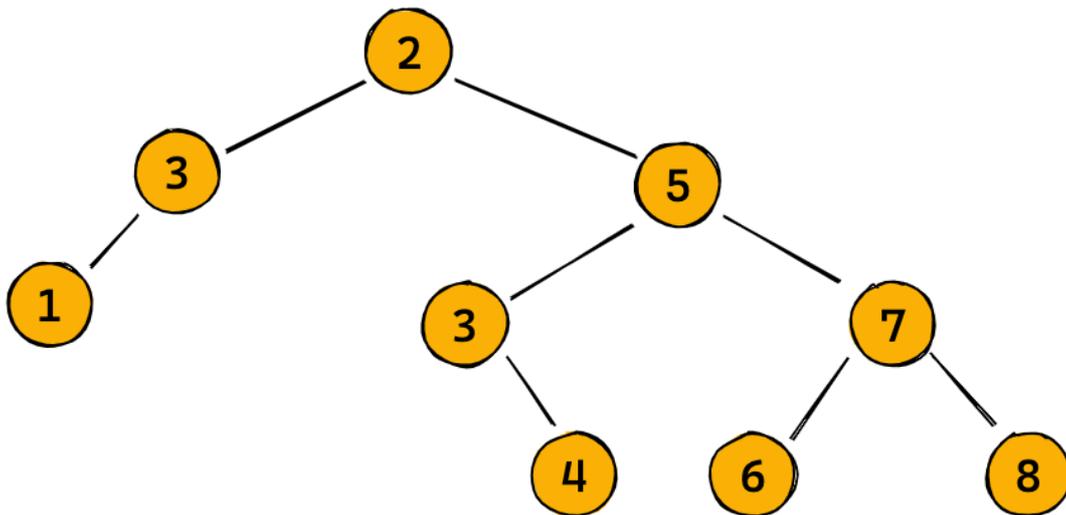
Let's try out `parse_tuple` with the tuple define earlier.

```
tree2 = parse_tuple(((1,3,None), 2, ((None, 3, 4), 5, (6, 7, 8))))
```

```
tree2
```

```
<__main__.TreeNode at 0x7faf034263c8>
```

We can now examine the tree to verify that it was constructed as expected.



```
tree2.key
```

```
2
```

```
tree2.left.key, tree2.right.key
```

```
(3, 5)
```

```
tree2.left.left.key, tree2.left.right, tree2.right.left.key, tree2.right.right.key
```

(1, None, 3, 7)

```
tree2.right.left.right.key, tree2.right.right.left.key, tree2.right.right.right.key
```

(4, 6, 8)

Exercise: Define a function `tree_to_tuple` that converts a binary tree into a tuple representing the same tree. E.g. `tree_to_tuple` converts the tree created above to the tuple `((1, 3, None), 2, ((None, 3, 4), 5, (6, 7, 8)))`. *Hint:* Use recursion.

```
def tree_to_tuple(node):  
    pass
```

Let's create another helper function to display all the keys in a tree-like structure for easier visualization.

```
def display_keys(node, space='\t', level=0):  
    # print(node.key if node else None, level)  
  
    # If the node is empty  
    if node is None:  
        print(space*level + 'Ø')  
        return  
  
    # If the node is a leaf  
    if node.left is None and node.right is None:  
        print(space*level + str(node.key))  
        return  
  
    # If the node has children  
    display_keys(node.right, space, level+1)  
    print(space*level + str(node.key))  
    display_keys(node.left, space, level+1)
```

Once again, the `display_keys` function uses recursion to print all the keys of the left and right subtree with proper indentation.

Exercise: Add print statements inside `display_keys` to display the arguments for each call of the function. Does the sequence of recursive calls make sense to you?

Let's try using the function.

```
display_keys(tree2, '  ')
```

8

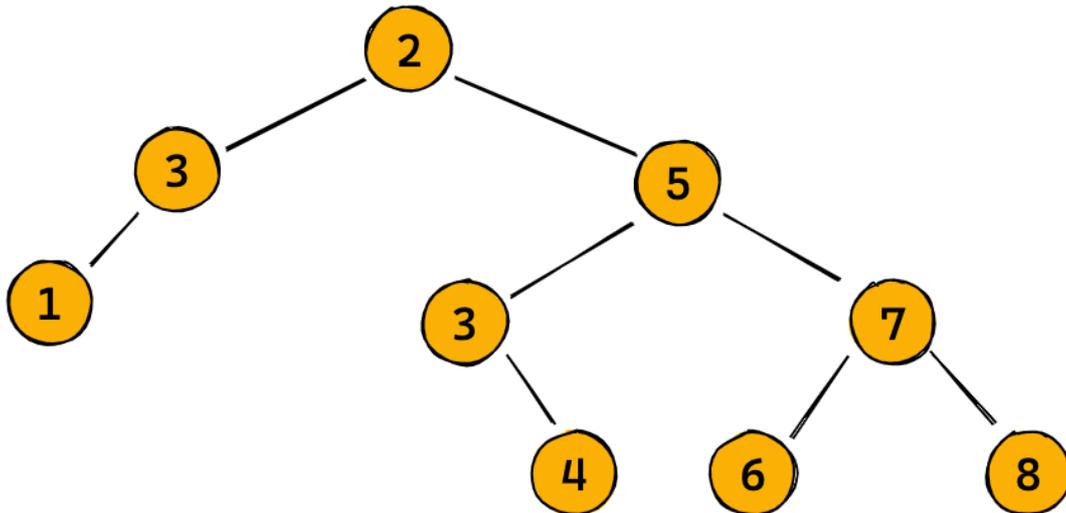
7

6

5

4
3
∅
2
∅
3
1

We can now visualize the tree that was just created (albeit rotated by 90 degrees). It's easy to see that it matches the expected structure.



Exercise: Create some more trees and visualize them using `display_keys`. You can use excalidraw.com as a digital whiteboard to create trees.

Traversing a Binary Tree

The following questions are frequently asked in coding interviews and assessments:

QUESTION 3: Write a function to perform the *inorder* traversal of a binary tree.

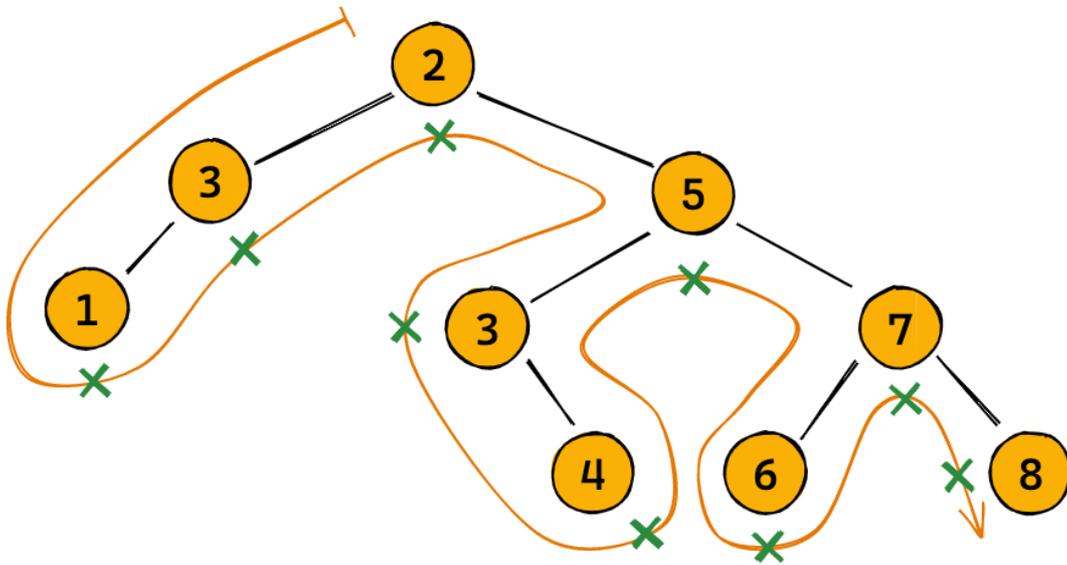
QUESTION 4: Write a function to perform the *preorder* traversal of a binary tree.

QUESTION 5: Write a function to perform the *postorder* traversal of a binary tree.

A *traversal* refers to the process of visiting each node of a tree exactly once. *Visiting a node* generally refers to adding the node's key to a list. There are three ways to traverse a binary tree and return the list of visited keys:

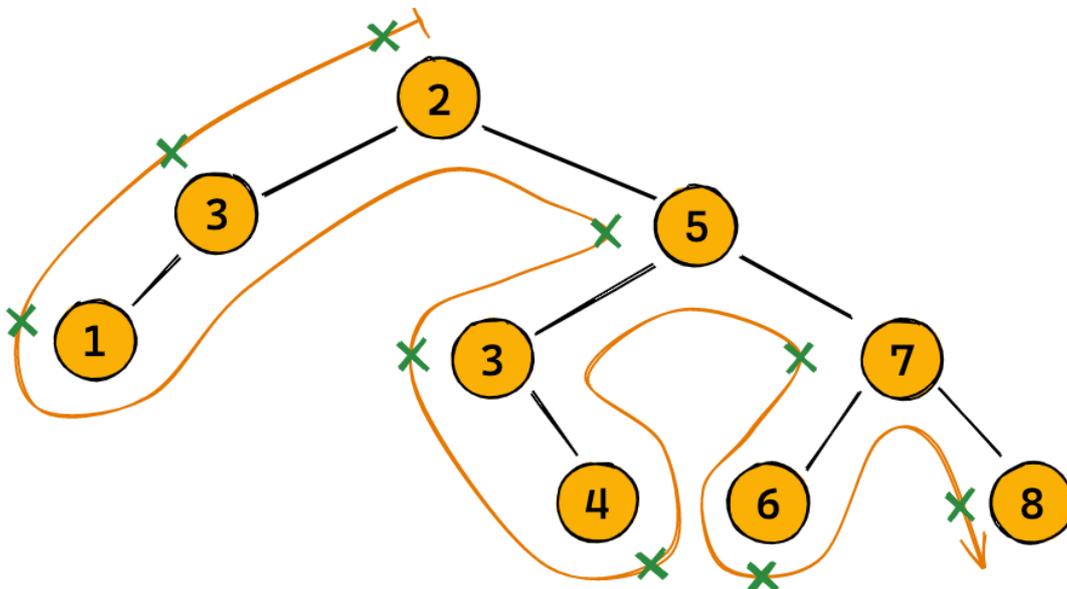
Inorder traversal

1. Traverse the left subtree recursively inorder.
2. Traverse the current node.
3. Traverse the right subtree recursively inorder.



Preorder traversal

1. Traverse the current node.
2. Traverse the left subtree recursively preorder.
3. Traverse the right subtree recursively preorder.



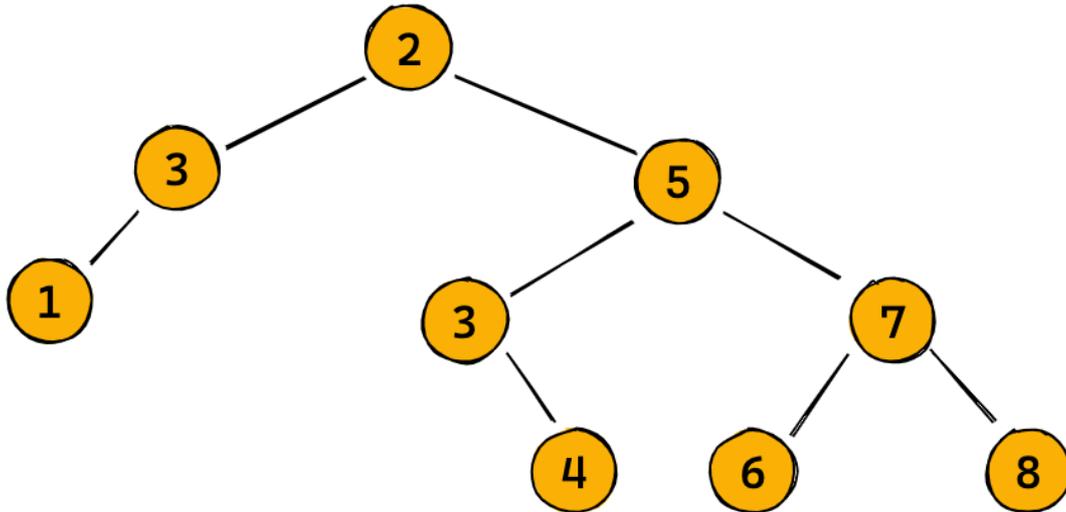
Can you guess how **postorder** traversal works??

Here's an implementation of inorder traversal of a binary tree.

```
def traverse_in_order(node):  
    if node is None:  
        return []  
    return(traverse_in_order(node.left) +
```

```
[node.key] +  
traverse_in_order(node.right))
```

Let's try it out with this tree:



```
tree = parse_tuple(((1,3, None), 2, ((None, 3, 4), 5, (6, 7, 8))))
```

```
display_keys(tree, '  ')
```

```
  8  
 7  
 6  
5  
 4  
 3  
  ∅  
2  
  ∅  
 3  
 1
```

```
traverse_in_order(tree)
```

```
[1, 3, 2, 3, 4, 5, 6, 7, 8]
```

Exercise: Implement functions for preorder and postorder traversal of a binary tree.

Test your implementations by making submissions to the following problems:

- <https://leetcode.com/problems/binary-tree-inorder-traversal/>
- <https://leetcode.com/problems/binary-tree-preorder-traversal/>
- <https://leetcode.com/problems/binary-tree-postorder-traversal/>

```
jovian.commit()
```

[jovian] Attempting to save notebook..

[jovian] Updating notebook "aakashns/python-binary-search-trees" on <https://jovian.ai/>

[jovian] Uploading notebook..

[jovian] Capturing environment..

[jovian] Committed successfully! <https://jovian.ai/aakashns/python-binary-search-trees>

'<https://jovian.ai/aakashns/python-binary-search-trees>'

Height and Size of a Binary Tree

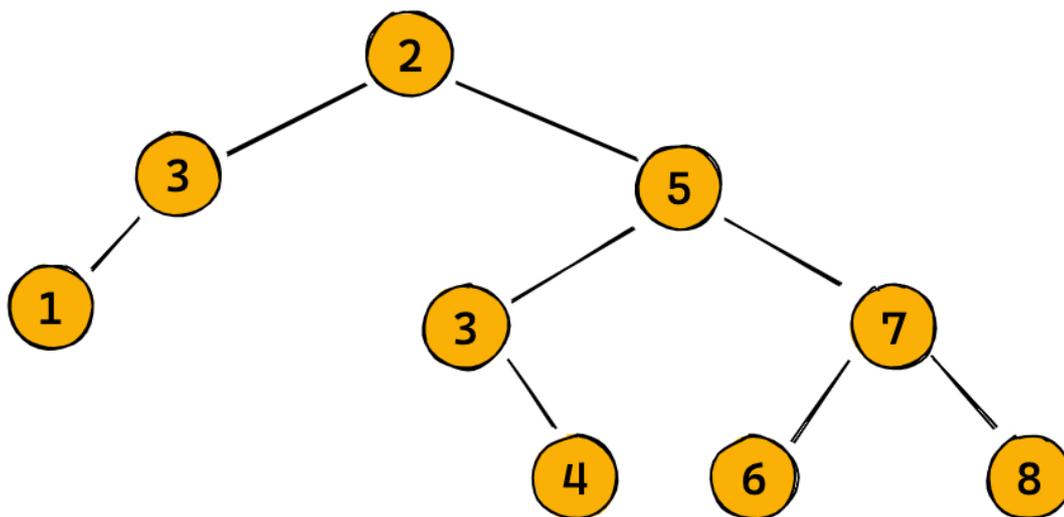
QUESTION 6: Write a function to calculate the height/depth of a binary tree

QUESTION 7: Write a function to count the number of nodes in a binary tree

The *height/depth* of a binary tree is defined as the length of the longest path from its root node to a leaf. It can be computed recursively, as follows:

```
def tree_height(node):  
    if node is None:  
        return 0  
    return 1 + max(tree_height(node.left), tree_height(node.right))
```

Let's compute the height of this tree:



```
tree_height(tree)
```

4

Here's a function to count the number of nodes in a binary tree.

```
def tree_size(node):
    if node is None:
        return 0
    return 1 + tree_size(node.left) + tree_size(node.right)
```

```
tree_size(tree)
```

9

Exercise: Try solving these problems relating to path lengths in a binary tree:

- <https://leetcode.com/problems/maximum-depth-of-binary-tree/>
- <https://leetcode.com/problems/minimum-depth-of-binary-tree/>
- <https://leetcode.com/problems/diameter-of-binary-tree/>

As a final step, let's compile all the functions we've written so far as methods within the `TreeNode` class itself. Encapsulation of data and functionality within the same class is a good programming practice.

```
class TreeNode():
    def __init__(self, key):
        self.key, self.left, self.right = key, None, None

    def height(self):
        if self is None:
            return 0
        return 1 + max(TreeNode.height(self.left), TreeNode.height(self.right))

    def size(self):
        if self is None:
            return 0
        return 1 + TreeNode.size(self.left) + TreeNode.size(self.right)

    def traverse_in_order(self):
        if self is None:
            return []
        return (TreeNode.traverse_in_order(self.left) +
                [self.key] +
                TreeNode.traverse_in_order(self.right))

    def display_keys(self, space='\t', level=0):
        # If the node is empty
        if self is None:
            print(space*level + '∅')
            return

        # If the node is a leaf
        if self.left is None and self.right is None:
            print(space*level + str(self.key))
```

```

        return

    # If the node has children
    display_keys(self.right, space, level+1)
    print(space*level + str(self.key))
    display_keys(self.left, space, level+1)

def to_tuple(self):
    if self is None:
        return None
    if self.left is None and self.right is None:
        return self.key
    return TreeNode.to_tuple(self.left), self.key, TreeNode.to_tuple(self.right)

def __str__(self):
    return "BinaryTree <{}>".format(self.to_tuple())

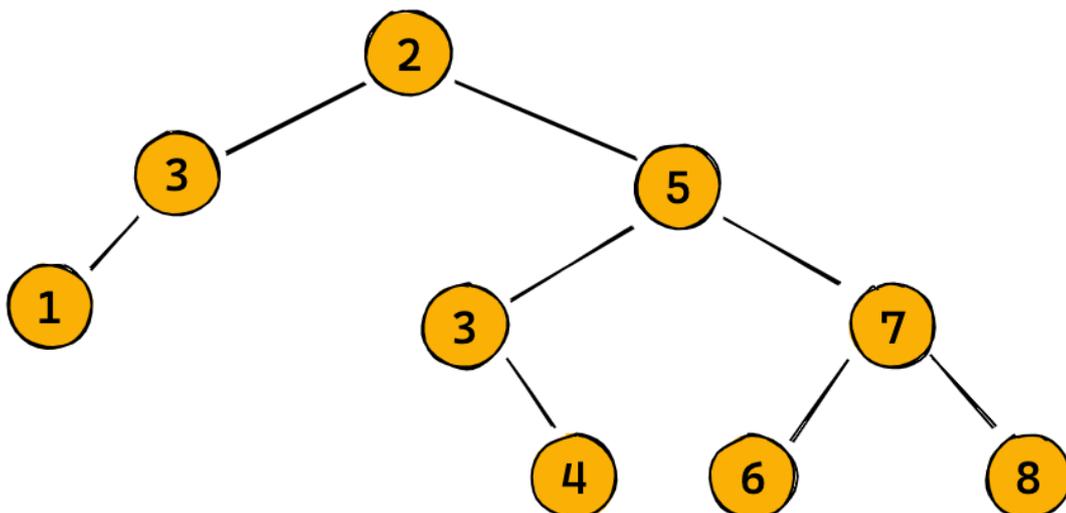
def __repr__(self):
    return "BinaryTree <{}>".format(self.to_tuple())

@staticmethod
def parse_tuple(data):
    if data is None:
        node = None
    elif isinstance(data, tuple) and len(data) == 3:
        node = TreeNode(data[1])
        node.left = TreeNode.parse_tuple(data[0])
        node.right = TreeNode.parse_tuple(data[2])
    else:
        node = TreeNode(data)
    return node

```

The class method invocations `TreeNode.height(node)` and `node.height()` are equivalent. Can you guess why we're using the former in the function definitions above? Hint: Track the recursive calls. Discuss on the forum: <https://jovian.ai/forum/c/data-structures-and-algorithms-in-python/lesson-2/82>

Let's try out the various methods defined above for this tree:



```
tree_tuple
```

```
((1, 3, None), 2, ((None, 3, 4), 5, (6, 7, 8)))
```

```
tree = TreeNode.parse_tuple(tree_tuple)
```

```
tree
```

```
BinaryTree <((1, 3, None), 2, ((None, 3, 4), 5, (6, 7, 8)))>
```

```
tree.display_keys('  ')
```

```
      8
     7
    6
   5
  4
 3
 ∅
2
 ∅
3
1
```

```
tree.height()
```

```
4
```

```
tree.size()
```

```
9
```

```
tree.traverse_in_order()
```

```
[1, 3, 2, 3, 4, 5, 6, 7, 8]
```

```
tree.to_tuple()
```

```
((1, 3, None), 2, ((None, 3, 4), 5, (6, 7, 8)))
```

Exercise: Create some more trees and try out the operations defined above. Add more operations to the `TreeNode` class.

Binary Search Tree (BST)

A binary search tree or BST is a binary tree that satisfies the following conditions:

1. The left subtree of any node only contains nodes with keys less than the node's key
2. The right subtree of any node only contains nodes with keys greater than the node's key

It follows from the above conditions that every subtree of a binary search tree must also be a binary search tree.

QUESTION 8: Write a function to check if a binary tree is a binary search tree (BST).

QUESTION 9: Write a function to find the maximum key in a binary tree.

QUESTION 10: Write a function to find the minimum key in a binary tree.

Here's a function that covers all of the above:

```
def remove_none(nums):
    return [x for x in nums if x is not None]

def is_bst(node):
    if node is None:
        return True, None, None

    is_bst_l, min_l, max_l = is_bst(node.left)
    is_bst_r, min_r, max_r = is_bst(node.right)

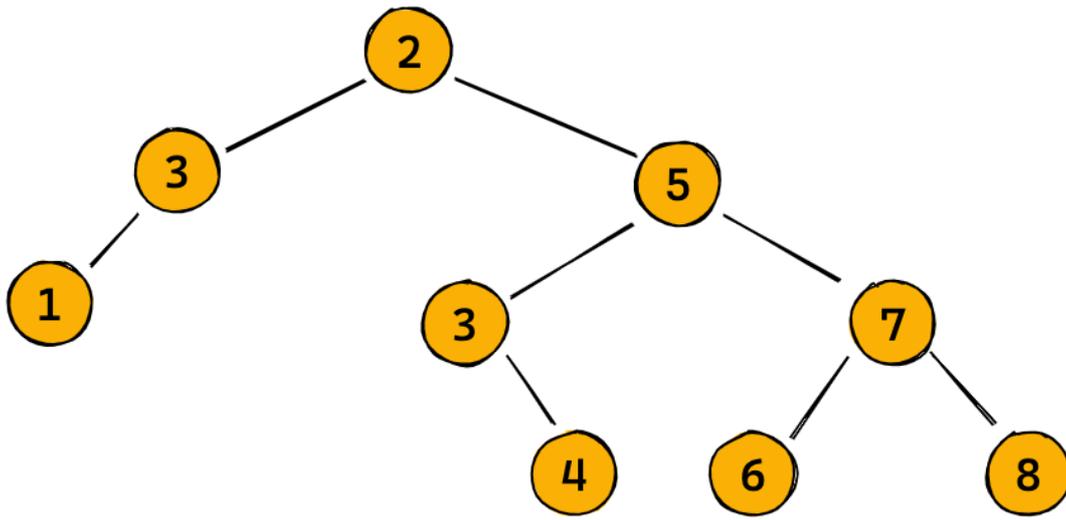
    is_bst_node = (is_bst_l and is_bst_r and
                   (max_l is None or node.key > max_l) and
                   (min_r is None or node.key < min_r))

    min_key = min(remove_none([min_l, node.key, min_r]))
    max_key = max(remove_none([max_l, node.key, max_r]))

    # print(node.key, min_key, max_key, is_bst_node)

    return is_bst_node, min_key, max_key
```

The following tree is not a BST (because a node with the key 3 appears in the left subtree of a node with the key 2):



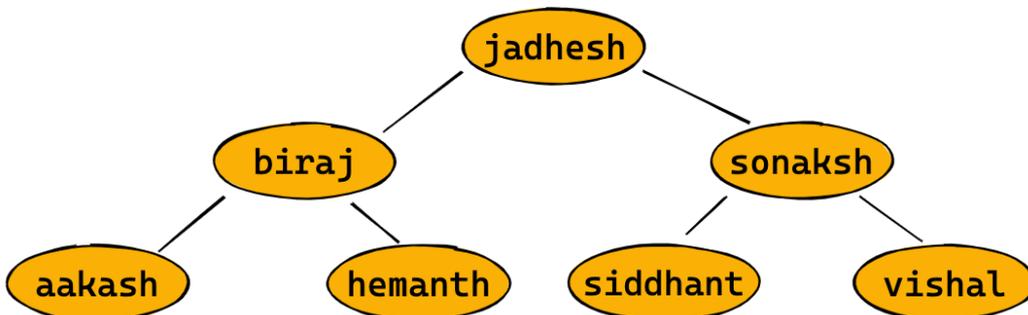
Let's verify this using `is_bst` .

```
tree1 = TreeNode.parse_tuple(((1, 3, None), 2, ((None, 3, 4), 5, (6, 7, 8))))
```

```
is_bst(tree1)
```

```
(False, 1, 8)
```

On the other hand, the following tree is a BST:



Let's create this tree and verify that it is a BST. Note that the `TreeNode` class also supports using strings as keys, as strings support the comparison operators `<` and `>` too.

```
tree2 = TreeNode.parse_tuple((('aakash', 'biraj', 'hemanth') , 'jadhesh', ('siddhant',
```

```
is_bst(tree2)
```

```
(True, 'aakash', 'vishal')
```

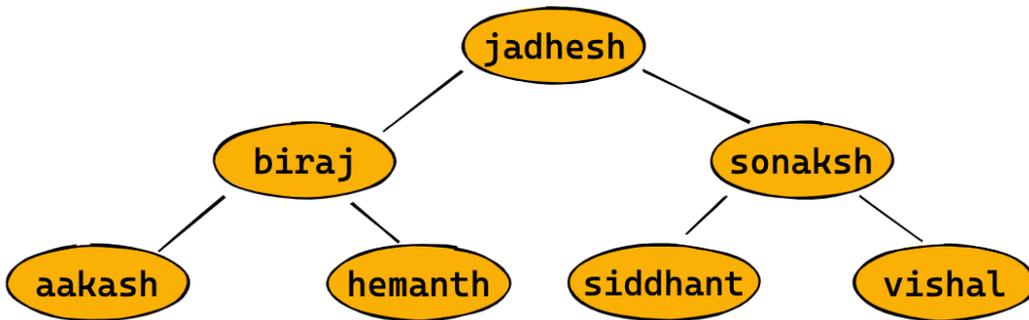
Exercise: Test the `is_bst` function with some more examples using the empty cells below.

Storing Key-Value Pairs using BSTs

Recall that we need to store user objects with each key in our BST. Let's define new class `BSTNode` to represent the nodes of our tree. Apart from having properties `key`, `left` and `right`, we'll also store a `value` and pointer to the parent node (for easier upward traversal).

```
class BSTNode():
    def __init__(self, key, value=None):
        self.key = key
        self.value = value
        self.left = None
        self.right = None
        self.parent = None
```

Let's try to recreate this BST with usernames as keys and user objects as values:



```
# Level 0
tree = BSTNode(jadhesh.username, jadhesh)
```

```
# View Level 0
tree.key, tree.value
```

```
('jadhesh',
 User(username='jadhesh', name='Jadhesh Verma', email='jadhesh@example.com'))
```

```
# Level 1
tree.left = BSTNode(biraj.username, biraj)
tree.right = BSTNode(sonaksh.username, sonaksh)
```

```
# View Level 1
tree.left.key, tree.left.value, tree.right.key, tree.right.value
```

```
('biraj',
 User(username='biraj', name='Biraj Das', email='biraj@example.com'),
 'sonaksh',
 User(username='sonaksh', name='Sonaksh Kumar', email='sonaksh@example.com'))
```

Exercise: Add the next layer of nodes to the tree and verify that they were added properly.

We can use the same `display_keys` function we defined earlier to visualize our tree.

```
display_keys(tree)
```

```
sonaksh
jadhesh
  biraj
```

Insertion into BST

QUESTION 11: Write a function to insert a new node into a BST.

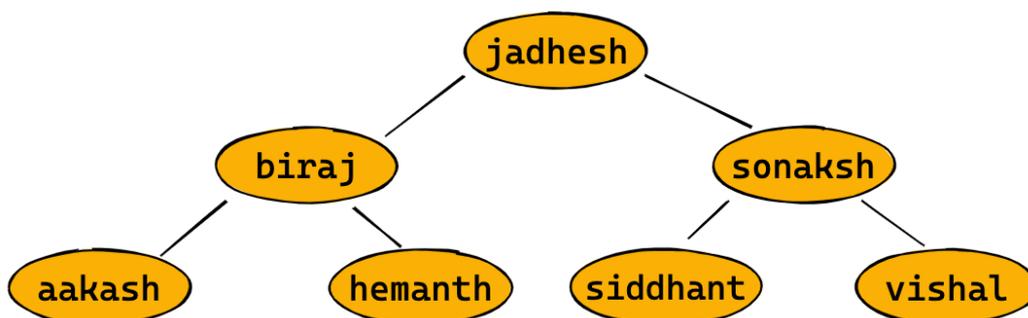
We use the BST-property to perform insertion efficiently:

1. Starting from the root node, we compare the key to be inserted with the current node's key
2. If the key is smaller, we recursively insert it in the left subtree (if it exists) or attach it as as the left child if no left subtree exists.
3. If the key is larger, we recursively insert it in the right subtree (if it exists) or attach it as as the right child if no right subtree exists.

Here's a recursive implementation of `insert` .

```
def insert(node, key, value):
    if node is None:
        node = BSTNode(key, value)
    elif key < node.key:
        node.left = insert(node.left, key, value)
        node.left.parent = node
    elif key > node.key:
        node.right = insert(node.right, key, value)
        node.right.parent = node
    return node
```

Let's use this to recreate our tree.



To create the first node, we can use the `insert` function with `None` as the target tree.

```
tree = insert(None, jadhesh.username, jadhesh)
```

The remaining nodes can now be inserted into `tree` .

```
insert(tree, biraj.username, biraj)
insert(tree, sonaksh.username, sonaksh)
insert(tree, aakash.username, aakash)
insert(tree, hemanth.username, hemanth)
insert(tree, siddhant.username, siddhant)
insert(tree, vishal.username, siddhant)
```

```
<__main__.BSTNode at 0x7faf03428ba8>
```

```
display_keys(tree)
```

```
    vishal
  sonaksh
    siddhant
jadhesh
    hemanth
  biraj
    aakash
```

Perfect! The tree was created as expected.

Note, however, that the order of insertion of nodes change the structure of the resulting tree.

```
tree2 = insert(None, aakash.username, aakash)
insert(tree2, biraj.username, biraj)
insert(tree2, hemanth.username, hemanth)
insert(tree2, jadhesh.username, jadhesh)
insert(tree2, siddhant.username, siddhant)
insert(tree2, sonaksh.username, sonaksh)
insert(tree2, vishal.username, vishal)
```

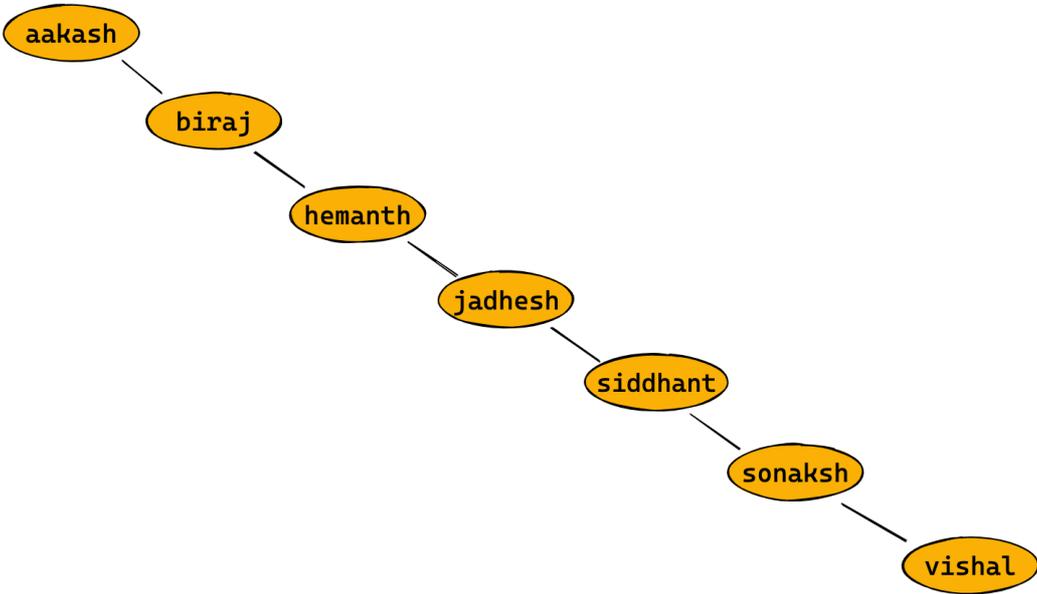
```
<__main__.BSTNode at 0x7faf03417080>
```

```
display_keys(tree2)
```

```
    vishal
  sonaksh
    ∅
  siddhant
    ∅
  jadhesh
    ∅
  hemanth
```

∅
biraj
∅
aakash
∅

Can you see why the tree created above is skewed/unbalanced?



Skewed/unbalanced BSTs are problematic because the height of such trees often ceases to be logarithmic compared to the number of nodes in the tree. For instance the above tree has 7 nodes and height 7.

The length of the path traversed by `insert` is equal to the height of the tree (in the worst case). It follows that if the tree is balanced, the time complexity of insertion is $O(\log N)$ otherwise it is $O(N)$.

```
tree_height(tree2)
```

7

Exercise: Create some more balanced and unbalanced BSTs using the `insert` function defined above.

Finding a Node in BST

QUESTION 11: Find the value associated with a given key in a BST.

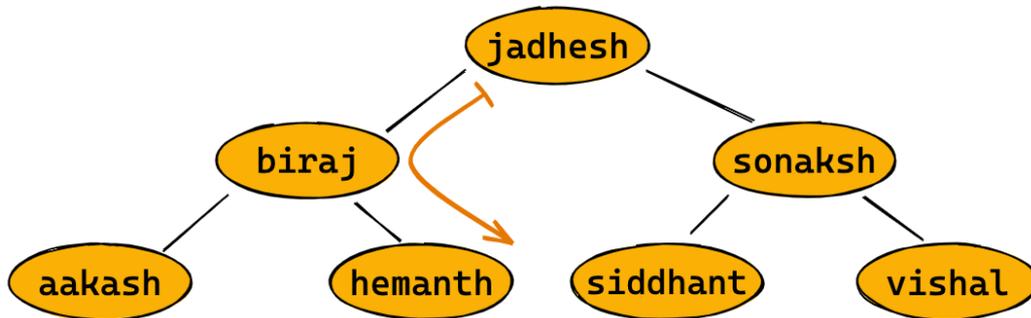
We can follow a recursive strategy similar to insertion to find the node with a given key within a BST.

```
def find(node, key):  
    if node is None:  
        return None
```

```

if key == node.key:
    return node
if key < node.key:
    return find(node.left, key)
if key > node.key:
    return find(node.right, key)

```



```
node = find(tree, 'hemanth')
```

```
node.key, node.value
```

```

('hemanth',
 User(username='hemanth', name='Hemanth Jain', email='hemanth@example.com'))

```

The length of the path followed by `find` is equal to the height of the tree (in the worst case). Thus it has a similar time complexity as `insert`.

Example: Try finding some more nodes from the BST created above (or create new trees).

Updating a value in a BST

QUESTION 12: Write a function to update the value associated with a given key within a BST

We can use `find` to locate the node to be updated, and simply update its value.

```

def update(node, key, value):
    target = find(node, key)
    if target is not None:
        target.value = value

```

```
update(tree, 'hemanth', User('hemanth', 'Hemanth J', 'hemanthj@example.com'))
```

```

node = find(tree, 'hemanth')
node.value

```

```
User(username='hemanth', name='Hemanth J', email='hemanthj@example.com')
```

The value of the node was successfully updated. The time complexity of `update` is the same as that of `find`.

Exercise: Try some more update operations using the BST created earlier.

List the nodes

QUESTION 13: Write a function to retrieve all the key-values pairs stored in a BST in the sorted order of keys.

The nodes can be listed in sorted order by performing an inorder traversal of the BST.

```
def list_all(node):  
    if node is None:  
        return []  
    return list_all(node.left) + [(node.key, node.value)] + list_all(node.right)
```

```
list_all(tree)
```

```
[('aakash',  
  User(username='aakash', name='Aakash Rai', email='aakash@example.com')),  
( 'biraj',  
  User(username='biraj', name='Biraj Das', email='biraj@example.com')),  
( 'hemanth',  
  User(username='hemanth', name='Hemanth J', email='hemanthj@example.com')),  
( 'jadhesh',  
  User(username='jadhesh', name='Jadhesh Verma', email='jadhesh@example.com')),  
( 'siddhant',  
  User(username='siddhant', name='Siddhant U', email='siddhantu@example.com')),  
( 'sonaksh',  
  User(username='sonaksh', name='Sonaksh Kumar', email='sonaksh@example.com')),  
( 'vishal',  
  User(username='siddhant', name='Siddhant U', email='siddhantu@example.com'))]
```

Exercise: Determine the time complexity and space complexity of `list_all`.

```
import jovian
```

```
jovian.commit()
```

```
[jovian] Attempting to save notebook..
```

[jovian] Updating notebook "aakashns/python-binary-search-trees" on <https://jovian.ai/>
[jovian] Uploading notebook..
[jovian] Capturing environment..
[jovian] Committed successfully! <https://jovian.ai/aakashns/python-binary-search-trees>
'<https://jovian.ai/aakashns/python-binary-search-trees>'

Balanced Binary Trees

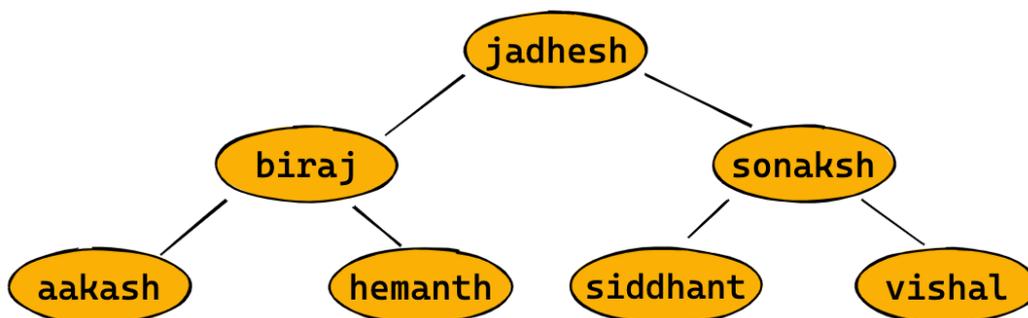
QUESTION 14: Write a function to determine if a binary tree is balanced.

Here's a recursive strategy:

1. Ensure that the left subtree is balanced.
2. Ensure that the right subtree is balanced.
3. Ensure that the difference between heights of left subtree and right subtree is not more than 1.

```
def is_balanced(node):  
    if node is None:  
        return True, 0  
    balanced_l, height_l = is_balanced(node.left)  
    balanced_r, height_r = is_balanced(node.right)  
    balanced = balanced_l and balanced_r and abs(height_l - height_r) <= 1  
    height = 1 + max(height_l, height_r)  
    return balanced, height
```

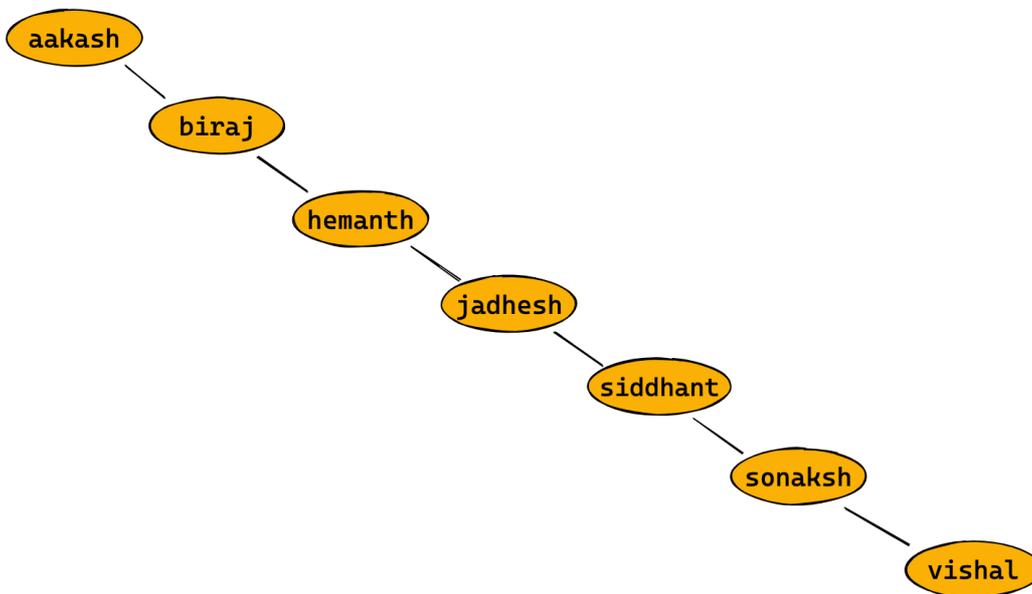
The following tree is balanced:



```
is_balanced(tree)
```

```
(True, 3)
```

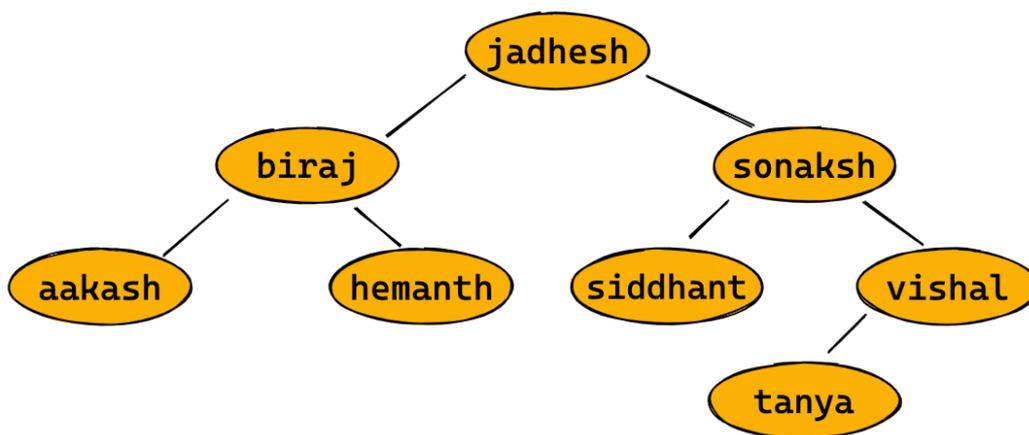
The following tree is not balanced:



```
is_balanced(tree2)
```

(False, 7)

Exercise: Is the tree shown below balanced? Why or why not? Create this tree and check if it's balanced using the `is_balanced` function.



Also try this related problem on *complete* binary trees: <https://leetcode.com/problems/check-completeness-of-a-binary-tree/>

Balanced Binary Search Trees

QUESTION 15: Write a function to create a balanced BST from a sorted list/array of key-value pairs.

We can use a recursive strategy here, turning the middle element of the list into the root, and recursively creating left and right subtrees.

```
def make_balanced_bst(data, lo=0, hi=None, parent=None):
    if hi is None:
        hi = len(data) - 1
    if lo > hi:
        return None

    mid = (lo + hi) // 2
    key, value = data[mid]

    root = BSTNode(key, value)
    root.parent = parent
    root.left = make_balanced_bst(data, lo, mid-1, root)
    root.right = make_balanced_bst(data, mid+1, hi, root)

    return root
```

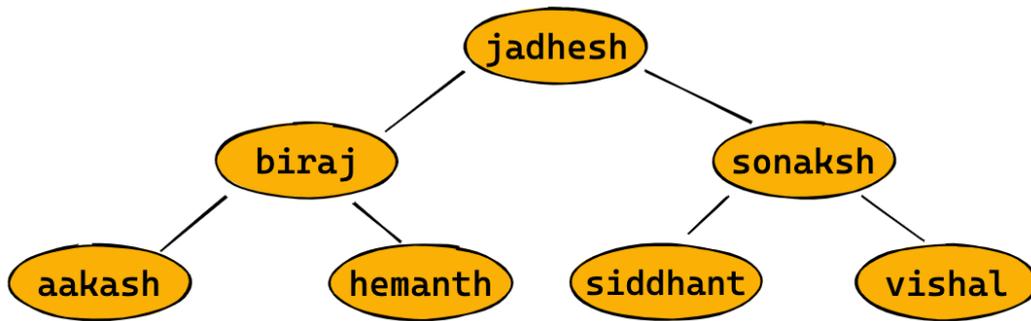
```
data = [(user.username, user) for user in users]
data
```

```
[('aakash',
  User(username='aakash', name='Aakash Rai', email='aakash@example.com')),
 ('biraj',
  User(username='biraj', name='Biraj Das', email='biraj@example.com')),
 ('hemanth',
  User(username='hemanth', name='Hemanth Jain', email='hemanth@example.com')),
 ('jadhesh',
  User(username='jadhesh', name='Jadhesh Verma', email='jadhesh@example.com')),
 ('siddhant',
  User(username='siddhant', name='Siddhant U', email='siddhantu@example.com')),
 ('sonaksh',
  User(username='sonaksh', name='Sonaksh Kumar', email='sonaksh@example.com')),
 ('vishal',
  User(username='vishal', name='Vishal Goel', email='vishal@example.com'))]
```

```
tree = make_balanced_bst(data)
```

```
display_keys(tree)
```

```
    vishal
  sonaksh
    siddhant
jadhesh
    hemanth
  biraj
    aakash
```



Recall that the same list of users, when inserted one-by-one resulted in a skewed tree.

```

tree3 = None
for username, user in data:
    tree3 = insert(tree3, username, user)
  
```

Balancing an Unbalanced BST

QUESTION 16: Write a function to balance an unbalanced binary search tree.

We first perform an inorder traversal, then create a balanced BST using the function defined earlier.

```

def balance_bst(node):
    return make_balanced_bst(list_all(node))
  
```

```

tree1 = None

for user in users:
    tree1 = insert(tree1, user.username, user)
  
```

```

display_keys(tree1)
  
```

```

          vishal
        sonaksh
         ∅
      siddhant
         ∅
    jadhesh
         ∅
  
```

```
    hemanth
      ∅
    biraj
      ∅
aakash
  ∅
```

```
tree2 = balance_bst(tree1)
```

```
display_keys(tree2)
```

```
    vishal
  sonaksh
    siddhant
jadhesh
  hemanth
    biraj
    aakash
```

After every insertion, we can balance the tree. This way the tree will remain balanced.

Complexity of the various operations in a balanced BST:

- Insert - $O(\log N) + O(N) = O(N)$
- Find - $O(\log N)$
- Update - $O(\log N)$
- List all - $O(N)$

What's the real improvement between $O(N)$ and $O(\log N)$?

```
import math

math.log(100000000, 2)
```

```
26.5754247590989
```

The logarithm (base 2) of 100 million is around 26. Thus, it takes only 26 operations to find or update a node within a BST (as opposed to 100 million).

```
%%time
for i in range(26):
    j = i*i
```

```
CPU times: user 8 µs, sys: 1e+03 ns, total: 9 µs
```

```
Wall time: 14.1 µs
```

Compared to linear time:

```
%%time
for i in range(100000000):
    j = i*i
```

CPU times: user 8.85 s, sys: 10.2 ms, total: 8.86 s

Wall time: 8.87 s

Thus, find and update from a balanced binary search tree is 300,000 times faster than our original solution. To speed up insertions, we may choose to perform the balancing periodically (e.g. once every 1000 insertions). This way, most insertions will be $O(\log N)$, but every 1000th insertion will take a few seconds. Another options is to rebalance the tree periodically at the end of every hour.

A Python-Friendly Treemap

We are now ready to return to our original problem statement.

QUESTION 1: As a senior backend engineer at Jovian, you are tasked with developing a fast in-memory data structure to manage profile information (username, name and email) for 100 million users. It should allow the following operations to be performed efficiently:

1. **Insert** the profile information for a new user.
2. **Find** the profile information of a user, given their username
3. **Update** the profile information of a user, given their usname
4. **List** all the users of the platform, sorted by username

You can assume that usernames are unique.

We can create a generic class `TreeMap` which supports all the operations specified in the original problem statement in a python-friendly manner.

```
class TreeMap():
    def __init__(self):
        self.root = None

    def __setitem__(self, key, value):
        node = find(self.root, key)
        if not node:
            self.root = insert(self.root, key, value)
            self.root = balance_bst(self.root)
        else:
            update(self.root, key, value)

    def __getitem__(self, key):
        node = find(self.root, key)
        return node.value if node else None

    def __iter__(self):
        return (x for x in list_all(self.root))
```

```
def __len__(self):
    return tree_size(self.root)

def display(self):
    return display_keys(self.root)
```

Exercise: What is the time complexity of `__len__` ? Can you reduce it to $O(1)$. Hint: Modify the `BSTNode` class.

Let's try using the `TreeMap` class below.

```
users
```

```
[User(username='aakash', name='Aakash Rai', email='aakash@example.com'),
 User(username='biraj', name='Biraj Das', email='biraj@example.com'),
 User(username='hemanth', name='Hemanth Jain', email='hemanth@example.com'),
 User(username='jadhesh', name='Jadhesh Verma', email='jadhesh@example.com'),
 User(username='siddhant', name='Siddhant U', email='siddhantu@example.com'),
 User(username='sonaksh', name='Sonaksh Kumar', email='sonaksh@example.com'),
 User(username='vishal', name='Vishal Goel', email='vishal@example.com')]
```

```
treemap = TreeMap()
```

```
treemap.display()
```

∅

```
treemap['aakash'] = aakash
treemap['jadhesh'] = jadhesh
treemap['sonaksh'] = sonaksh
```

```
treemap.display()
```

```
sonaksh
jadhesh
aakash
```

```
treemap['jadhesh']
```

```
User(username='jadhesh', name='Jadhesh Verma', email='jadhesh@example.com')
```

```
len(treemap)
```

3

```
treemap['biraj'] = biraj
treemap['hemanth'] = hemanth
```

```
treemap['siddhant'] = siddhant
treemap['vishal'] = vishal
```

```
treemap.display()
```

```
    vishal
  sonaksh
    siddhant
jadhesh
    hemanth
  biraj
    aakash
```

```
for key, value in treemap:
    print(key, value)
```

```
aakash User(username='aakash', name='Aakash Rai', email='aakash@example.com')
biraj User(username='biraj', name='Biraj Das', email='biraj@example.com')
hemanth User(username='hemanth', name='Hemanth Jain', email='hemanth@example.com')
jadhesh User(username='jadhesh', name='Jadhesh Verma', email='jadhesh@example.com')
siddhant User(username='siddhant', name='Siddhant U', email='siddhantu@example.com')
sonaksh User(username='sonaksh', name='Sonaksh Kumar', email='sonaksh@example.com')
vishal User(username='vishal', name='Vishal Goel', email='vishal@example.com')
```

```
list(treemap)
```

```
[('aakash',
  User(username='aakash', name='Aakash Rai', email='aakash@example.com')),
 ('biraj',
  User(username='biraj', name='Biraj Das', email='biraj@example.com')),
 ('hemanth',
  User(username='hemanth', name='Hemanth Jain', email='hemanth@example.com')),
 ('jadhesh',
  User(username='jadhesh', name='Jadhesh Verma', email='jadhesh@example.com')),
 ('siddhant',
  User(username='siddhant', name='Siddhant U', email='siddhantu@example.com')),
 ('sonaksh',
  User(username='sonaksh', name='Sonaksh Kumar', email='sonaksh@example.com')),
 ('vishal',
  User(username='vishal', name='Vishal Goel', email='vishal@example.com'))]
```

```
treemap['aakash'] = User(username='aakash', name='Aakash N S', email='aakashns@example.com')
```

```
treemap['aakash']
```

```
User(username='aakash', name='Aakash N S', email='aakashns@example.com')
```

Exercise: Try out some more examples below. Can our treemap actually handle millions of users profiles?

Let's save our work before cotinuing.

```
import jovian
```

```
jovian.commit()
```

```
[jovian] Attempting to save notebook..
```

```
[jovian] Updating notebook "aakashns/python-binary-search-trees" on https://jovian.ai/
```

```
[jovian] Uploading notebook..
```

```
[jovian] Capturing environment..
```

```
[jovian] Committed successfully! https://jovian.ai/aakashns/python-binary-search-trees
```

```
'https://jovian.ai/aakashns/python-binary-search-trees'
```

Self-Balancing Binary Trees and AVL Trees

A *self-balancing binary tree* remains balanced after every insertion or deletion. Several decades of research has gone into creating self-balancing binary trees, and many approaches have been devised e.g. B-trees, Red Black Trees and AVL (Adelson-Velsky Landis) trees.

We'll take a brief look at AVL trees. Self-balancing in AVL trees is achieved by tracking the *balance factor* (difference between the height of the left subtree and the right subtree) for each node and *rotating* unbalanced subtrees along the path of insertion/deletion to balance them.



In a balanced BST, the balance factor of each node is either 0, -1, or 1. When we perform an insertion, then the balance factor of certain nodes along the path of insertion may change to 2 or -2. Those nodes can be "rotated" one-by-one to bring the balance factor back to 1, 0 or -1.

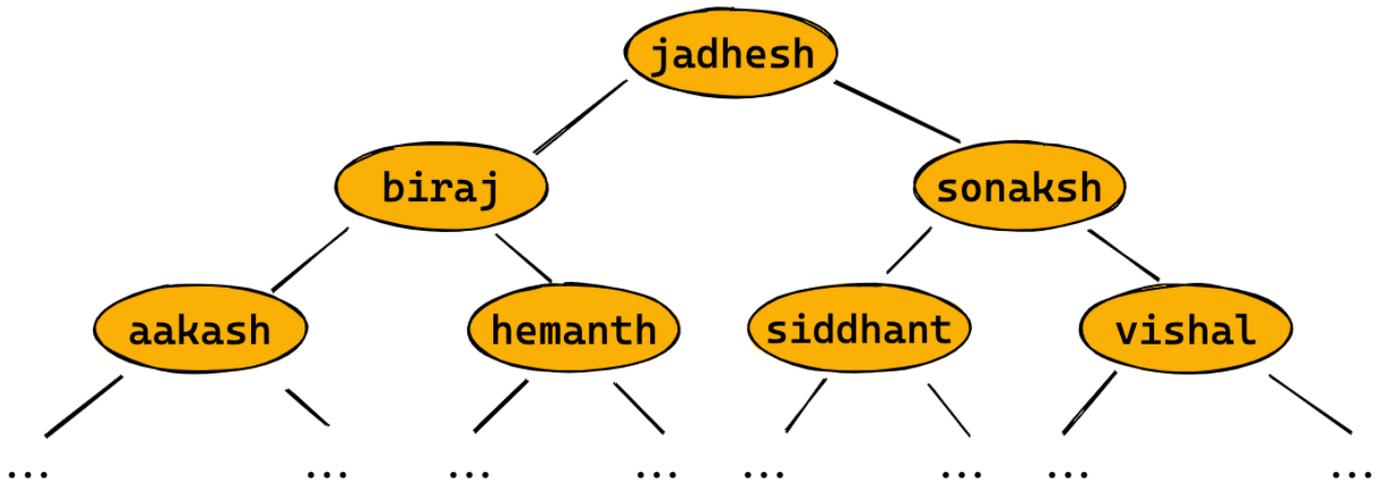
There are 4 different scenarios for balancing, two of which require a single rotation, while the others require 2 rotations:

Source: [HackerRank](#)

Since each rotation takes constant time, and at most $\log N$ rotations may be required, this operation is far more efficient than creating a balanced binary tree from scratch, allowing insertion and deletion to be performed in $O(\log N)$ time. Here are some references for AVL Trees:

- Explanation of the various cases: https://youtu.be/jDM6_TnYlqE?t=482
- Implementation: <https://www.geeksforgeeks.org/avl-tree-set-1-insertion/>

Summary and Exercises



Binary trees form the basis of many modern programming language features (e.g. maps in C++ and Java) and data storage systems (filesystem indexes, relational databases like MySQL). You might wonder if dictionaries in Python are also binary search trees. They're not. They're hash tables, which is a different but equally interesting and important data structure. We'll explore hash tables in a future tutorial.

We've covered a lot of ground this in this tutorial, including several common interview questions. Here are a few more problems you can try out:

1. Implement rotations and self-balancing insertion
2. Implement deletion of a node from a binary search tree
3. Implement deletion of a node from a BST (with balancing)
4. Find the lowest common ancestor of two nodes in a tree (Hint: Use the parent property)
5. Find the next node in lexicographic order for a given node
6. Given a number k , find the k -th node in a BST.

Try more questions here:

- <https://medium.com/techie-delight/binary-tree-interview-questions-and-practice-problems-439df7e5ea1f>
- <https://leetcode.com/tag/tree/>

```
import jovian
```

```
jovian.commit()
```

[jovian] Attempting to save notebook..

Assignment 2 - Hash Tables in Python

This assignment is a part of the course ["Data Structures and Algorithms in Python"](#).

In this assignment, you will implement Python dictionaries from scratch using hash tables.

As you go through this notebook, you will find the symbol ??? in certain places. To complete this assignment, you must replace all the ??? with appropriate values, expressions or statements to ensure that the notebook runs properly end-to-end.

Guidelines

1. Make sure to run all the code cells, otherwise you may get errors like `NameError` for undefined variables.
2. Do not change variable names, delete cells or disturb other existing code. It may cause problems during evaluation.
3. In some cases, you may need to add some code cells or new statements before or after the line of code containing the ???.
4. Since you'll be using a temporary online service for code execution, save your work by running `jovian.commit` at regular intervals.
5. Questions marked (Optional) will not be considered for evaluation, and can be skipped. They are for your learning.
6. If you are stuck, you can ask for help on the [community forum](#). Post errors or ask for hints, but **please don't ask for OR share the full working answer code** on the forum.
7. There are some tests included with this notebook to help you test your implementation. However, after submission your code will be tested with some hidden test cases. Make sure to test your code exhaustively to cover all edge cases.

Important Links

- Submit your work here: <https://jovian.ai/learn/data-structures-and-algorithms-in-python/assignment/assignment-2-hash-table-and-python-dictionaries>
- Ask questions and get help: <https://jovian.ai/forum/c/data-structures-and-algorithms-in-python/assignment-2/88>
- Lesson 2 video for review: <https://jovian.ai/aakashns/python-binary-search-trees>
- Lesson 2 notebook for review: <https://jovian.ai/aakashns/python-binary-search-trees>

How to Run the Code and Save Your Work

Option 1: Running using free online resources (1-click, recommended): Click the **Run** button at the top of this page and select **Run on Binder**. You can also select "Run on Colab" or "Run on Kaggle", but you'll need to create an account on [Google Colab](#) or [Kaggle](#) to use these platforms.

Option 2: Running on your computer locally: To run the code on your computer locally, you'll need to set up [Python](#) & [Conda](#), download the notebook and install the required libraries. Click the **Run** button at the top of this page, select the **Run Locally** option, and follow the instructions.

Saving your work: You can save a snapshot of the assignment to your [Jovian](#) profile, so that you can access it later and continue your work. Keep saving your work by running `jovian.commit` from time to time.

```
project='python-hash-tables-assignment'
```

```
!pip install jovian --upgrade --quiet
```

```
import jovian
jovian.commit(project=project, privacy='secret', environment=None)
```

[jovian] Updating notebook "evanmarie/python-hash-tables-assignment" on <https://jovian.ai>

[jovian] Committed successfully! <https://jovian.ai/evanmarie/python-hash-tables-assignment>

'<https://jovian.ai/evanmarie/python-hash-tables-assignment>'

Problem Statement - Python Dictionaries and Hash Tables

In this assignment, you will recreate Python dictionaries from scratch using data structure called *hash table*. Dictionaries in Python are used to store key-value pairs. Keys are used to store and retrieve values. For example, here's a dictionary for storing and retrieving phone numbers using people's names.

```
phone_numbers = {
    'Aakash' : '9489484949',
    'Hemanth' : '9595949494',
    'Siddhant' : '9231325312'
}
phone_numbers
```

```
{'Aakash': '9489484949', 'Hemanth': '9595949494', 'Siddhant': '9231325312'}
```

You can access a person's phone number using their name as follows:

```
phone_numbers['Aakash']
```

```
'9489484949'
```

You can store new phone numbers, or update existing ones as follows:

```
# Add a new value
phone_numbers['Vishal'] = '8787878787'
# Update existing value
phone_numbers['Aakash'] = '7878787878'
# View the updated dictionary
phone_numbers
```

```
{'Aakash': '7878787878',
 'Hemanth': '9595949494',
 'Siddhant': '9231325312',
 'Vishal': '8787878787'}
```

You can also view all the names and phone numbers stored in `phone_numbers` using a `for` loop.

```
for name in phone_numbers:  
    print('Name:', name, ', Phone Number:', phone_numbers[name])
```

Name: Aakash , Phone Number: 7878787878

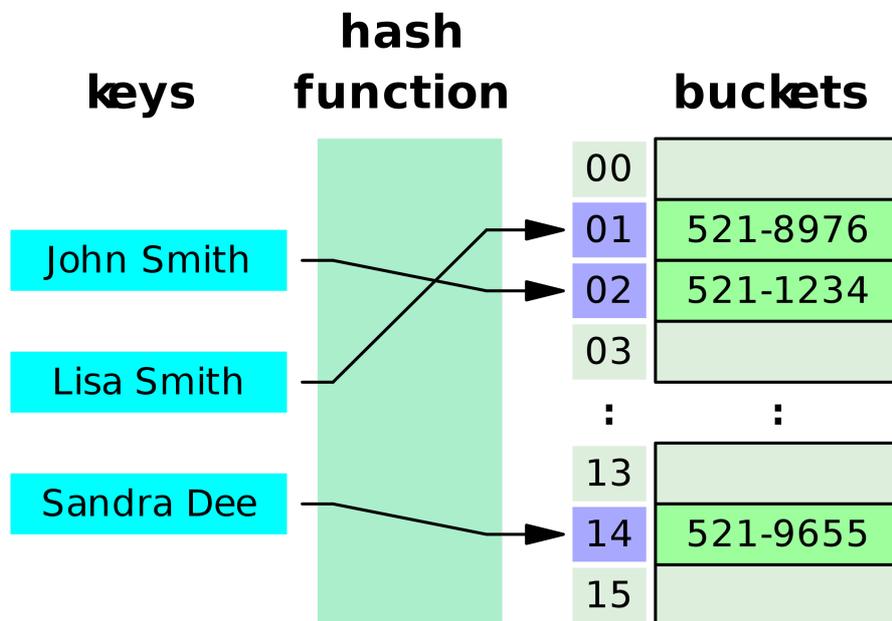
Name: Hemanth , Phone Number: 9595949494

Name: Siddhant , Phone Number: 9231325312

Name: Vishal , Phone Number: 8787878787

Dictionaries in Python are implemented using a data structure called **hash table**. A hash table uses a list/array to store the key-value pairs, and uses a *hashing function* to determine the index for storing or retrieving the data associated with a given key.

Here's a visual representation of a hash table ([source](#)):



Your objective in this assignment is to implement a `HashTable` class which supports the following operations:

1. **Insert:** Insert a new key-value pair
2. **Find:** Find the value associated with a key
3. **Update:** Update the value associated with a key
4. **List:** List all the keys stored in the hash table

The `HashTable` class will have the following structure (note the function signatures):

```
class HashTable:  
    def insert(self, key, value):  
        """Insert a new key-value pair"""  
        pass  
  
    def find(self, key):  
        """Find the value associated with a key"""
```

```
    pass

def update(self, key, value):
    """Change the value associated with a key"""
    pass

def list_all(self):
    """List all the keys"""
    pass
```

Before we begin our implementation, let's save and commit our work.

```
jovian.commit(project=project)
```

```
[jovian] Updating notebook "evanmarie/python-hash-tables-assignment" on
https://jovian.ai
```

```
[jovian] Committed successfully! https://jovian.ai/evanmarie/python-hash-tables-assignment
```

```
'https://jovian.ai/evanmarie/python-hash-tables-assignment'
```

Data List

We'll build the `HashTable` class step-by-step. As a first step is to create a Python list which will hold all the key-value pairs. We'll start by creating a list of a fixed size.

```
MAX_HASH_TABLE_SIZE = 4096
```

QUESTION 1: Create a Python list of size `MAX_HASH_TABLE_SIZE`, with all the values set to `None`.

Hint: Use the [* operator](#).

```
# List of size MAX_HASH_TABLE_SIZE with all values None
data_list = [None] * 4096
```

If the list was created successfully, the following cells should output `True`.

```
len(data_list) == 4096
```

True

```
data_list[99] == None
```

True

Let's save our work before continuing.

```
jovian.commit(project=project)
```

```
[jovian] Updating notebook "evanmarie/python-hash-tables-assignment" on
```

<https://jovian.ai>

[jovian] Committed successfully! <https://jovian.ai/evanmarie/python-hash-tables-assignment>

'<https://jovian.ai/evanmarie/python-hash-tables-assignment>'

Hashing Function

A *hashing function* is used to convert strings and other non-numeric data types into numbers, which can then be used as list indices. For instance, if a hashing function converts the string "Aakash" into the number 4, then the key-value pair ('Aakash', '7878787878') will be stored at the position 4 within the data list.

Here's a simple algorithm for hashing, which can convert strings into numeric list indices.

1. Iterate over the string, character by character
2. Convert each character to a number using Python's built-in `ord` function.
3. Add the numbers for each character to obtain the hash for the entire string
4. Take the remainder of the result with the size of the data list

QUESTION 2: Complete the `get_index` function below which implements the hashing algorithm described above.

```
def get_index(data_list, a_string):  
    # Variable to store the result (updated after each iteration)  
    result = 0  
  
    for a_character in a_string:  
        # Convert the character to a number (using ord)  
        a_number = ord(a_character)  
        # Update result by adding the number  
        result += a_number  
  
    # Take the remainder of the result with the size of the data list  
    list_index = result % len(data_list)  
    return list_index
```

If the `get_index` function was defined correctly, the following cells should output `True` .

```
get_index(data_list, '') == 0
```

True

```
get_index(data_list, 'Aakash') == 585
```

True

```
get_index(data_list, 'Don O Leary') == 941
```

True

(Optional) Test the `get_index` function using the empty cells below.

```
get_index(data_list, 'apple')
```

530

```
get_index(data_list, 'chewing gum')
```

1102

Insert

To insert a key-value pair into a hash table, we can simply get the hash of the key, and store the pair at that index in the data list.

```
key, value = 'Aakash', '7878787878'
```

```
idx = get_index(data_list, key)  
idx
```

585

```
data_list[idx] = (key, value)
```

Here's the same operation expressed in a single line of code.

```
data_list[get_index(data_list, 'Hemanth')] = ('Hemanth', '9595949494')
```

Find

To retrieve the value associated with a pair, we can get the hash of the key and look up that index in the data list.

```
idx = get_index(data_list, 'Aakash')  
idx
```

585

```
key, value = data_list[idx]  
value
```

'7878787878'

List

To get the list of keys, we can use a simple [list comprehension](#).

```
pairs = [kv[0] for kv in data_list if kv is not None]
```

```
pairs
```

```
['Aakash', 'Hemanth']
```

Let's save our work before continuing.

```
jovian.commit(project=project)
```

```
[jovian] Updating notebook "evanmarie/python-hash-tables-assignment" on  
https://jovian.ai
```

```
[jovian] Committed successfully! https://jovian.ai/evanmarie/python-hash-tables-assignment
```

```
'https://jovian.ai/evanmarie/python-hash-tables-assignment'
```

Basic Hash Table Implementation

We can now use the hashing function defined above to implement a basic hash table in Python.

QUESTION 3: Complete the hash table implementation below by following the instructions in the comments.

Hint: Insert and update can have identical implementations.

```
class BasicHashTable:
    def __init__(self, max_size=MAX_HASH_TABLE_SIZE):
        # 1. Create a list of size `max_size` with all values None
        self.data_list = [None] * max_size

    def insert(self, key, value):
        # 1. Find the index for the key using get_index
        idx = get_index(self.data_list, key)

        # 2. Store the key-value pair at the right index
        self.data_list[idx] = (key, value)

    def find(self, key):
        # 1. Find the index for the key using get_index
        idx = get_index(self.data_list, key)

        # 2. Retrieve the data stored at the index
        kv = self.data_list[idx]

        # 3. Return the value if found, else return None
        if kv is None:
            return None
        else:
            key, value = kv
            return value

    def update(self, key, value):
        # 1. Find the index for the key using get_index
```

```

idx = get_index(self.data_list, key)

# 2. Store the new key-value pair at the right index
self.data_list[idx] = (key, value)

def list_all(self):
    # 1. Extract the key from each key-value pair
    return [kv[0] for kv in self.data_list if kv is not None]

```

If the `BasicHashTable` class was defined correctly, the following cells should output `True` .

```

basic_table = BasicHashTable(max_size=1024)
len(basic_table.data_list) == 1024

```

True

```

# Insert some values
basic_table.insert('Aakash', '9999999999')
basic_table.insert('Hemanth', '8888888888')

# Find a value
basic_table.find('Hemanth') == '8888888888'

```

True

```

# Update a value
basic_table.update('Aakash', '7777777777')

# Check the updated value
basic_table.find('Aakash') == '7777777777'

```

True

```

# Get the list of keys
basic_table.list_all() == ['Aakash', 'Hemanth']

```

True

(Optional) Test your implementation of `BasicHashTable` with some more examples below.

```

basic_table.insert('Peppa', '09274298492')
basic_table.insert('Bridget', '484792942784')
basic_table.insert('Toots', '83726903827')
basic_table.insert('Felix', '027486929047')
basic_table.insert('Charlie', '3828467390')
basic_table.insert('Moody', '102938476')

```

```

basic_table.list_all()

```

```

['Peppa', 'Felix', 'Moody', 'Toots', 'Aakash', 'Charlie', 'Bridget', 'Hemanth']

```

```
basic_table.find('Charlie')
```

```
'3828467390'
```

```
basic_table.find('Peppa')
basic_table.update('Peppa', '6666666666')
basic_table.find('Peppa') == '6666666666'
```

```
True
```

```
basic_table.find('Charlie')
basic_table.update('Charlie', '8888888888')
basic_table.find('Charlie') == '8888888888'
```

```
True
```

```
basic_table.find('Moody')
basic_table.update('Moody', '9999999999')
basic_table.find('Moody') == '9999999999'
```

```
True
```

Let's save our work before continuing.

```
jovian.commit(project=project)
```

```
[jovian] Updating notebook "evanmarie/python-hash-tables-assignment" on
```

```
https://jovian.ai
```

```
[jovian] Committed successfully! https://jovian.ai/evanmarie/python-hash-tables-assignment
```

```
'https://jovian.ai/evanmarie/python-hash-tables-assignment'
```

Handling Collisions with Linear Probing

As you might have wondered, multiple keys can have the same hash. For instance, the keys "listen" and "silent" have the same hash. This is referred to as *collision*. Data stored against one key may override the data stored against another, if they have the same hash.

```
basic_table.insert('listen', 99)
```

```
basic_table.insert('silent', 200)
```

```
basic_table.find('listen')
```

```
200
```

As you can see above, the value for the key `listen` was overwritten by the value for the key `silent`. Our hash table implementation is incomplete because it does not handle collisions correctly.

To handle collisions we'll use a technique called linear probing. Here's how it works:

1. While inserting a new key-value pair if the target index for a key is occupied by another key, then we try the next index, followed by the next and so on till we the closest empty location.
2. While finding a key-value pair, we apply the same strategy, but instead of searching for an empty location, we look for a location which contains a key-value pair with the matching key.
3. While updating a key-value pair, we apply the same strategy, but instead of searching for an empty location, we look for a location which contains a key-value pair with the matching key, and update its value.

We'll define a function called `get_valid_index`, which starts searching the data list from the index determined by the hashing function `get_index` and returns the first index which is either empty or contains a key-value pair matching the given key.

QUESTION 4: Complete the function `get_valid_index` below by following the instructions in the comments.

```
def get_valid_index(data_list, key):
    # Start with the index returned by get_index
    idx = get_index(data_list, key)

    while True:
        # Get the key-value pair stored at idx
        kv = data_list[idx]

        # If it is None, return the index
        if kv is None:
            return idx

        # If the stored key matches the given key, return the index
        k, v = kv
        if k == key:
            return idx

        # Move to the next index
        idx += 1

        # Go back to the start if you have reached the end of the array
        if idx == len(data_list):
            idx = 0
```

If `get_valid_index` was defined correctly, the following cells should output `True`.

```
# Create an empty hash table
data_list2 = [None] * MAX_HASH_TABLE_SIZE

# New key 'listen' should return expected index
get_valid_index(data_list2, 'listen') == 655
```

True

```
# Insert a key-value pair for the key 'listen'
data_list2[get_index(data_list2, 'listen')] = ('listen', 99)

# Colliding key 'silent' should return next index
get_valid_index(data_list2, 'silent') == 656
```

True

(Optional) Test your implementation of `get_valid_index` on some more examples using the empty cells below.

```
get_valid_index(data_list2, 'left')
```

427

```
data_list2[get_index(data_list2, 'left')] = ('left', 111)
```

```
get_valid_index(data_list2, 'felt') == 428 # Valid index, for word with same hash
```

True

```
get_valid_index(data_list2, 'lips')
```

440

```
data_list2[get_index(data_list2, 'lips')] = ('lips', 222)
```

```
get_valid_index(data_list2, 'slip') == 441 # Valid index, for word with same hash
```

True

```
# Colliding keys were stored after one another. Get valid index works.
get_valid_index(data_list2, 'left') == 427
get_valid_index(data_list2, 'felt') == 428
get_valid_index(data_list2, 'lips') == 440
get_valid_index(data_list2, 'slip') == 441
```

True

Let's save our work before continuing.

```
jovian.commit(project=project)
```

[jovian] Updating notebook "evanmarie/python-hash-tables-assignment" on <https://jovian.ai>

[jovian] Committed successfully! <https://jovian.ai/evanmarie/python-hash-tables-assignment>

'<https://jovian.ai/evanmarie/python-hash-tables-assignment>'

Hash Table with Linear Probing

We can now implement a hash table with linear probing.

QUESTION 5: Complete the hash table (with linear probing) implementation below by following the instructions in the comments.

```
class ProbingHashTable:
    def __init__(self, max_size=MAX_HASH_TABLE_SIZE):
        # 1. Create a list of size `max_size` with all values None
        self.data_list = [None] * max_size

    def insert(self, key, value):
        # 1. Find the index for the key using get_valid_index
        idx = get_valid_index(self.data_list, key)

        # 2. Store the key-value pair at the right index
        self.data_list[idx] = (key, value)

    def find(self, key):
        # 1. Find the index for the key using get_valid_index
        idx = get_valid_index(self.data_list, key)

        # 2. Retrieve the data stored at the index
        kv = self.data_list[idx]

        # 3. Return the value if found, else return None
        return None if kv is None else kv[1]

    def update(self, key, value):
        # 1. Find the index for the key using get_valid_index
        idx = get_index(self.data_list, key)

        # 2. Store the new key-value pair at the right index
        self.data_list[idx] = (key, value)

    def list_all(self):
        # 1. Extract the key from each key-value pair
        return [kv[0] for kv in self.data_list if kv is not None]

    def print_debug_info(self, key):
        idx = get_valid_index(self.data_list, key)
        print(f'Hash Index: {idx}, Key: {key}, Value: {self.data_list[idx]}')
```

If the `ProbingHashTable` class was defined correctly, the following cells should output `True` .

```
# Create a new hash table
probing_table = ProbingHashTable()
```

```
# Insert a value
probing_table.insert('listen', 99)

# Check the value
probing_table.find('listen') == 99
```

True

```
print(probing_table.find('listen'))
```

99

```
# Insert a colliding key
probing_table.insert('silent', 200)

# Check the new and old keys
probing_table.find('listen') == 99 and probing_table.find('silent') == 200
```

True

```
# Update a key
probing_table.insert('listen', 101)

# Check the value
probing_table.find('listen') == 101
```

True

```
probing_table.list_all() == ['listen', 'silent']
```

True

(Optional) Test your implementation of `ProbingHashTable` using the empty cells below.

```
print(probing_table.find('silent'))
```

200

```
print(probing_table.find('listen'))
```

101

```
probing_table.list_all()
```

```
['listen', 'silent']
```

```
probing_table.print_debug_info('listen')
probing_table.print_debug_info('silent')
```

Hash Index: 655, Key: listen, Value: ('listen', 101)

Hash Index: 656, Key: silent, Value: ('silent', 200)

```
# More tests and checks to make sure everything is working
```

```
# Insert new colliding keys:
```

```
probing_table.insert('felt', 123)  
probing_table.insert('left', 313)  
probing_table.insert('lips', 232)  
probing_table.insert('slip', 222)  
probing_table.insert('lisp', 333)
```

```
# Check the values
```

```
probing_table.find('felt') == 123  
probing_table.find('left') == 313  
probing_table.find('lips') == 232  
probing_table.find('slip') == 222  
probing_table.find('lisp') == 333
```

True

```
# Check the colliding keys
```

```
probing_table.find('felt') == 123 and probing_table.find('left') == 313  
probing_table.find('lips') == 232 and probing_table.find('slip') == 222  
probing_table.find('lisp') == 333
```

True

```
probing_table.list_all()
```

```
['felt', 'left', 'lips', 'slip', 'lisp', 'listen', 'silent']
```

```
print(probing_table.find('felt'))  
print(probing_table.find('left'))  
print(probing_table.find('lips'))  
print(probing_table.find('lisp'))  
print(probing_table.find('slip'))  
print(probing_table.find('listen'))  
print(probing_table.find('silent'))
```

```
123  
313  
232  
333  
222  
101  
200
```

```
# Use printing function to check that all the information has made it into the table
probing_table.print_debug_info('listen')
probing_table.print_debug_info('silent')
probing_table.print_debug_info('felt')
probing_table.print_debug_info('left')
probing_table.print_debug_info('lips')
probing_table.print_debug_info('slip')
probing_table.print_debug_info('lisp')
```

Hash Index: 655, Key: listen, Value: ('listen', 101)

Hash Index: 656, Key: silent, Value: ('silent', 200)

Hash Index: 427, Key: felt, Value: ('felt', 123)

Hash Index: 428, Key: left, Value: ('left', 313)

Hash Index: 440, Key: lips, Value: ('lips', 232)

Hash Index: 441, Key: slip, Value: ('slip', 222)

Hash Index: 442, Key: lisp, Value: ('lisp', 333)

Save your work before continuing.

```
jovian.commit(project=project)
```

[jovian] Updating notebook "evanmarie/python-hash-tables-assignment" on
<https://jovian.ai>

[jovian] Committed successfully! <https://jovian.ai/evanmarie/python-hash-tables-assignment>

'<https://jovian.ai/evanmarie/python-hash-tables-assignment>'

Make a Submission

Congrats! You have now implemented hash tables from scratch. The rest of this assignment is optional.

You can make a submission on this page: <https://jovian.ai/learn/data-structures-and-algorithms-in-python/assignment/assignment-2-hash-table-and-python-dictionaries>. Submit the link to your Jovian notebook (the output of the previous cell).

You can also make a direct submission by executing the following cell:

```
jovian.submit(assignment="pythonsa-assignment2")
```

[jovian] Updating notebook "evanmarie/python-hash-tables-assignment" on
<https://jovian.ai>

[jovian] Committed successfully! <https://jovian.ai/evanmarie/python-hash-tables-assignment>

[jovian] Submitting assignment..

[jovian] Verify your submission at <https://jovian.ai/learn/data-structures-and-algorithms-in-python/assignment/assignment-2-hash-table-and-python-dictionaries>

If you are stuck, you can get help on the forum: <https://joyian.ai/forum/c/data-structures-and-algorithms-in-python/assignment-2/88>

(Optional) Python Dictionaries using Hash Tables

We can now implement Python dictionaries using hash tables. Also, Python provides a built-in function called `hash` which we can use instead of our custom hash function. It is likely to have far fewer collisions

(Optional) Question: Implement a python-friendly interface for the hash table.

```
MAX_HASH_TABLE_SIZE = 4096

class HashTable:
    def __init__(self, max_size=MAX_HASH_TABLE_SIZE):
        self.data_list = [None] * max_size

    def get_valid_index(self, key):
        # Use Python's in-built `hash` function and implement linear probing
        pass # change this

    def __getitem__(self, key):
        # Implement the logic for "find" here
        pass # change this

    def __setitem__(self, key, value):
        # Implement the logic for "insert/update" here
        pass # change this

    def __iter__(self):
        return (x for x in self.data_list if x is not None)

    def __len__(self):
        return len([x for x in self])

    def __repr__(self):
        from textwrap import indent
        pairs = [indent("{} : {}".format(repr(kv[0]), repr(kv[1])), ' ') for kv in self]
        return "{\n" + "{}".format(',\n'.join(pairs)) + "\n}"

    def __str__(self):
        return repr(self)
```

If the `HashTable` class was defined correctly, the following cells should output `True` .

```
# Create a hash table
table = HashTable()

# Insert some key-value pairs
table['a'] = 1
table['b'] = 34
```

```
# Retrieve the inserted values
table['a'] == 1 and table['b'] == 34
```

```
# Update a value
table['a'] = 99

# Check the updated value
table['a'] == 99
```

```
# Get a list of key-value pairs
list(table) == [('a', 99), ('b', 34)]
```

Since we have also implemented the `__repr__` and `__str__` functions, the output of the next cell should be:

```
{
  'a' : 99,
  'b' : 34
}
```

```
table
```

Let's save our work before continuing.

```
import jovian
```

```
jovian.commit(project=project)
```

(Optional) Hash Table Improvements

Here are some more improvements/changes you can make to your hash table implementation:

- **Track the size of the hash table** i.e. number of key-value pairs so that `len(table)` has complexity $O(1)$.
- **Implement deletion with tombstones** as described here:
<https://research.cs.vt.edu/AVresearch/hashing/deletion.php>
- **Implement dynamic resizing** to automatically grow/shrink the data list:
https://charlesreid1.com/wiki/Hash_Maps/Dynamic_Resizing
- **Implement separate chaining**, an alternative to linear probing for collision resolution:
<https://www.youtube.com/watch/T9gct6Dx-jo>

(Optional) Complexity Analysis

With choice of a good hashing function and other improvements like dynamic resizing, you can

Operation	Average-case time complexity	Worst-case time complexity
Insert/Update	$O(1)$	$O(n)$

Operation	Average-case time complexity	Worst-case time complexity
Find	$O(1)$	$O(n)$
Delete	$O(1)$	$O(n)$
List	$O(n)$	$O(n)$

Here are some questions to ponder upon?

- What is average case complexity? How does it differ from worst-case complexity?
- Do you see why insert/find/update have average-case complexity of $O(1)$ and worst-case complexity of $O(n)$?
- How is the complexity of hash tables different from binary search trees?
- When should you prefer using hash table over binary trees or vice versa?

Discuss your answers on the forum: <https://jovian.ai/forum/c/data-structures-and-algorithms-in-python/assignment-2/88>

Merge Sort, Quicksort and Divide-n-Conquer Algorithms in Python

Part 3 of "Data Structures and Algorithms in Python"

[Data Structures and Algorithms in Python](#) is a beginner-friendly introduction to common data structures (linked lists, stacks, queues, graphs) and algorithms (search, sorting, recursion, dynamic programming) in Python, designed to help you prepare for coding interviews and assessments.

Earn a verified certificate of accomplishment for this course by signing up here: <http://pythondsa.com>. Ask questions, get help & participate in discussions on the [course community forum](#).

Prerequisites

This course assumes very little background in programming and mathematics, and you can learn the required concepts here:

- Basic programming with Python ([variables](#), [data types](#), [loops](#), [functions](#) etc.)
- Some high school mathematics ([polynomials](#), [vectors](#), [matrices](#) and [probability](#))
- No prior knowledge of data structures or algorithms is required

We'll cover any additional mathematical and theoretical concepts we need as we go along.

How to Run the Code

The best way to learn the material is to execute the code and experiment with it yourself. This tutorial is an executable [Jupyter notebook](#). You can *run* this tutorial and experiment with the code examples in a couple of ways: *using free online resources* (recommended) or *on your computer*.

Option 1: Running using free online resources (1-click, recommended)

The easiest way to start executing the code is to click the **Run** button at the top of this page and select **Run on Binder**. You can also select "Run on Colab" or "Run on Kaggle", but you'll need to create an account on [Google Colab](#) or [Kaggle](#) to use these platforms.

Option 2: Running on your computer locally

To run the code on your computer locally, you'll need to set up [Python](#), download the notebook and install the required libraries. We recommend using the [Conda](#) distribution of Python. Click the **Run** button at the top of this page, select the **Run Locally** option, and follow the instructions.

Problem

In this notebook, we'll focus on solving the following problem:

QUESTION 1: You're working on a new feature on Jovian called "Top Notebooks of the Week". Write a function to sort a list of notebooks in decreasing order of likes. Keep in mind that up to millions of notebooks can be created every week, so your function needs to be as efficient as possible.

The problem of sorting a list of objects comes up over and over in computer science and software development, and it's important to understand common approaches for sorting, and the trade-offs they offer. Before we solve the above problem, we'll solve a simplified version of the problem:

QUESTION 2: Write a program to sort a list of numbers.

"Sorting" usually refers to "sorting in ascending order", unless specified otherwise.

The Method

Here's a systematic strategy we'll apply for solving problems:

1. State the problem clearly. Identify the input & output formats.
2. Come up with some example inputs & outputs. Try to cover all edge cases.
3. Come up with a correct solution for the problem. State it in plain English.
4. Implement the solution and test it using example inputs. Fix bugs, if any.
5. Analyze the algorithm's complexity and identify inefficiencies, if any.
6. Apply the right technique to overcome the inefficiency. Repeat steps 3 to 6.

1. State the problem clearly. Identify the input & output formats.

Problem

We need to write a function to sort a list of numbers in increasing order.

Input

1. `nums`: A list of numbers e.g. [4, 2, 6, 3, 4, 6, 2, 1]

Output

2. `sorted_nums`: The sorted version of `nums` e.g. [1, 2, 2, 3, 4, 4, 6, 6]

The signature of our function would be as follows:

```
def sort(nums):  
    pass
```

2. Come up with some example inputs & outputs.

Here are some scenarios we may want to test out:

1. Some lists of numbers in random order.
2. A list that's already sorted.
3. A list that's sorted in descending order.

4. A list containing repeating elements.
5. An empty list.
6. A list containing just one element.
7. A list containing one element repeated many times.
8. A really long list.

Let's create some test cases for these scenarios. We'll represent each test case as a dictionary for easier automated testing.

```
# List of numbers in random order
test0 = {
    'input': {
        'nums': [4, 2, 6, 3, 4, 6, 2, 1]
    },
    'output': [1, 2, 2, 3, 4, 4, 6, 6]
}
```

```
# List of numbers in random order
test1 = {
    'input': {
        'nums': [5, 2, 6, 1, 23, 7, -12, 12, -243, 0]
    },
    'output': [-243, -12, 0, 1, 2, 5, 6, 7, 12, 23]
}
```

```
# A list that's already sorted
test2 = {
    'input': {
        'nums': [3, 5, 6, 8, 9, 10, 99]
    },
    'output': [3, 5, 6, 8, 9, 10, 99]
}
```

```
# A list that's sorted in descending order
test3 = {
    'input': {
        'nums': [99, 10, 9, 8, 6, 5, 3]
    },
    'output': [3, 5, 6, 8, 9, 10, 99]
}
```

```
# A list containing repeating elements
test4 = {
    'input': {
        'nums': [5, -12, 2, 6, 1, 23, 7, 7, -12, 6, 12, 1, -243, 1, 0]
    },
}
```

```
'output': [-243, -12, -12, 0, 1, 1, 1, 2, 5, 6, 6, 7, 7, 12, 23]
}
```

```
# An empty list
test5 = {
  'input': {
    'nums': []
  },
  'output': []
}
```

```
# A list containing just one element
test6 = {
  'input': {
    'nums': [23]
  },
  'output': [23]
}
```

```
# A list containing one element repeated many times
test7 = {
  'input': {
    'nums': [42, 42, 42, 42, 42, 42, 42]
  },
  'output': [42, 42, 42, 42, 42, 42, 42]
}
```

To create the final test case (a really long list), we can start with a sorted list created using `range` and shuffle it to create the input.

```
import random

in_list = list(range(10000))
out_list = list(range(10000))
random.shuffle(in_list)

test8 = {
  'input': {
    'nums': in_list
  },
  'output': out_list
}
```

```
tests = [test0, test1, test2, test3, test4, test5, test6, test7, test8]
```

3. Come up with a correct solution. State it in plain English.

It's easy to come up with a correct solution. Here's one:

1. Iterate over the list of numbers, starting from the left
2. Compare each number with the number that follows it
3. If the number is greater than the one that follows it, swap the two elements
4. Repeat steps 1 to 3 till the list is sorted.

We need to repeat steps 1 to 3 at most $n-1$ times to ensure that the array is sorted. Can you explain why? Hint: After one iteration, the largest number in the list.

This method is called **bubble sort**, as it causes smaller elements to *bubble* to the top and larger to *sink* to the bottom. Here's a visual representation of the process:

6 5 3 1 8 7 2 4

4. Implement the solution and test it using example inputs.

The implementation is straightforward. We'll create a copy of the list inside our function, to avoid changing it while sorting.

```
def bubble_sort(nums):  
    # Create a copy of the list, to avoid changing it  
    nums = list(nums)  
  
    # 4. Repeat the process n-1 times  
    for _ in range(len(nums) - 1):  
  
        # 1. Iterate over the array (except last element)  
        for i in range(len(nums) - 1):  
  
            # 2. Compare the number with  
            if nums[i] > nums[i+1]:  
  
                # 3. Swap the two elements  
                nums[i], nums[i+1] = nums[i+1], nums[i]  
  
    # Return the sorted list  
    return nums
```

Notice how we're a tuple assignment to swap two elements in a single line of code.

```
x, y = 2, 3  
x, y = y, x  
x, y
```

(3, 2)

Let's test it with an example.

```
nums0, output0 = test0['input']['nums'], test0['output']

print('Input:', nums0)
print('Expected output:', output0)
result0 = bubble_sort(nums0)
print('Actual output:', result0)
print('Match:', result0 == output0)
```

```
Input: [4, 2, 6, 3, 4, 6, 2, 1]
Expected output: [1, 2, 2, 3, 4, 4, 6, 6]
Actual output: [1, 2, 2, 3, 4, 4, 6, 6]
Match: True
```

```
result0 == output0
```

True

We can evaluate all the cases together using the `evaluate_test_cases` helper function from the `jovian` library.

```
!pip install jovian --upgrade --quiet
```

```
from jovian.pythondsa import evaluate_test_cases
```

```
results = evaluate_test_cases(bubble_sort, tests)
```

TEST CASE #0

Input:

```
{'nums': [4, 2, 6, 3, 4, 6, 2, 1]}
```

Expected Output:

```
[1, 2, 2, 3, 4, 4, 6, 6]
```

Actual Output:

```
[1, 2, 2, 3, 4, 4, 6, 6]
```

Execution Time:

```
0.022 ms
```

Test Result:

PASSED

TEST CASE #1

Input:

```
{'nums': [5, 2, 6, 1, 23, 7, -12, 12, -243, 0]}
```

Expected Output:

```
[-243, -12, 0, 1, 2, 5, 6, 7, 12, 23]
```

Actual Output:

```
[-243, -12, 0, 1, 2, 5, 6, 7, 12, 23]
```

Execution Time:

0.022 ms

Test Result:

PASSED

TEST CASE #2

Input:

```
{'nums': [3, 5, 6, 8, 9, 10, 99]}
```

Expected Output:

```
[3, 5, 6, 8, 9, 10, 99]
```

Actual Output:

```
[3, 5, 6, 8, 9, 10, 99]
```

Execution Time:

0.013 ms

Test Result:

PASSED

TEST CASE #3

Input:

```
{'nums': [99, 10, 9, 8, 6, 5, 3]}
```

Expected Output:

```
[3, 5, 6, 8, 9, 10, 99]
```

Actual Output:

```
[3, 5, 6, 8, 9, 10, 99]
```

Execution Time:

0.014 ms

Test Result:

PASSED

TEST CASE #4

Input:

```
{'nums': [5, -12, 2, 6, 1, 23, 7, 7, -12, 6, 12, 1, -243, 1, 0]}
```

Expected Output:

```
[-243, -12, -12, 0, 1, 1, 1, 2, 5, 6, 6, 7, 7, 12, 23]
```

Actual Output:

```
[-243, -12, -12, 0, 1, 1, 1, 2, 5, 6, 6, 7, 7, 12, 23]
```

Execution Time:

0.044 ms

Test Result:

PASSED

TEST CASE #5

Input:

```
{'nums': []}
```

Expected Output:

```
[]
```

Actual Output:

```
[]
```

Execution Time:

0.003 ms

Test Result:

PASSED

TEST CASE #6

Input:

```
{'nums': [23]}
```

Expected Output:

```
[23]
```

Actual Output:

```
[23]
```

Execution Time:

0.004 ms

Test Result:

PASSED

TEST CASE #7

Input:

```
{'nums': [42, 42, 42, 42, 42, 42, 42]}
```

Expected Output:

```
[42, 42, 42, 42, 42, 42, 42]
```

Actual Output:

```
[42, 42, 42, 42, 42, 42, 42]
```

Execution Time:

0.011 ms

Test Result:

PASSED

TEST CASE #8

Input:

```
{'nums': [1115, 504, 1825, 4829, 2383, 7123, 2607, 6877, 5968, 6385, 3705, 352, 1855, 8455, 1261, 88...
```

Expected Output:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 2...
```

Actual Output:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 2...
```

Execution Time:

11269.663 ms

Test Result:

PASSED

SUMMARY

TOTAL: 9, PASSED: 9, FAILED: 0

Great, looks like all our test cases passed! Although the last test case (a list of 10,000 numbers) took about 12 seconds to execute.

5. Analyze the algorithm's complexity and identify inefficiencies

The core operations in bubble sort are "compare" and "swap". To analyze the time complexity, we can simply count the total number of comparisons being made, since the total number of swaps will be less than or equal to the total number of comparisons (can you see why?).

```
for _ in range(len(nums) - 1):  
    for i in range(len(nums) - 1):
```

```
if nums[i] > nums[i+1]:
    nums[i], nums[i+1] = nums[i+1], nums[i]
```

There are two loops, each of length $n-1$, where n is the number of elements in `nums`. So the total number of comparisons is $(n-1) * (n-1)$ i.e. $(n-1)^2$ i.e. $n^2 - 2n + 1$.

Expressing this in the Big O notation, we can conclude that the time complexity of bubble sort is $O(n^2)$ (also known as quadratic complexity).

Exercise: Verify that the bubble sort requires $O(1)$ additional space.

The space complexity of bubble sort is $O(n)$, even though it requires only constant/zero additional space, because the space required to store the inputs is also considered while calculating space complexity.

As we saw from the last test, a list of 10,000 numbers takes about 12 seconds to be sorted using bubble sort. A list of ten times the size will 100 times longer i.e. about 20 minutes to be sorted, which is quite inefficient. A list of a million elements would take close to 2 days to be sorted.

The inefficiency in bubble sort comes from the fact that we're shifting elements by at most one position at a time.

6 5 3 1 8 7 2 4

Insertion Sort

Before we look at explore more efficient sorting techniques, here's another simple sorting technique called insertion sort, where we keep the initial portion of the array sorted and insert the remaining elements one by one at the right position.

```
def insertion_sort(nums):
    nums = list(nums)
    for i in range(len(nums)):
        cur = nums.pop(i)
        j = i-1
        while j >=0 and nums[j] > cur:
            j -= 1
        nums.insert(j+1, cur)
    return nums
```

```
nums0, output0 = test0['input']['nums'], test0['output']
```

```
print('Input:', nums0)
print('Expected output:', output0)
result0 = insertion_sort(nums0)
```

```
print('Actual output:', result0)
print('Match:', result0 == output0)
```

Input: [4, 2, 6, 3, 4, 6, 2, 1]

Expected output: [1, 2, 2, 3, 4, 4, 6, 6]

Actual output: [1, 2, 2, 3, 4, 4, 6, 6]

Match: True

Exercises:

1. Read the source code of the `insertion_sort` function and describe the algorithm in plain English. Reading source code is an essential skill for software development.
2. Determine the time and space complexity of insertion sort. Is it any better than bubble sort? Why or why not?

Save and upload your work to Jovian

Whether you're running this Jupyter notebook online or on your computer, it's essential to save your work from time to time. You can continue working on a saved notebook later or share it with friends and colleagues to let them execute your code. [Jovian](#) offers an easy way of saving and sharing your Jupyter notebooks online.

```
!pip install jovian --upgrade --quiet
```

```
import jovian
```

```
jovian.commit()
```

```
[jovian] Updating notebook "evanmarie/python-sorting-divide-and-conquer" on
https://jovian.ai
```

```
[jovian] Committed successfully! https://jovian.ai/evanmarie/python-sorting-divide-and-conquer
```

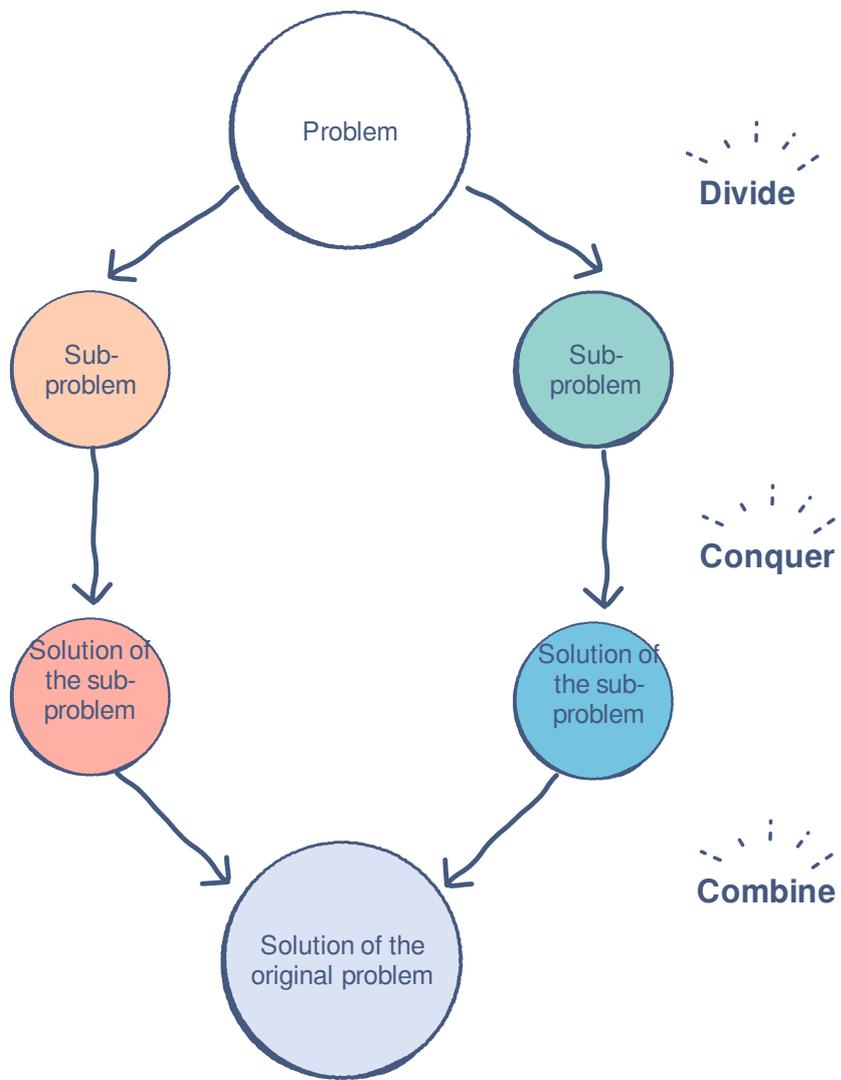
```
'https://jovian.ai/evanmarie/python-sorting-divide-and-conquer'
```

6. Apply the right technique to overcome the inefficiency. Repeat Steps 3 to 6.

To performing sorting more efficiently, we'll apply a strategy called **Divide and Conquer**, which has the following general steps:

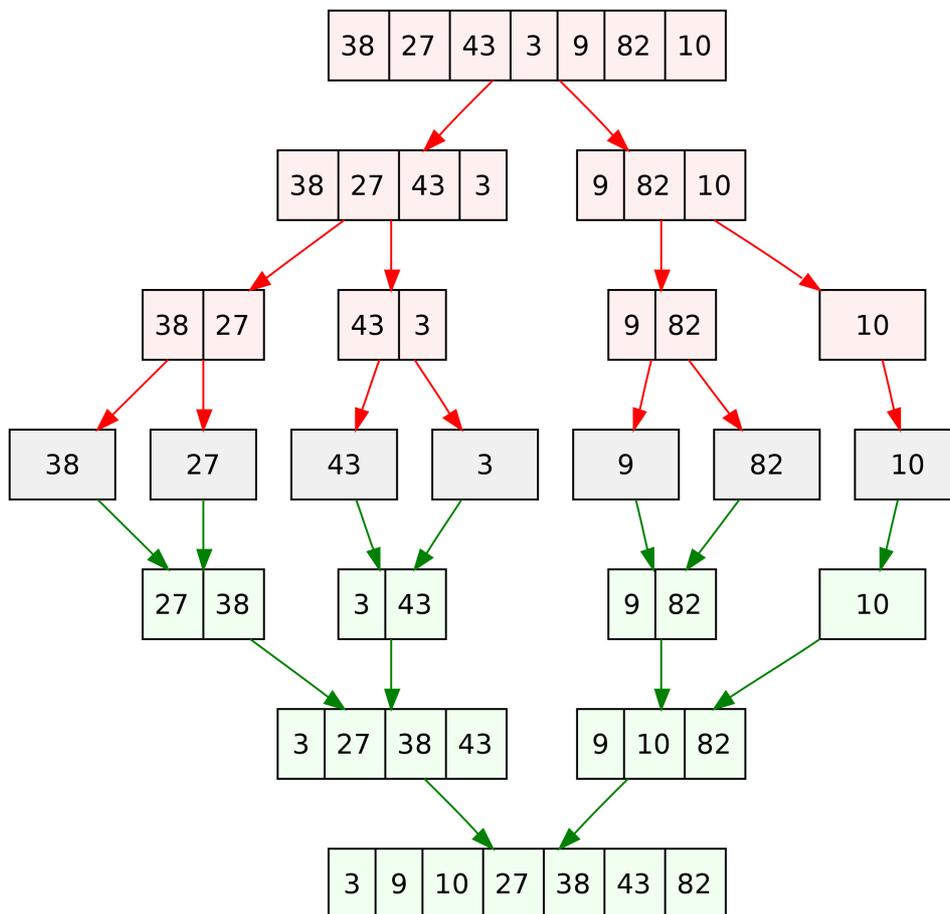
1. Divide the inputs into two roughly equal parts.
2. Recursively solve the problem individually for each of the two parts.
3. Combine the results to solve the problem for the original inputs.
4. Include terminating conditions for small or indivisible inputs.

Here's a visual representation of the strategy:



Merge Sort

Following a visual representation of the divide and conquer applied for sorting numbers. This algorithm is known as merge sort:



7. Come up with a correct solution. State it in Plain English.

Here's a step-by-step description for merge sort:

1. If the input list is empty or contains just one element, it is already sorted. Return it.
2. If not, divide the list of numbers into two roughly equal parts.
3. Sort each part recursively using the merge sort algorithm. You'll get back two sorted lists.
4. Merge the two sorted lists to get a single sorted list

Can you guess how the "merge" operation step 4 works? Hint: Watch this animation:

<https://youtu.be/GW0USDwhBgo?t=28>

QUESTION 3: Write a function to merge two sorted arrays.

Try to explain how the merge operation works in your own words below:

1. If the portion of the list you are currently considering is empty or only contains one element, consider it sorted.
2. Take the entire list of numbers, and divide it in half until it the parts are too small to be further divided
3. Take the smallest sublist of the list, and compare and sort its contents, do so for each sublist.
4. After all sublists have been sorted, start with the smallest sublists, and begin recombining them by comparing their first elements and adding them to the new sorted list in sorted order, then the second elements, and so on. Do this until the sorting has been merged back together.

8. Implement the solution and test it using example inputs

Let's implement the merge sort algorithm assuming we already have a helper function called `merge` for merging two sorted arrays.

```
def merge_sort(nums):
    # Terminating condition (list of 0 or 1 elements)
    if len(nums) <= 1:
        return nums

    # Get the midpoint
    mid = len(nums) // 2

    # Split the list into two halves
    left = nums[:mid]
    right = nums[mid:]

    # Solve the problem for each half recursively
    left_sorted, right_sorted = merge_sort(left), merge_sort(right)

    # Combine the results of the two halves
    sorted_nums = merge(left_sorted, right_sorted)

    return sorted_nums
```

To merge two sorted arrays, we can repeatedly compare the two least elements of each array, and copy over the smaller one into a new array.

Here's a visual representation of the merge operation:

[1, 4, 7]	[0, 2, 3]	[?, ?, ?, ?, ?, ?]
[1, 4, 7]	[0, 2, 3]	[0, ?, ?, ?, ?, ?]
[1, 4, 7]	[0, 2, 3]	[0, 1, ?, ?, ?, ?]
[1, 4, 7]	[0, 2, 3]	[0, 1, 2, ?, ?, ?]
[1, 4, 7]	[0, 2, 3]	[0, 1, 2, 3, ?, ?]
[1, 4, 7]	[0, 2, 3]	[0, 1, 2, 3, 4, ?]
[1, 4, 7]	[0, 2, 3]	[0, 1, 2, 3, 4, 7]
[1, 4, 7]	[0, 2, 3]	[0, 1, 2, 3, 4, 7]

```
def merge(nums1, nums2):
    # List to store the results
    merged = []
```

```

# Indices for iteration
i, j = 0, 0

# Loop over the two lists
while i < len(nums1) and j < len(nums2):

    # Include the smaller element in the result and move to next element
    if nums1[i] <= nums2[j]:
        merged.append(nums1[i])
        i += 1
    else:
        merged.append(nums2[j])
        j += 1

# Get the remaining parts
nums1_tail = nums1[i:]
nums2_tail = nums2[j:]

# Return the final merged array
return merged + nums1_tail + nums2_tail

```

Let's test the merge operation, before we test merge sort.

```
merge([1, 4, 7, 9, 11], [-1, 0, 2, 3, 8, 12])
```

```
[-1, 0, 1, 2, 3, 4, 7, 8, 9, 11, 12]
```

It seems to work as expected. We can now test the `merge_sort` function.

```

nums0, output0 = test0['input']['nums'], test0['output']

print('Input:', nums0)
print('Expected output:', output0)
result0 = merge_sort(nums0)
print('Actual output:', result0)
print('Match:', result0 == output0)

```

```
Input: [4, 2, 6, 3, 4, 6, 2, 1]
```

```
Expected output: [1, 2, 2, 3, 4, 4, 6, 6]
```

```
Actual output: [1, 2, 2, 3, 4, 4, 6, 6]
```

```
Match: True
```

Let's test all the cases using the `evaluate_test_cases` function from `jovian`.

```
results = evaluate_test_cases(merge_sort, tests)
```

TEST CASE #0

Input:

```
{'nums': [4, 2, 6, 3, 4, 6, 2, 1]}
```

Expected Output:

```
[1, 2, 2, 3, 4, 4, 6, 6]
```

Actual Output:

```
[1, 2, 2, 3, 4, 4, 6, 6]
```

Execution Time:

```
0.026 ms
```

Test Result:

```
PASSED
```

TEST CASE #1

Input:

```
{'nums': [5, 2, 6, 1, 23, 7, -12, 12, -243, 0]}
```

Expected Output:

```
[-243, -12, 0, 1, 2, 5, 6, 7, 12, 23]
```

Actual Output:

```
[-243, -12, 0, 1, 2, 5, 6, 7, 12, 23]
```

Execution Time:

```
0.027 ms
```

Test Result:

```
PASSED
```

TEST CASE #2

Input:

```
{'nums': [3, 5, 6, 8, 9, 10, 99]}
```

Expected Output:

```
[3, 5, 6, 8, 9, 10, 99]
```

Actual Output:

[3, 5, 6, 8, 9, 10, 99]

Execution Time:

0.015 ms

Test Result:

PASSED

TEST CASE #3

Input:

{'nums': [99, 10, 9, 8, 6, 5, 3]}

Expected Output:

[3, 5, 6, 8, 9, 10, 99]

Actual Output:

[3, 5, 6, 8, 9, 10, 99]

Execution Time:

0.019 ms

Test Result:

PASSED

TEST CASE #4

Input:

{'nums': [5, -12, 2, 6, 1, 23, 7, 7, -12, 6, 12, 1, -243, 1, 0]}

Expected Output:

[-243, -12, -12, 0, 1, 1, 1, 2, 5, 6, 6, 7, 7, 12, 23]

Actual Output:

[-243, -12, -12, 0, 1, 1, 1, 2, 5, 6, 6, 7, 7, 12, 23]

Execution Time:

0.031 ms

Test Result:

PASSED

TEST CASE #5

Input:

```
{'nums': []}
```

Expected Output:

```
[]
```

Actual Output:

```
[]
```

Execution Time:

0.002 ms

Test Result:

PASSED

TEST CASE #6

Input:

```
{'nums': [23]}
```

Expected Output:

```
[23]
```

Actual Output:

```
[23]
```

Execution Time:

0.002 ms

Test Result:

PASSED

TEST CASE #7

Input:

```
{'nums': [42, 42, 42, 42, 42, 42, 42]}
```

Expected Output:

```
[42, 42, 42, 42, 42, 42, 42]
```

Actual Output:

```
[42, 42, 42, 42, 42, 42, 42]
```

Execution Time:

0.017 ms

Test Result:

PASSED

TEST CASE #8

Input:

```
{'nums': [1115, 504, 1825, 4829, 2383, 7123, 2607, 6877, 5968, 6385, 3705, 352, 1855, 8455, 1261, 88...]}
```

Expected Output:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 2...]
```

Actual Output:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 2...]
```

Execution Time:

37.191 ms

Test Result:

PASSED

SUMMARY

TOTAL: 9, PASSED: 9, FAILED: 0

All the test cases have passed! Our function works as expected.

Notice the last test case, the merge sort function took just 50 milliseconds to sort 10,000 numbers, which took bubble sort about 12 seconds.

```
jovian.commit()
```

```
[jovian] Updating notebook "evanmarie/python-sorting-divide-and-conquer" on  
https://jovian.ai
```

```
[jovian] Committed successfully! https://jovian.ai/evanmarie/python-sorting-divide-and-conquer
```

```
'https://jovian.ai/evanmarie/python-sorting-divide-and-conquer'
```

9. Analyze the algorithm's complexity and identify inefficiencies

Analyzing the complexity of recursive algorithms can be tricky. It helps to track and follow the chain of recursive calls. We'll add some print statements to our `merge_sort` and `merge` functions to display the *tree* of recursive function calls.

```
def merge(nums1, nums2, depth=0):  
    print(' '*depth, 'merge:', nums1, nums2)  
    i, j, merged = 0, 0, []  
    while i < len(nums1) and j < len(nums2):  
        if nums1[i] <= nums2[j]:  
            merged.append(nums1[i])  
            i += 1  
        else:  
            merged.append(nums2[j])  
            j += 1  
    return merged + nums1[i:] + nums2[j:]  
  
def merge_sort(nums, depth=0):  
    print(' '*depth, 'merge_sort:', nums)  
    if len(nums) < 2:  
        return nums  
    mid = len(nums) // 2  
    return merge(merge_sort(nums[:mid], depth+1),  
                merge_sort(nums[mid:], depth+1),  
                depth+1)
```

```
merge_sort([5, -12, 2, 6, 1, 23, 7, 7, -12])
```

```
merge_sort: [5, -12, 2, 6, 1, 23, 7, 7, -12]  
merge_sort: [5, -12, 2, 6]  
merge_sort: [5, -12]
```

```

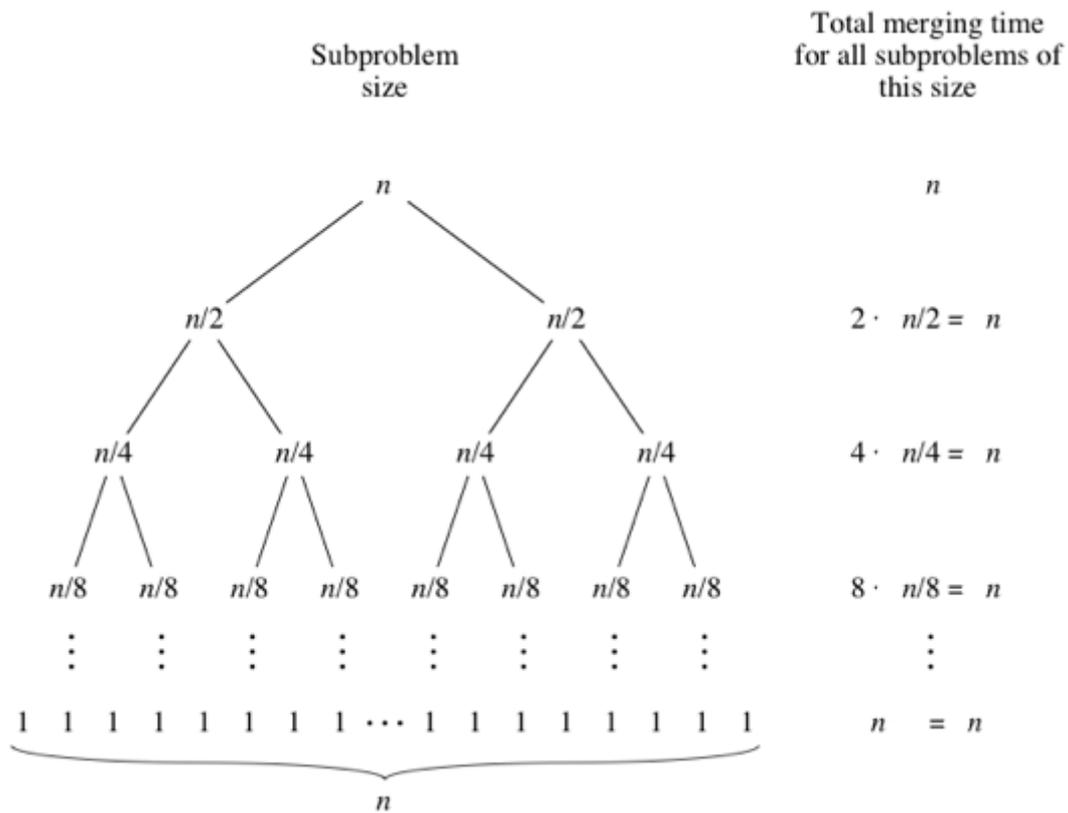
merge_sort: [5]
merge_sort: [-12]
merge: [5] [-12]
merge_sort: [2, 6]
merge_sort: [2]
merge_sort: [6]
merge: [2] [6]
merge: [-12, 5] [2, 6]
merge_sort: [1, 23, 7, 7, -12]
merge_sort: [1, 23]
merge_sort: [1]
merge_sort: [23]
merge: [1] [23]
merge_sort: [7, 7, -12]
merge_sort: [7]
merge_sort: [7, -12]
merge_sort: [7]
merge_sort: [-12]
merge: [7] [-12]
merge: [7] [-12, 7]
merge: [1, 23] [-12, 7, 7]
merge: [-12, 2, 5, 6] [-12, 1, 7, 7, 23]
[-12, -12, 1, 2, 5, 6, 7, 7, 23]

```

We can now see that each `merge_sort` call itself invokes `merge_sort` twice (but with an array half the size), and also invokes the `merge` function once to merge the two resulting arrays. The two calls to `merge_sort` themselves make two recursive calls followed by an invocation of `merge`. The division continues till we reach an list of size 1 or 0. Thus, the merge sort algorithm ultimately boils down to a series of `merge` operations performed on arrays of varying sizes. Inside the `merge` function we perform comparisons and add numbers to a new array.

Exercise: Verify that time complexity of the merge operation is $O(n)$, where n is the sum of the sizes of the two input lists. Hint: Count the number of comparisons.

To find the overall complexity of `merge_sort`, we simply need to count how many times the `merge` function was invoked and the size of the input list for each invocation. Here's how all the subproblems can be visualized:



Counting from the top and starting from 0, the k^{th} level of the above tree involves 2^k invocations of `merge` with sublists of size roughly $n/2^k$, where n is the size of the original input list. Therefore the total number of comparisons at each level of the tree is $2^k * n/2^k = n$.

Thus, if the height of the tree is h , the total number of comparisons is $n * h$. Since there are n sublists of size 1 at the lowest level, it follows that $2^{(h-1)} = n$ i.e. $h = \log n + 1$. Thus the time complexity of the merge sort algorithms is $O(n \log n)$.

As we already saw, it took just 50 ms to sort an array of 10,000 elements. Even an array of 1 million elements will take only a few seconds to be sorted.

Space Complexity

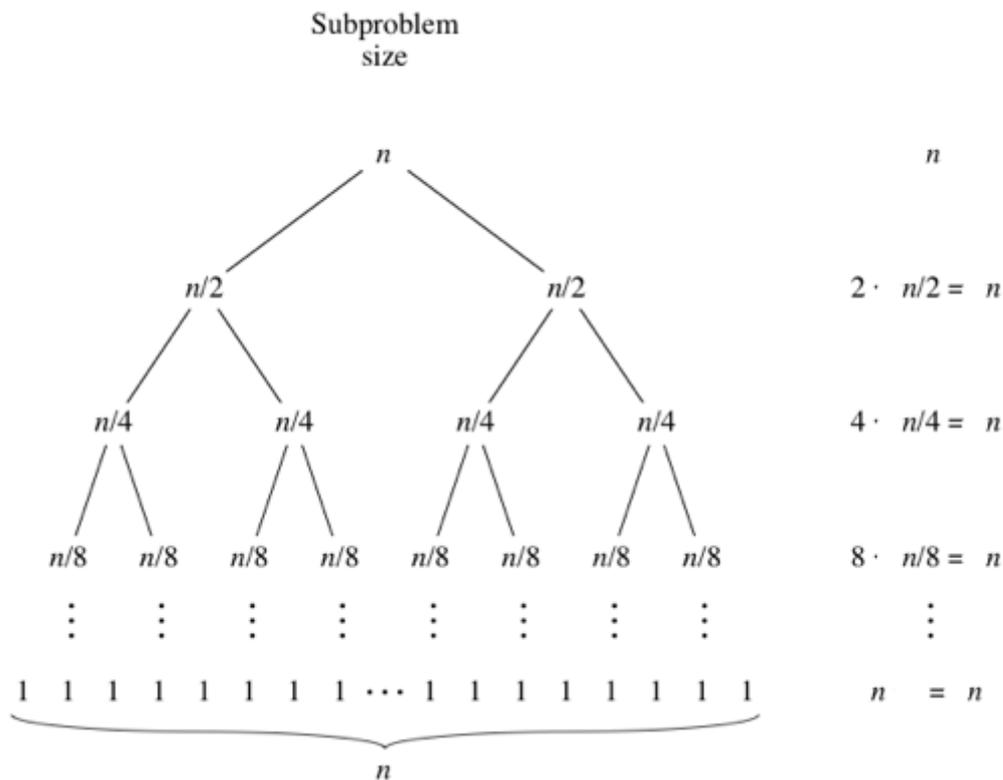
To find the space complexity of merge sort, it helps to recall that a new list with equal to the sum of the sizes of the two lists is created in each invocation of `merge`.

```

i, j, merged = 0, 0, []
while i < len(nums1) and j < len(nums2):
    if nums1[i] <= nums2[j]:
        merged.append(nums1[i])
        i += 1
    else:
        merged.append(nums2[j])
        j += 1

```

At first glance, it may seem that $O(n)$ space is required for each level of the tree, so the space complexity of merge sort is $O(n \log n)$.



However, since the original sublists can be discarded after the merge operation, the additional space can be freed or reused for future merge calls. Thus, merge sort requires $O(n)$ additional space i.e. the space complexity is $O(n)$.

There are several extensions and variations and extensions of merge sort:

- K-way merge sort: https://en.wikipedia.org/wiki/K-way_merge_algorithm
- Counting inversions in an array: <https://www.geeksforgeeks.org/counting-inversions/>
- Merge sort and insertion sort hybrids: <https://en.wikipedia.org/wiki/Timsort>

```
jovian.commit()
```

[jovian] Updating notebook "evanmarie/python-sorting-divide-and-conquer" on <https://jovian.ai>

[jovian] Committed successfully! <https://jovian.ai/evanmarie/python-sorting-divide-and-conquer>

'<https://jovian.ai/evanmarie/python-sorting-divide-and-conquer>'

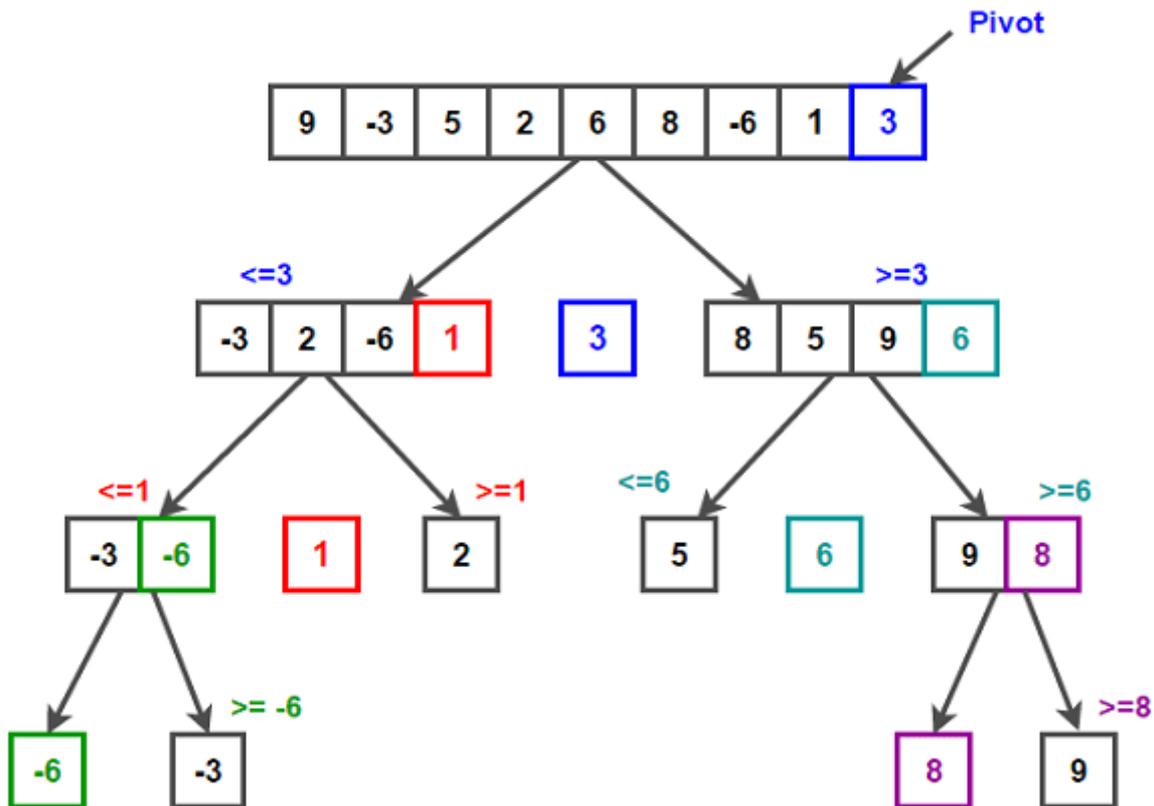
10. Apply the right technique to overcome the inefficiency. Repeat Steps 3 to 6.

The fact that merge sort requires allocating additional space as large as the input itself makes it somewhat slow in practice because memory allocation is far more expensive than comparisons or swapping.

Quicksort

To overcome the space inefficiencies of merge sort, we'll study another divide-and-conquer based sorting algorithm called **quicksort**, which works as follows:

1. If the list is empty or has just one element, return it. It's already sorted.
2. Pick a random element from the list. This element is called a *pivot*.
3. Reorder the list so that all elements with values less than or equal to the pivot come before the pivot, while all elements with values greater than the pivot come after it. This operation is called *partitioning*.
4. The pivot element divides the array into two parts which can be sorted independently by making a recursive call to quicksort.



The key observation here is that after the partition, the pivot element is at its right place in the sorted array, and the two parts of the array can be sorted independently in-place.

Here's an implementation of quicksort, assuming we already have a helper function called `partition` which picks a pivot, partitions the array in-place, and returns the position of the pivot element.

```
def quicksort(nums, start=0, end=None):
    # print('quicksort', nums, start, end)
    if end is None:
        nums = list(nums)
        end = len(nums) - 1

    if start < end:
        pivot = partition(nums, start, end)
        quicksort(nums, start, pivot-1)
        quicksort(nums, pivot+1, end)

    return nums
```

Here's how the partition operation works([source](#)):

[1, 5, 6, 2, 0, 11, 3]	advance left
[1, 5, 6, 2, 0, 11, 3]	advance right
[1, 5, 6, 2, 0, 11, 3]	swap
[1, 0, 6, 2, 5, 11, 3]	advance left
[1, 0, 6, 2, 5, 11, 3]	advance right
[1, 0, 6, 2, 5, 11, 3]	swap
[1, 0, 2, 6, 5, 11, 3]	advance left
[1, 0, 2, 6, 5, 11, 3]	swap pivot
[1, 0, 2, 3, 5, 11, 6]	result

Here's an implementation of partition, which uses the last element of the list as a pivot:

```
def partition(nums, start=0, end=None):
    # print('partition', nums, start, end)
    if end is None:
        end = len(nums) - 1

    # Initialize right and left pointers
    l, r = start, end-1

    # Iterate while they're apart
    while r > l:
        # print(' ', nums, l, r)
        # Increment left pointer if number is less or equal to pivot
        if nums[l] <= nums[end]:
            l += 1

        # Decrement right pointer if number is greater than pivot
        elif nums[r] > nums[end]:
            r -= 1

        # Two out-of-place elements found, swap them
        else:
            nums[l], nums[r] = nums[r], nums[l]
    # print(' ', nums, l, r)
```

```
# Place the pivot between the two parts
if nums[l] > nums[end]:
    nums[l], nums[end] = nums[end], nums[l]
    return l
else:
    return end
```

Let's see the partition function in action:

```
l1 = [1, 5, 6, 2, 0, 11, 3]
pivot = partition(l1)
print(l1, pivot)
```

```
[1, 0, 2, 3, 5, 11, 6] 3
```

As expected the list was partitioned using 3 as the pivot element, which finally ends up at the index 2 within the partitioned list.

Exercise: Add print statements inside the partition function to display the list, the left pointer and the right pointer at the beginning and end of every loop, to study how the partitioning works.

We can now see quicksort in action:

```
nums0, output0 = test0['input']['nums'], test0['output']

print('Input:', nums0)
print('Expected output:', output0)
result0 = quicksort(nums0)
print('Actual output:', result0)
print('Match:', result0 == output0)
```

```
Input: [4, 2, 6, 3, 4, 6, 2, 1]
```

```
Expected output: [1, 2, 2, 3, 4, 4, 6, 6]
```

```
Actual output: [1, 2, 2, 3, 4, 4, 6, 6]
```

```
Match: True
```

Let's test all the cases using the `evaluate_test_cases` function from `jovian`.

```
from jovian.pythondsa import evaluate_test_cases

results = evaluate_test_cases(quicksort, tests)
```

TEST CASE #0

Input:

```
{'nums': [4, 2, 6, 3, 4, 6, 2, 1]}
```

Expected Output:

[1, 2, 2, 3, 4, 4, 6, 6]

Actual Output:

[1, 2, 2, 3, 4, 4, 6, 6]

Execution Time:

0.018 ms

Test Result:

PASSED

TEST CASE #1

Input:

{'nums': [5, 2, 6, 1, 23, 7, -12, 12, -243, 0]}

Expected Output:

[-243, -12, 0, 1, 2, 5, 6, 7, 12, 23]

Actual Output:

[-243, -12, 0, 1, 2, 5, 6, 7, 12, 23]

Execution Time:

0.016 ms

Test Result:

PASSED

TEST CASE #2

Input:

{'nums': [3, 5, 6, 8, 9, 10, 99]}

Expected Output:

[3, 5, 6, 8, 9, 10, 99]

Actual Output:

[3, 5, 6, 8, 9, 10, 99]

Execution Time:

0.011 ms

Test Result:

PASSED

TEST CASE #3

Input:

{'nums': [99, 10, 9, 8, 6, 5, 3]}

Expected Output:

[3, 5, 6, 8, 9, 10, 99]

Actual Output:

[3, 5, 6, 8, 9, 10, 99]

Execution Time:

0.008 ms

Test Result:

PASSED

TEST CASE #4

Input:

{'nums': [5, -12, 2, 6, 1, 23, 7, 7, -12, 6, 12, 1, -243, 1, 0]}

Expected Output:

[-243, -12, -12, 0, 1, 1, 1, 2, 5, 6, 6, 7, 7, 12, 23]

Actual Output:

[-243, -12, -12, 0, 1, 1, 1, 2, 5, 6, 6, 7, 7, 12, 23]

Execution Time:

0.021 ms

Test Result:

PASSED

TEST CASE #5

Input:

```
{'nums': []}
```

Expected Output:

```
[]
```

Actual Output:

```
[]
```

Execution Time:

0.003 ms

Test Result:

PASSED

TEST CASE #6

Input:

```
{'nums': [23]}
```

Expected Output:

```
[23]
```

Actual Output:

```
[23]
```

Execution Time:

0.002 ms

Test Result:

PASSED

TEST CASE #7

Input:

```
{'nums': [42, 42, 42, 42, 42, 42, 42]}
```

Expected Output:

```
[42, 42, 42, 42, 42, 42, 42]
```

Actual Output:

```
[42, 42, 42, 42, 42, 42, 42]
```

Execution Time:

```
0.011 ms
```

Test Result:

PASSED

TEST CASE #8

Input:

```
{'nums': [1115, 504, 1825, 4829, 2383, 7123, 2607, 6877, 5968, 6385, 3705, 352, 1855, 8455, 1261, 88...]}
```

Expected Output:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 2...]
```

Actual Output:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 2...]
```

Execution Time:

```
24.157 ms
```

Test Result:

PASSED

SUMMARY

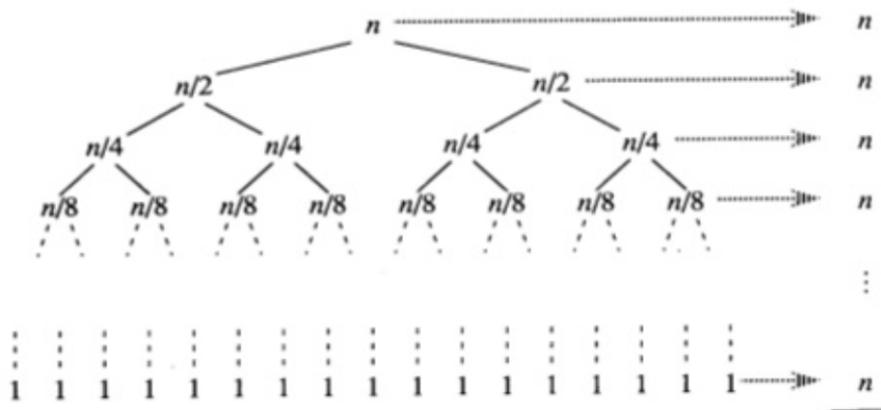
TOTAL: 9, PASSED: 9, FAILED: 0

All the test cases have passed successfully! You will also notice that is also marginally faster than merge sort for larger lists.

Time Complexity of Quicksort

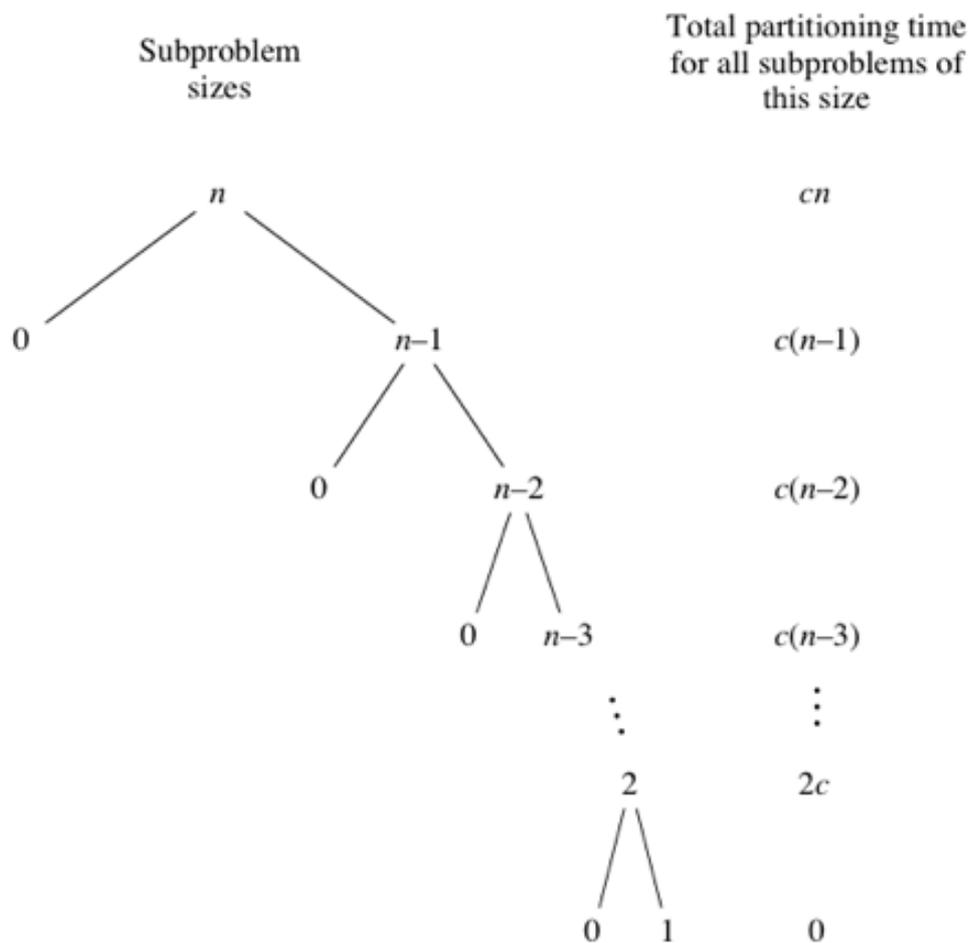
If we assume that

Best case partitioning:



If we partition the list into two nearly equal parts, then the complexity analysis is similar to that of merge sort and quicksort has the complexity $O(n \log n)$. This is called the average-case complexity.

Worst case partitioning:



In this case, the partition is called n times with lists of sizes $n, n-1, \dots$ so that total comparisons are $n + (n-1) + (n-2) + \dots + 2 + 1 = n * (n-1) / 2$. So the worst-case complexity of quicksort is $O(n^2)$.

Exercise: Verify that quicksort requires $O(1)$ additional space.

Despite the quadratic worst case time complexity Quicksort is preferred in many situations because its running time is closer to $O(n \log n)$ in practice, especially with a good strategy for picking a pivot. Some these are:

- [Picking a random pivot](#)
- [Picking median of medians](#)

Here are some other problems you can solve using the partitioning strategy of Quicksort:

<https://www.techiedelight.com/problems-solved-using-partitioning-logic-quicksort/>

```
jovian.commit()
```

```
[jovian] Updating notebook "evanmarie/python-sorting-divide-and-conquer" on
https://jovian.ai
```

```
[jovian] Committed successfully! https://jovian.ai/evanmarie/python-sorting-divide-and-conquer
```

```
'https://jovian.ai/evanmarie/python-sorting-divide-and-conquer'
```

Custom Comparison Functions

Let's return to our original problem statement now.

QUESTION 1: You're working on a new feature on Jovian called "Top Notebooks of the Week". Write a function to sort a list of notebooks in decreasing order of likes. Keep in mind that up to millions of notebooks can be created every week, so your function needs to be as efficient as possible.

First, we need to sort objects, not just numbers. Also, we want to sort them in the descending order of likes. To achieve this, all we need is a custom comparison function to compare two notebooks.

Let's create a class that captures basic information about notebooks.

```
class Notebook:
    def __init__(self, title, username, likes):
        self.title, self.username, self.likes = title, username, likes

    def __repr__(self):
        return 'Notebook <"{}/{}", {} likes>'.format(self.username, self.title, self.likes)
```

Let's create some sample notebooks

```
nb0 = Notebook('pytorch-basics', 'aakashns', 373)
nb1 = Notebook('linear-regression', 'siddhant', 532)
nb2 = Notebook('logistic-regression', 'vikas', 31)
nb3 = Notebook('feedforward-nn', 'sonaksh', 94)
nb4 = Notebook('cifar10-cnn', 'biraj', 2)
nb5 = Notebook('cifar10-resnet', 'tanya', 29)
nb6 = Notebook('anime-gans', 'hemanth', 80)
nb7 = Notebook('python-fundamentals', 'vishal', 136)
nb8 = Notebook('python-functions', 'aakashns', 74)
nb9 = Notebook('python-numpy', 'siddhant', 92)
```

```
notebooks = [nb0, nb1, nb2, nb3, nb4, nb5, nb6, nb7, nb8, nb9]
```

```
notebooks
```

```
[Notebook <"aakashns/pytorch-basics", 373 likes>,
Notebook <"siddhant/linear-regression", 532 likes>,
Notebook <"vikas/logistic-regression", 31 likes>,
Notebook <"sonaksh/feedforward-nn", 94 likes>,
Notebook <"biraj/cifar10-cnn", 2 likes>,
Notebook <"tanya/cifar10-resnet", 29 likes>,
Notebook <"hemanth/anime-gans", 80 likes>,
Notebook <"vishal/python-fundamentals", 136 likes>,
Notebook <"aakashns/python-functions", 74 likes>,
Notebook <"siddhant/python-numpy", 92 likes>]
```

Next, we'll define a custom comparison function for comparing two notebooks. It will return the strings 'lesser', 'equal' or 'greater' to establish order between the two objects.

```
def compare_likes(nb1, nb2):
    if nb1.likes > nb2.likes:
        return 'lesser'
    elif nb1.likes == nb2.likes:
        return 'equal'
    elif nb1.likes < nb2.likes:
        return 'greater'
```

Note that we say nb1 is *lesser* than nb2 if it has higher likes, because we want to sort the notebooks in decreasing order of likes.

Here's an implementation of merge sort which accepts a custom comparison function.

```
def default_compare(x, y):
    if x < y:
        return 'less'
    elif x == y:
        return 'equal'
    else:
        return 'greater'

def merge_sort(objs, compare=default_compare):
    if len(objs) < 2:
        return objs
    mid = len(objs) // 2
    return merge(merge_sort(objs[:mid], compare),
                merge_sort(objs[mid:], compare),
                compare)

def merge(left, right, compare):
    i, j, merged = 0, 0, []
    while i < len(left) and j < len(right):
        result = compare(left[i], right[j])
        if result == 'lesser' or result == 'equal':
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            j += 1
    return merged + left[i:] + right[j:]
```

```
sorted_notebooks = merge_sort(notebooks, compare_likes)
```

```
sorted_notebooks
```

```
[Notebook <"siddhant/linear-regression", 532 likes>,
Notebook <"aakashns/pytorch-basics", 373 likes>,
Notebook <"vishal/python-fundamentals", 136 likes>,
Notebook <"sonaksh/feedforward-nn", 94 likes>,
```

```
Notebook <"siddhant/python-numpy", 92 likes>,
Notebook <"hemanth/anime-gans", 80 likes>,
Notebook <"aakashns/python-functions", 74 likes>,
Notebook <"vikas/logistic-regression", 31 likes>,
Notebook <"tanya/cifar10-resnet", 29 likes>,
Notebook <"biraj/cifar10-cnn", 2 likes>]
```

As you can see, the notebooks are now sorted by likes. Since we have written a generic `merge_sort` function that works with any compare function, we can also use it to sort the notebooks by title.

```
def compare_titles(nb1, nb2):
    if nb1.title < nb2.title:
        return 'lesser'
    elif nb1.title == nb2.title:
        return 'equal'
    elif nb1.title > nb2.title:
        return 'greater'
```

```
merge_sort(notebooks, compare_titles)
```

```
[Notebook <"hemanth/anime-gans", 80 likes>,
Notebook <"biraj/cifar10-cnn", 2 likes>,
Notebook <"tanya/cifar10-resnet", 29 likes>,
Notebook <"sonaksh/feedforward-nn", 94 likes>,
Notebook <"siddhant/linear-regression", 532 likes>,
Notebook <"vikas/logistic-regression", 31 likes>,
Notebook <"aakashns/python-functions", 74 likes>,
Notebook <"vishal/python-fundamentals", 136 likes>,
Notebook <"siddhant/python-numpy", 92 likes>,
Notebook <"aakashns/pytorch-basics", 373 likes>]
```

Exercise: Implement and test generic versions of bubble sort, insertion sort and quicksort using the empty cells below.

```
def compare_usernames(nb1, nb2):
    if nb1.username < nb2.username:
        return 'lesser'
    elif nb1.username == nb2.username:
        return 'equal'
    elif nb1.username > nb2.username:
        return 'greater'
```

```
merge_sort(notebooks, compare_usernames)
```

```
[Notebook <"aakashns/pytorch-basics", 373 likes>,
Notebook <"aakashns/python-functions", 74 likes>,
Notebook <"biraj/cifar10-cnn", 2 likes>,
Notebook <"hemanth/anime-gans", 80 likes>,
Notebook <"siddhant/linear-regression", 532 likes>,
```

```
Notebook <"siddhant/python-numpy", 92 likes>,
Notebook <"sonaksh/feedforward-nn", 94 likes>,
Notebook <"tanya/cifar10-resnet", 29 likes>,
Notebook <"vikas/logistic-regression", 31 likes>,
Notebook <"vishal/python-fundamentals", 136 likes>]
```

```
def compare_twice(nb1, nb2):
    if nb1.username < nb2.username and nb1.title < nb2.title:
        return 'lesser'
    elif nb1.username == nb2.username:
        return 'equal'
    elif nb1.username > nb2.username and nb1.title > nb2.title:
        return 'greater'
```

```
merge_sort(notebooks, compare_twice)
```

```
[Notebook <"aakashns/python-functions", 74 likes>,
Notebook <"hemanth/anime-gans", 80 likes>,
Notebook <"biraj/cifar10-cnn", 2 likes>,
Notebook <"siddhant/python-numpy", 92 likes>,
Notebook <"tanya/cifar10-resnet", 29 likes>,
Notebook <"sonaksh/feedforward-nn", 94 likes>,
Notebook <"siddhant/linear-regression", 532 likes>,
Notebook <"vikas/logistic-regression", 31 likes>,
Notebook <"vishal/python-fundamentals", 136 likes>,
Notebook <"aakashns/pytorch-basics", 373 likes>]
```

```
def bubble_sort_names(notebooks):
    # Create a copy of the list, to avoid changing it
    notebooks = list(notebooks)

    # 4. Repeat the process n-1 times
    for _ in range(len(notebooks) - 1):

        # 1. Iterate over the array (except last element)
        for notebook in range(len(notebooks) - 1):

            # 2. Compare the number with
            if notebook.username[notebook] > notebook.username[notebook+1]:

                # 3. Swap the two elements
                notebook.username[notebook], notebook.username[notebook+1] = notebook.u

    # Return the sorted list
    return nums
```

```
bubble_sort_names(notebooks)
```

AttributeError

Traceback (most recent call last)

```
/tmp/ipykernel_43/1248293739.py in <module>
```

```
----> 1 bubble_sort_names(notebooks)
```

```
/tmp/ipykernel_43/1528811517.py in bubble_sort_names(notebooks)
```

```
10
```

```
11         # 2. Compare the number with
```

```
----> 12         if notebook.username[notebook] > notebook.username[notebook+1]:
```

```
13
```

```
14         # 3. Swap the two elements
```

AttributeError: 'int' object has no attribute 'username'

Let's save our work before continuing.

```
jovian.commit()
```

Summary and Exercises

We've covered the following sorting algorithms in this tutorial:

1. Bubble sort
2. Insertion sort
3. Merge sort
4. Quick sort

There are several other sorting algorithms, and most languages provide library functions for sorting that use a hybrid approach depending on the size and type of element in the list/array.

Try out some problems on sorting and divide-n-conquer here:

- <https://leetcode.com/tag/sort/>
- <https://www.techiedelight.com/sorting-interview-questions/>
- [HackerRank](#)
- <https://leetcode.com/tag/divide-and-conquer/>
- <https://www.geeksforgeeks.org/divide-and-conquer/>

Use this problem solving template: <https://jovian.ai/aakashns/python-problem-solving-template>

Assignment 3 - Divide-n-Conquer Algorithms in Python

This assignment is a part of the course ["Data Structures and Algorithms in Python"](#).

In this assignment, you will implement an efficient algorithm for polynomial multiplication.

As you go through this notebook, you will find the symbol ??? in certain places. To complete this assignment, you must replace all the ??? with appropriate values, expressions or statements to ensure that the notebook runs properly end-to-end.

Guidelines

1. Make sure to run all the code cells, otherwise you may get errors like `NameError` for undefined variables.
2. Do not change variable names, delete cells or disturb other existing code. It may cause problems during evaluation.
3. In some cases, you may need to add some code cells or new statements before or after the line of code containing the ???.
4. Since you'll be using a temporary online service for code execution, save your work by running `jovian.commit` at regular intervals.
5. Questions marked (Optional) will not be considered for evaluation, and can be skipped. They are for your learning.
6. If you are stuck, you can ask for help on the [community forum] (TODO - add link). Post errors or ask for hints, but **please don't ask for OR share the full working answer code** on the forum.
7. There are some tests included with this notebook to help you test your implementation. However, after submission your code will be tested with some hidden test cases. Make sure to test your code exhaustively to cover all edge cases.

Important Links

- Submit your work here: <https://jovian.ai/learn/data-structures-and-algorithms-in-python/assignment/assignment-3-sorting-and-divide-conquer-practice>
- Ask questions and get help: <https://jovian.ai/forum/c/data-structures-and-algorithms-in-python/assignment-3/89>
- Lesson 3 video for review: <https://jovian.ai/learn/data-structures-and-algorithms-in-python/lesson/lesson-3-sorting-algorithms-and-divide-and-conquer>
- Lesson 3 notebook for review: <https://jovian.ai/aakashns/python-sorting-divide-and-conquer>

How to Run the Code and Save Your Work

Option 1: Running using free online resources (1-click, recommended): Click the **Run** button at the top of this page and select **Run on Binder**. You can also select "Run on Colab" or "Run on Kaggle", but you'll need to create an account on [Google Colab](#) or [Kaggle](#) to use these platforms.

Option 2: Running on your computer locally: To run the code on your computer locally, you'll need to set up [Python](#) & [Conda](#), download the notebook and install the required libraries. Click the **Run** button at the top of this page, select the **Run Locally** option, and follow the instructions.

Saving your work: You can save a snapshot of the assignment to your [Jovian](#) profile, so that you can access it later and continue your work. Keep saving your work by running `jovian.commit` from time to time.

```
project='python-divide-and-conquer-assignment'
```

```
!pip install jovian --upgrade --quiet
```

```
import jovian
jovian.commit(project=project, privacy='secret', environment=None)
```

```
[jovian] Updating notebook "evanmarie/python-divide-and-conquer-assignment" on
https://jovian.ai
```

```
[jovian] Committed successfully! https://jovian.ai/evanmarie/python-divide-and-conquer-assignment
```

```
'https://jovian.ai/evanmarie/python-divide-and-conquer-assignment'
```

Problem Statement - Polynomial Multiplication

Given two polynomials represented by two lists, write a function that efficiently multiplies given two polynomials. For example, the lists `[2, 0, 5, 7]` and `[3, 4, 2]` represent the polynomials $2 + 5x^2 + 7x^3$ and $3 + 4x + 2x^2$.

Their product is

$$(2 \times 3) + (2 \times 4 + 0 \times 3)x + (2 \times 2 + 3 \times 5 + 4 \times 0)x^2 + (7 \times 3 + 5 \times 4 + 0 \times 2)x^3 \\ + (7 \times 4 + 5 \times 2)x^4 + (7 \times 2)x^5$$

i.e.

$$6 + 8x + 19x^2 + 41x^3 + 38x^4 + 14x^5$$

It can be represented by the list `[6, 8, 19, 41, 38, 14]`.

The Method

Here's the systematic strategy we'll apply for solving problems:

1. State the problem clearly. Identify the input & output formats.
2. Come up with some example inputs & outputs. Try to cover all edge cases.
3. Come up with a correct solution for the problem. State it in plain English.
4. Implement the solution and test it using example inputs. Fix bugs, if any.
5. Analyze the algorithm's complexity and identify inefficiencies, if any.
6. Apply the right technique to overcome the inefficiency. Repeat steps 3 to 6.

This approach is explained in detail in [Lesson 1](#) of the course. Let's apply this approach step-by-step.

Solution

1. State the problem clearly. Identify the input & output formats.

While this problem is stated clearly enough, it's always useful to try and express in your own words, in a way that makes it most clear for you.

Problem

Given two lists of integers, representing two polynomials, write a function that accurately multiplies the two polynomials, element by element.

Input

1. list 1 = polynomial 1
2. list 2 = polynomial 2

Output

1. List of integers representing the combined polynomials.

Based on the above, we can now create a signature of our function:

```
def multiply(poly1, poly2):  
    pass
```

```
import jovian
```

```
jovian.commit()
```

```
[jovian] Updating notebook "evanmarie/python-divide-and-conquer-assignment" on  
https://jovian.ai
```

```
[jovian] Committed successfully! https://jovian.ai/evanmarie/python-divide-and-conquer-assignment
```

```
'https://jovian.ai/evanmarie/python-divide-and-conquer-assignment'
```

2. Come up with some example inputs & outputs. Try to cover all edge cases.

Our function should be able to handle any set of valid inputs we pass into it. List a few scenarios here:

1. One list is empty
2. Both lists are empty
3. One list contains only one element aside from the zeros as place holders for nonexistent elements
4. Both lists contain only 1 element aside from zeros as place holders for nonexistent elements
5. One list is all zeros

Create a test case of each of the above scenarios. We'll express our test cases as dictionaries, to test them easily. Each dictionary will contain 2 keys: `input` (a dictionary itself containing one key for each argument to the

function and output (the expected result from the function).

```
# A normal test case
test0 = {
  'input': {
    'poly1': [2, 0, 5, 7],
    'poly2': [3, 4, 2],
  },
  'output': [6, 8, 19, 41, 38, 14]
}
```

```
# One empty list
test1 = {
  'input': {
    'poly1': [],
    'poly2': [3, 4, 5]
  },
  'output': []
}
```

```
# Two empty lists
test2 = {
  'input': {
    'poly1': [],
    'poly2': []
  },
  'output': []
}
```

```
# Working with zeros
test3 = {
  'input': {
    'poly1': [0, 4],
    'poly2': [1, 4, 5, 0, 4]
  },
  'output': [0, 4, 16, 20, 0, 16]
}
```

```
# Working with zeros
test4 = {
  'input': {
    'poly1': [0, 2],
    'poly2': [2, 0]
  },
  'output': [0, 4, 0]
}
```

```
# A list with just element
test5 = {
  'input': {
    'poly1': [4],
    'poly2': [1, 2, 3, 4]
  },
  'output': [4, 8, 12, 16]
}
```

```
test6 = {
  'input': {
    'poly1': [0, 0, 0],
    'poly2': [1, 2, 3, 4]
  },
  'output': [0]
}
```

```
# A list with all zeros, except one high exponent
test7 = {
  'input': {
    'poly1': [0, 0, 0, 7],
    'poly2': [1, 2, 3, 4]
  },
  'output': [0, 0, 0, 7, 14, 21, 28]
}
```

```
# A very long polynomial multiplied by one half its length
test8 = {
  'input': {
    'poly1': [1, 2, 3, 4, 5, 6, 7, 8, 9],
    'poly2': [1, 2, 3, 4]
  },
  'output': [1, 4, 10, 20, 30, 40, 50, 60, 70, 70, 59, 36]
}
```

```
# Giganto polynomials
test9 = {
  'input': {
    'poly1': [1, 2, 3, 4, 5, 6, 7, 8, 9],
    'poly2': [1, 2, 3, 4]
  },
  'output': [1, 4, 10, 20, 30, 40, 50, 60, 70, 70, 59, 36]
}
```

Let's store all the test cases in a list, for easier automated testing.

```
tests = [test0, test1, test2, test3, test4, test5, test6, test7, test8, test9]
```

```
import jovian
```

```
jovian.commit()
```

[jovian] Updating notebook "evanmarie/python-divide-and-conquer-assignment" on <https://jovian.ai>

[jovian] Committed successfully! <https://jovian.ai/evanmarie/python-divide-and-conquer-assignment>

```
'https://jovian.ai/evanmarie/python-divide-and-conquer-assignment'
```

3. Come up with a correct solution for the problem. State it in plain English.

Our first goal should always be to come up with a *correct* solution to the problem, which may not necessarily be the most *efficient* solution.

Here's the simplest solution: If you have lists `poly1` and `poly2` representing polynomials of length m and n respectively, the highest degree of the exponents are $m - 1$ and $n - 1$ respectively. Their product has the degree $(m - 1) + (n - 1)$ i.e $m + n - 2$. The list representing the product has the length $m + n - 1$. So, we can create a list `result` of length $m + n - 1$, and set

```
result[k] = Sum of all the pairs poly1[i] * poly2[j] where i+j = k
```

Example:

$$(2 + 5x^2 + 7x^3) \times (3 + 4x + 2x^2)$$

$$= (2 \times 3) + (2 \times 4 + 0 \times 3)x + (2 \times 2 + 3 \times 5 + 4 \times 0)x^2 + (7 \times 3 + 5 \times 4 + 0 \times 2)x^3 + (7 \times 4 + 5 \times 2)x^4 + (7 \times 2)x^5$$

$$= 6 + 8x + 19x^2 + 41x^3 + 38x^4 + 14x^5$$

Explain this solution in your own words below:

1. To get the length of the resulting list, we must combine the lengths of the two lists we are multiplying.
2. Each list's greatest exponent will be one less than its length.
3. Greatest exponent in the product list will thus equal the combine lengths of both lists minus 2.
4. The length of the resulting product list will be the length of both lists minus 1
5. Each item's index in the resulting list will be the sum of the indices from of the two input lists and will also represent the exponent level of that result list's item / coefficient.

(add more steps if required)

Let's save and upload our work before continuing.

```
import jovian
```

```
jovian.commit()
```

[jovian] Updating notebook "evanmarie/python-divide-and-conquer-assignment" on

<https://jovian.ai>

[jovian] Committed successfully! <https://jovian.ai/evanmarie/python-divide-and-conquer-assignment>

'<https://jovian.ai/evanmarie/python-divide-and-conquer-assignment>'

4. Implement the solution and test it using example inputs. Fix bugs, if any.

Implement the solution

```
def multiply_basic(poly1, poly2):
    # print("Poly1: ", poly1)
    # print("Poly2: ", poly2)

    result = [0] * (len(poly1) + len(poly2) - 1)
    # print("Result Pre-Loop: ", result)

    # Take care of edge cases
    if poly1 == [] or poly2 == []:
        return []
    if sum(poly1) == 0 or sum(poly2) == 0:
        return [0]

    # Multiply the polynomials
    for i, coeff1 in enumerate(poly1):
        for j, coeff2 in enumerate(poly2):
            # print('Loop i: ', i)
            # print('Coeff1: ', coeff1)
            # print('Loop j: ', j)
            # print('Coeff2: ', coeff2)

            result[i+j] += coeff1 * coeff2
            # print("Result Post-Loop: ", result)

    return result
```

Test your solution using the test cases you've defined above.

```
from jovian.pythondsa import evaluate_test_cases
```

```
evaluate_test_cases(multiply_basic, tests)
```

TEST CASE #0

Input:

```
{'poly1': [2, 0, 5, 7], 'poly2': [3, 4, 2]}
```

Expected Output:

[6, 8, 19, 41, 38, 14]

Actual Output:

[6, 8, 19, 41, 38, 14]

Execution Time:

0.012 ms

Test Result:

PASSED

TEST CASE #1

Input:

{'poly1': [], 'poly2': [3, 4, 5]}

Expected Output:

[]

Actual Output:

[]

Execution Time:

0.003 ms

Test Result:

PASSED

TEST CASE #2

Input:

{'poly1': [], 'poly2': []}

Expected Output:

[]

Actual Output:

[]

Execution Time:

0.004 ms

Test Result:

PASSED

TEST CASE #3

Input:

{'poly1': [0, 4], 'poly2': [1, 4, 5, 0, 4]}

Expected Output:

[0, 4, 16, 20, 0, 16]

Actual Output:

[0, 4, 16, 20, 0, 16]

Execution Time:

0.012 ms

Test Result:

PASSED

TEST CASE #4

Input:

{'poly1': [0, 2], 'poly2': [2, 0]}

Expected Output:

[0, 4, 0]

Actual Output:

[0, 4, 0]

Execution Time:

0.008 ms

Test Result:

PASSED

TEST CASE #5

Input:

```
{'poly1': [4], 'poly2': [1, 2, 3, 4]}
```

Expected Output:

```
[4, 8, 12, 16]
```

Actual Output:

```
[4, 8, 12, 16]
```

Execution Time:

```
0.004 ms
```

Test Result:

PASSED

TEST CASE #6

Input:

```
{'poly1': [0, 0, 0], 'poly2': [1, 2, 3, 4]}
```

Expected Output:

```
[0]
```

Actual Output:

```
[0]
```

Execution Time:

```
0.003 ms
```

Test Result:

PASSED

TEST CASE #7

Input:

```
{'poly1': [0, 0, 0, 7], 'poly2': [1, 2, 3, 4]}
```

Expected Output:

```
[0, 0, 0, 7, 14, 21, 28]
```

Actual Output:

```
[0, 0, 0, 7, 14, 21, 28]
```

Execution Time:

```
0.006 ms
```

Test Result:

PASSED

TEST CASE #8

Input:

```
{'poly1': [1, 2, 3, 4, 5, 6, 7, 8, 9], 'poly2': [1, 2, 3, 4]}
```

Expected Output:

```
[1, 4, 10, 20, 30, 40, 50, 60, 70, 70, 59, 36]
```

Actual Output:

```
[1, 4, 10, 20, 30, 40, 50, 60, 70, 70, 59, 36]
```

Execution Time:

```
0.008 ms
```

Test Result:

PASSED

TEST CASE #9

Input:

```
{'poly1': [1, 2, 3, 4, 5, 6, 7, 8, 9], 'poly2': [1, 2, 3, 4]}
```

Expected Output:

```
[1, 4, 10, 20, 30, 40, 50, 60, 70, 70, 59, 36]
```

Actual Output:

```
[1, 4, 10, 20, 30, 40, 50, 60, 70, 70, 59, 36]
```

Execution Time:

0.009 ms

Test Result:

PASSED

SUMMARY

TOTAL: 10, PASSED: 10, FAILED: 0

```
[[6, 8, 19, 41, 38, 14], True, 0.012),  
([], True, 0.003),  
([], True, 0.004),  
([0, 4, 16, 20, 0, 16], True, 0.012),  
([0, 4, 0], True, 0.008),  
([4, 8, 12, 16], True, 0.004),  
([0], True, 0.003),  
([0, 0, 0, 7, 14, 21, 28], True, 0.006),  
([1, 4, 10, 20, 30, 40, 50, 60, 70, 70, 59, 36], True, 0.008),  
([1, 4, 10, 20, 30, 40, 50, 60, 70, 70, 59, 36], True, 0.009)]
```

```
import jovian
```

```
jovian.commit()
```

[jovian] Updating notebook "evanmarie/python-divide-and-conquer-assignment" on <https://jovian.ai>

[jovian] Committed successfully! <https://jovian.ai/evanmarie/python-divide-and-conquer-assignment>

```
'https://jovian.ai/evanmarie/python-divide-and-conquer-assignment'
```

5. Analyze the algorithm's complexity and identify inefficiencies, if any.

Can you analyze the time and space complexity of this algorithm?

```
multiply_basic_time_complexity = 'O(N^2)'
```

```
multiply_basic_space_complexity = 'O(2N)'
```

```
import jovian
```

```
jovian.commit()
```

[jovian] Updating notebook "evanmarie/python-divide-and-conquer-assignment" on <https://jovian.ai>

[jovian] Committed successfully! <https://jovian.ai/evanmarie/python-divide-and-conquer-assignment>

'<https://jovian.ai/evanmarie/python-divide-and-conquer-assignment>'

6. Apply the right technique to overcome the inefficiency. Repeat steps 3 to 6.

We can apply the divide and conquer technique to solve this problem more efficiently. Given two polynomials A and B , we can express each of them as a sum of two polynomials as follows:

The Divide Step: Define

$$A_0(x) = a_0 + a_1x + \dots + a_{\lfloor \frac{n}{2} \rfloor - 1}x^{\lfloor \frac{n}{2} \rfloor - 1},$$

$$A_1(x) = a_{\lfloor \frac{n}{2} \rfloor} + a_{\lfloor \frac{n}{2} \rfloor + 1}x + \dots + a_nx^{n - \lfloor \frac{n}{2} \rfloor}.$$

$$\text{Then } A(x) = A_0(x) + A_1(x)x^{\lfloor \frac{n}{2} \rfloor}.$$

Similarly we define $B_0(x)$ and $B_1(x)$ such that

$$B(x) = B_0(x) + B_1(x)x^{\lfloor \frac{n}{2} \rfloor}.$$

Then

$$A(x)B(x) = A_0(x)B_0(x) + A_0(x)B_1(x)x^{\lfloor \frac{n}{2} \rfloor} + A_1(x)B_0(x)x^{\lfloor \frac{n}{2} \rfloor} + A_1(x)B_1(x)x^{2\lfloor \frac{n}{2} \rfloor}.$$

We need to compute the terms $A_0 * B_0$, $A_1 * B_0 + A_0 * B_1$ and $A_1 * B_1$. This can obviously be done using 4 multiplications, but here's a way of doing it with just three multiplications:

$$Y = (A_0 + A_1)(B_0 + B_1)$$

$$U = A_0B_0$$

$$Z = A_1B_1$$

U and Z are what we originally wanted and

$$A_0B_1 + A_1B_0 = Y - U - Z.$$

Each of the products can themselves be computed recursively. For a more detailed explanation of this approach see <http://www.cse.ust.hk/~dekai/271/notes/L03/L03.pdf>.

Need help? Discuss and ask questions on the forum: <https://jovian.ai/forum/c/data-structures-and-algorithms-in-python/assignment-3/89>

7. Come up with a correct solution for the problem. State it in plain English.

Explain the approach described above in your own words below:

EX: $X + X^2 = A = [0, 1, 1]$ $A_0 = [0, 1]$ $A_1 = [0, 1]$ (factored out $X^{n/2}$)

1. Factor out $x^{n/2}$
2. A_0 first half of poly1, A_1 is second of poly1
3. B_0 first half of poly2, B_1 is second of poly2
4. Divide A and B down recursively this way,
5. Multiply the split down (A_0+A_1) times (B_0+B_1)
6. Subtract from that the product of A_0 and B_0
7. Subtract further the product of A_1 and B_1

(add more steps if required)

Let's save and upload our work before continuing.

```
import jovian
```

```
jovian.commit()
```

```
[jovian] Updating notebook "evanmarie/python-divide-and-conquer-assignment" on  
https://jovian.ai
```

```
[jovian] Committed successfully! https://jovian.ai/evanmarie/python-divide-and-conquer-assignment
```

```
'https://jovian.ai/evanmarie/python-divide-and-conquer-assignment'
```

8. Implement the solution and test it using example inputs. Fix bugs, if any.

We are now ready to implement the solution. You may find the following functions `add`, `split` and `increase_exponent` useful.

```
def add(poly1, poly2):  
    """Add two polynomials"""  
    result = [0] * max(len(poly1), len(poly2))  
    for i in range(len(result)):  
        if i < len(poly1):  
            result[i] += poly1[i]  
        if i < len(poly2):  
            result[i] += poly2[i]  
    return result
```

```
add([1, 2, 3, 4], [0, 4, 3])
```

```
[1, 6, 6, 4]
```

```
def split(poly1, poly2):  
    """Split each polynomial into two smaller polynomials"""  
    mid = max(len(poly1), len(poly2)) // 2  
    return (poly1[:mid] or [0], poly1[mid:] or [0]), \  
           (poly2[:mid] or [0], poly2[mid:] or [0])
```

```
split([1, 2, 3, 4], [0, 4, 3, 6, 7, 8, 2])
```

```
(([1, 2, 3], [4]), ([0, 4, 3], [6, 7, 8, 2]))
```

```
def increase_exponent(poly, n):  
    """Multiply poly1 by x^n"""  
    return [0] * n + poly
```

```
increase_exponent([1, 2, 3, 4], 3)
```

```
[0, 0, 0, 1, 2, 3, 4]
```

Implement the optimized multiplication algorithm below. You may use the some or all of the helper functions defined above.

Test your solution using the empty cells below.

```
def multiply_polys(poly1, poly2):  
    # The product of the sections being multiplied:  
    product = []  
    # If either element is less than 1 item in length  
    if len(poly1) < 1 or len(poly2) < 1:  
        return product  
  
    else:  
        # Loop through poly1 and create a list of segments to be used  
        for element1 in range(len(poly1)):  
            product_segments = []  
            # Loop through poly1 and poly2, creating a list of segments, which are  
            # the multiplied smaller segments from poly1 and poly2, to be used  
            for element2 in range(len(poly2)):  
                product_segments.append(poly1[element1] * poly2[element2])  
  
            # adjust the exponents of the segments and element from poly 1  
            product_segments = increase_exponent(product_segments, element1)  
            if element1 == 0:  
                # is all of the product segments  
                product = product_segments  
            else:
```

```
    # the product is the current product plus all the product segments
    product = add(product, product_segments)
```

```
return product
```

```
# Please pardon all of the comments, but this algorithm made them
# necessary for my own sanity. Thank you for understanding!
```

```
def multiply_optimized(poly1, poly2):
```

```
    # Check edge cases: if the lengths of the lists are 0
```

```
    if len(poly1) == 0 and len(poly2) == 0:
```

```
        return []
```

```
    # Check edge cases: if the either list is an empty list
```

```
    if poly1 == [] or poly2 == []:
```

```
        return []
```

```
    # Check edge cases: if the either list is all zeros
```

```
    if sum(poly1) == 0 or sum(poly2) == 0:
```

```
        return [0]
```

```
    # Check edge cases: if the either list is an empty list
```

```
    if poly1 == [] or poly2 == []:
```

```
        return []
```

```
    # Check edge cases: if poly1 list is empty
```

```
    elif len(poly1) == 0 and len(poly2) > 0:
```

```
        return poly2
```

```
    # Check edge cases: if poly2 list is empty
```

```
    elif len(poly1) > 0 and len(poly2) == 0:
```

```
        return poly1
```

```
    # If the polynomials are more than one element in length,
```

```
    # split them
```

```
    if len(poly1) > 1 and len(poly2) > 1:
```

```
        # a0 and a1 are the two halves of poly1, and b0 and b1
```

```
        # are the two halves of poly2
```

```
        (a0, a1), (b0, b1) = split(poly1, poly2)
```

```
    # Write the code version of the algorithm above using the split
```

```
    # sections of the algorithm creating Y, U, and Z and performing
```

```
    # the operations explained.
```

```
    # Y = (a0*b1) + (a1 * b0)
```

```
    algo_y = add(multiply_polys(a0, b1), multiply_polys(a1, b0))
```

```
    # U = a0 * b0
```

```
    algo_u = multiply_polys(a0, b0)
```

```
    # Z = a1 * b1
```

```
    algo_z = multiply_polys(a1, b1)
```

```
    # adjust the exponents in Y (^1/2N)
```

```
    algo_y = increase_exponent(algo_y, max(len(a0), len(b0)))
```

```
    # adjust the exponents in Z (^1/2N)
```

```
    algo_z = increase_exponent(algo_z, ((len(poly1) + len(poly2)) - 1) - len(algo_z
```

```
    # add Y and Z to get closer to a final combination of elements
```

```
    result = add(algo_y, algo_z)
    # add U to the previous sum, resulting in our combination of all elements
    result = add(algo_u, result)

else:
    # If the above does not apply, just multiply poly1 and poly2
    result = multiply_polys(poly1, poly2)

return result
```

```
evaluate_test_cases(multiply_optimized, tests)
```

TEST CASE #0

Input:

```
{'poly1': [2, 0, 5, 7], 'poly2': [3, 4, 2]}
```

Expected Output:

```
[6, 8, 19, 41, 38, 14]
```

Actual Output:

```
[6, 8, 19, 41, 38, 14]
```

Execution Time:

```
0.041 ms
```

Test Result:

```
PASSED
```

TEST CASE #1

Input:

```
{'poly1': [], 'poly2': [3, 4, 5]}
```

Expected Output:

```
[]
```

Actual Output:

```
[]
```

Execution Time:

0.003 ms

Test Result:

PASSED

TEST CASE #2

Input:

```
{'poly1': [], 'poly2': []}
```

Expected Output:

```
[]
```

Actual Output:

```
[]
```

Execution Time:

0.003 ms

Test Result:

PASSED

TEST CASE #3

Input:

```
{'poly1': [0, 4], 'poly2': [1, 4, 5, 0, 4]}
```

Expected Output:

```
[0, 4, 16, 20, 0, 16]
```

Actual Output:

```
[0, 4, 16, 20, 0, 16]
```

Execution Time:

0.04 ms

Test Result:

PASSED

TEST CASE #4

Input:

```
{'poly1': [0, 2], 'poly2': [2, 0]}
```

Expected Output:

```
[0, 4, 0]
```

Actual Output:

```
[0, 4, 0]
```

Execution Time:

```
0.023 ms
```

Test Result:

PASSED

TEST CASE #5

Input:

```
{'poly1': [4], 'poly2': [1, 2, 3, 4]}
```

Expected Output:

```
[4, 8, 12, 16]
```

Actual Output:

```
[4, 8, 12, 16]
```

Execution Time:

```
0.008 ms
```

Test Result:

PASSED

TEST CASE #6

Input:

```
{'poly1': [0, 0, 0], 'poly2': [1, 2, 3, 4]}
```

Expected Output:

[0]

Actual Output:

[0]

Execution Time:

0.003 ms

Test Result:

PASSED

TEST CASE #7

Input:

{'poly1': [0, 0, 0, 7], 'poly2': [1, 2, 3, 4]}

Expected Output:

[0, 0, 0, 7, 14, 21, 28]

Actual Output:

[0, 0, 0, 7, 14, 21, 28]

Execution Time:

0.043 ms

Test Result:

PASSED

TEST CASE #8

Input:

{'poly1': [1, 2, 3, 4, 5, 6, 7, 8, 9], 'poly2': [1, 2, 3, 4]}

Expected Output:

[1, 4, 10, 20, 30, 40, 50, 60, 70, 70, 59, 36]

Actual Output:

[1, 4, 10, 20, 30, 40, 50, 60, 70, 70, 59, 36]

Execution Time:

0.081 ms

Test Result:

PASSED

TEST CASE #9

Input:

{'poly1': [1, 2, 3, 4, 5, 6, 7, 8, 9], 'poly2': [1, 2, 3, 4]}

Expected Output:

[1, 4, 10, 20, 30, 40, 50, 60, 70, 70, 59, 36]

Actual Output:

[1, 4, 10, 20, 30, 40, 50, 60, 70, 70, 59, 36]

Execution Time:

0.08 ms

Test Result:

PASSED

SUMMARY

TOTAL: 10, PASSED: 10, FAILED: 0

[[[6, 8, 19, 41, 38, 14], True, 0.041),
([], True, 0.003),
([], True, 0.003),
([0, 4, 16, 20, 0, 16], True, 0.04),
([0, 4, 0], True, 0.023),
([4, 8, 12, 16], True, 0.008),
([0], True, 0.003),
([0, 0, 0, 7, 14, 21, 28], True, 0.043),
([1, 4, 10, 20, 30, 40, 50, 60, 70, 70, 59, 36], True, 0.081),
([1, 4, 10, 20, 30, 40, 50, 60, 70, 70, 59, 36], True, 0.08)]

```
import jovian
```

```
jovian.commit()
```

[jovian] Updating notebook "evanmarie/python-divide-and-conquer-assignment" on <https://jovian.ai>

[jovian] Committed successfully! <https://jovian.ai/evanmarie/python-divide-and-conquer-assignment>

'<https://jovian.ai/evanmarie/python-divide-and-conquer-assignment>'

Make a Submission

Congrats! You have now implemented hash tables from scratch. The rest of this assignment is optional.

You can make a submission on this page: <https://jovian.ai/learn/data-structures-and-algorithms-in-python/assignment/assignment-3-sorting-and-divide-conquer-practice>

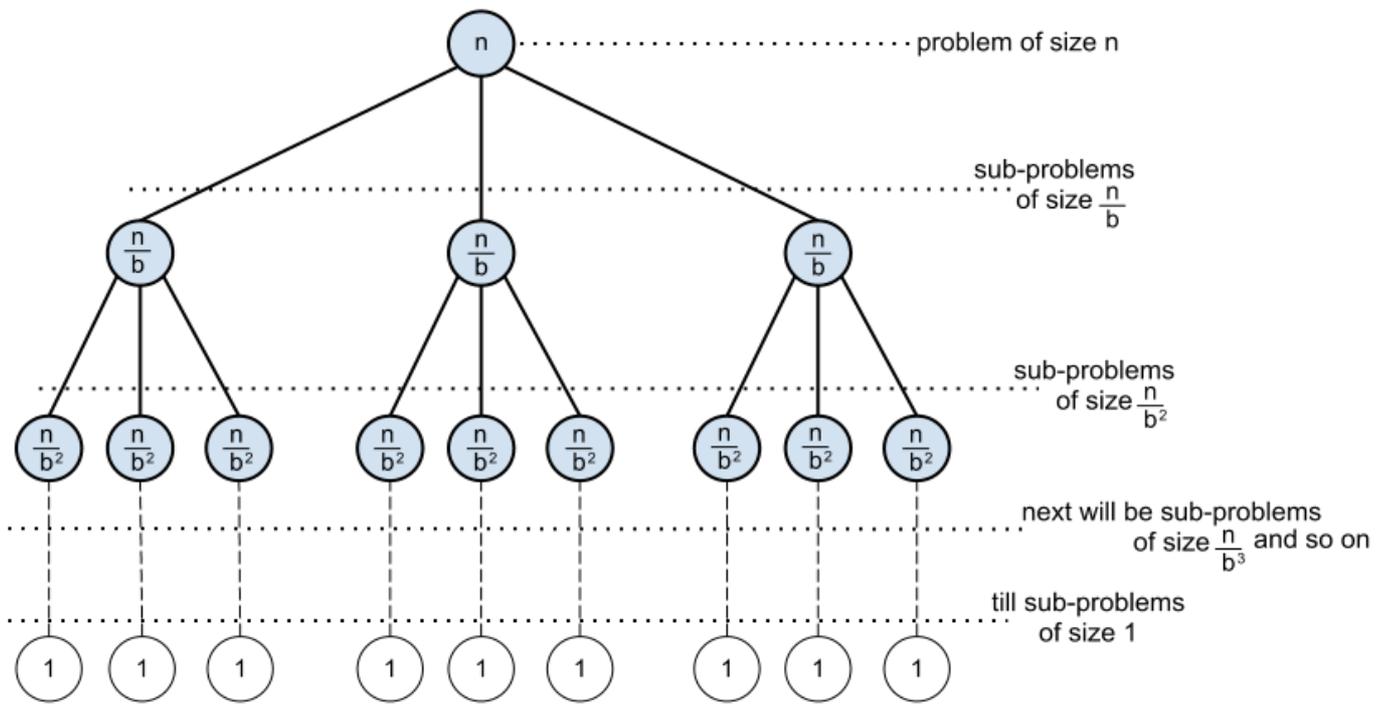
Submit the link to your Jovian notebook (the output of the previous cell). You can also make a direct submission by executing the following cell:

```
jovian.submit(assignment="pythondsa-assignment3")
```

(Optional) 9. Analyze the algorithm's complexity and identify inefficiencies, if any.

Can you analyze the time and space complexity of this algorithm?

Hint: See the tree of subproblems below ([source](#)). Substitute the right values for n and b to determine the time complexity.



The height of the tree is answer to the following question:

How many times we divide problem of size n by b until we get down to problem of size 1? The other way of asking same question:

when $\frac{n}{b^x} = 1$ [in binary tree $b = 2$]

i.e. $n = b^x$ which is $\log_b n$ [by definition of logarithm]

```
import jovian
```

```
jovian.commit()
```

Longest Common Subsequence

QUESTION 1: Write a function to find the length of the **longest common subsequence** between two sequences. E.g. Given the strings "serendipitous" and "precipitation", the longest common subsequence is "reipito" and its length is 7.

A "sequence" is a group of items with a deterministic ordering. Lists, tuples and ranges are some common sequence types in Python.

A "subsequence" is a sequence obtained by deleting zero or more elements from another sequence. For example, "edpt" is a subsequence of "serendipitous".

General case

s e **r** **e** n d **i** **p** **i** **t** **o** u s
p **r** **e** c **i** **p** **i** **t** a t i **o** n

Test cases

1. General case (string)
2. General case (list)
3. No common subsequence
4. One is a subsequence of the other
5. One sequence is empty
6. Both sequences are empty
7. Multiple subsequences with same length
 - A. "abcdef" and "badcfe"

Longest common subsequence test cases:

```
T0 = {
  'input': {
    'sequence1': 'serendipitous',
    'sequence2': 'precipitation'
  },
  'output': 7
}

T1 = {
  'input': {
    'sequence1': [1, 3, 5, 6, 7, 2, 5, 2, 3],
    'sequence2': [6, 2, 4, 7, 1, 5, 6, 2, 3]
  },
  'output': 5
}
```

```
T2 = {
  'input': {
    'sequence1': 'longest',
    'sequence2': 'stone'
  },
  'output': 3
}

T3 = {
  'input': {
    'sequence1': 'asdfwevad',
    'sequence2': 'opkpoiklkj'
  },
  'output': 0
}

T4 = {
  'input': {
    'sequence1': 'dense',
    'sequence2': 'condensed'
  },
  'output': 5
}

T5 = {
  'input': {
    'sequence1': '',
    'sequence2': 'opkpoiklkj'
  },
  'output': 0
}

T6 = {
  'input': {
    'sequence1': '',
    'sequence2': ''
  },
  'output': 0
}

T7 = {
  'input': {
    'sequence1': 'abcdef',
    'sequence2': 'badcfe'
  },
  'output': 3
}

T8 = {
  'input': {
    'sequence1': 'blasphemous',
```

```
    'sequence2': 'contagious'  
  },  
  'output': 4  
}
```

```
longest_subsequence_tests = [T0, T1, T2, T3, T4, T5, T6, T7, T8]
```

Recursive Solution

1. Create two counters `idx1` and `idx2` starting at 0. Our recursive function will compute the LCS of `seq1[idx1:]` and `seq2[idx2:]`
2. If `seq1[idx1]` and `seq2[idx2]` are equal, then this character belongs to the LCS of `seq1[idx1:]` and `seq2[idx2:]` (why?). Further the length this is LCS is one more than LCS of `seq1[idx1+1:]` and `seq2[idx2+1:]`

analogous
alcohol

3. If not, then the LCS of `seq1[idx1:]` and `seq2[idx2:]` is the longer one among the LCS of `seq1[idx1+1:]`, `seq2[idx2:]` and the LCS of `seq1[idx1:]`, `seq2[idx2+1:]`

absent
best

5. If either `seq1[idx1:]` or `seq2[idx2:]` is empty, then their LCS is empty.

Here's what the tree of recursive calls looks like:


```
# return whichever had the better result, path1 or path2, which is the longest  
# subsequence once we have made our way all the way through the two sequences.  
return max(path1, path2)
```

```
from jovian.pythondsa import evaluate_test_cases
```

```
%%time  
longest_subsequence_recursive(T8['input']['sequence1'], T8['input']['sequence2'])
```

CPU times: user 69.6 ms, sys: 1.03 ms, total: 70.7 ms

Wall time: 69.1 ms

4

```
%%time  
longest_subsequence_recursive(T1['input']['sequence1'], T1['input']['sequence2'])
```

CPU times: user 4.2 ms, sys: 25 µs, total: 4.22 ms

Wall time: 4.16 ms

5

```
%%time  
longest_subsequence_recursive(**T3['input']) == T3['output']
```

```
# Shortcut way to run tests, since I renamed seq1 and 2 to sequence1 and 2  
# in the tests to match my variable names.  
# This way it compares the output of running my code on the given test  
# and returns True for success and False for failure.
```

CPU times: user 75.1 ms, sys: 2.09 ms, total: 77.2 ms

Wall time: 75.4 ms

True

```
evaluate_test_cases(longest_subsequence_recursive, longest_subsequence_tests)
```

TEST CASE #0

Input:

```
{'sequence1': 'serendipitous', 'sequence2': 'precipitation'}
```

Expected Output:

7

Actual Output:

7

Execution Time:

274.021 ms

Test Result:

PASSED

TEST CASE #1

Input:

```
{'sequence1': [1, 3, 5, 6, 7, 2, 5, 2, 3], 'sequence2': [6, 2, 4, 7, 1, 5, 6, 2, 3]}
```

Expected Output:

5

Actual Output:

5

Execution Time:

4.186 ms

Test Result:

PASSED

TEST CASE #2

Input:

```
{'sequence1': 'longest', 'sequence2': 'stone'}
```

Expected Output:

3

Actual Output:

3

Execution Time:

0.167 ms

Test Result:

PASSED

TEST CASE #3

Input:

{'sequence1': 'asdfwevad', 'sequence2': 'opkpoiklk1j'}

Expected Output:

0

Actual Output:

0

Execution Time:

73.718 ms

Test Result:

PASSED

TEST CASE #4

Input:

{'sequence1': 'dense', 'sequence2': 'condensed'}

Expected Output:

5

Actual Output:

5

Execution Time:

0.142 ms

Test Result:

PASSED

TEST CASE #5

Input:

```
{'sequence1': '', 'sequence2': 'opkpoiklk1j'}
```

Expected Output:

0

Actual Output:

0

Execution Time:

0.005 ms

Test Result:

PASSED

TEST CASE #6

Input:

```
{'sequence1': '', 'sequence2': ''}
```

Expected Output:

0

Actual Output:

0

Execution Time:

0.003 ms

Test Result:

PASSED

TEST CASE #7

Input:

```
{'sequence1': 'abcdef', 'sequence2': 'badcfe'}
```

Expected Output:

3

Actual Output:

3

Execution Time:

0.049 ms

Test Result:

PASSED

TEST CASE #8

Input:

```
{'sequence1': 'blasphemous', 'sequence2': 'contagious'}
```

Expected Output:

4

Actual Output:

4

Execution Time:

72.816 ms

Test Result:

PASSED

SUMMARY

TOTAL: 9, PASSED: 9, FAILED: 0

[(7, True, 274.021),

```
(5, True, 4.186),
(3, True, 0.167),
(0, True, 73.718),
(5, True, 0.142),
(0, True, 0.005),
(0, True, 0.003),
(3, True, 0.049),
(4, True, 72.816)]
```

```
# Defining function to find length of the longest subsequence between two sequences.
# Arguments = two sequences and two pointers to keep track of the search for common
# characters in the sequences
```

```
def longest_subsequence_recursive2(sequence1, sequence2, index1=0, index2=0):
    # This version will keep track of what the subsequences actually are in
    # this list to which I will add all matches.
    subsequence = []

    # if the index pointers reach the the end of the sequences they are tracking
    if index1 == len(sequence1) or index2 == len(sequence2):

        # Reached the end
        return 0

    # if location of index1 matches that of index2, we have a match
    if sequence1[index1] == sequence2[index2]:

        # add the matching item to the subsequence list that keeps track of
        # matches.
        subsequence.append(sequence1[index1])

        # return recursively calling the function on the sequences and adding to the
        # count of subsequence items and moving the index pointers up by 1.
        return 1 + longest_subsequence_recursive(sequence1, sequence2,
                                                index1+1, index2+1)

    else:
        # this is where we split and choose one "child node" or the other, referred
        # to as path1 or path2
        # move index1 forward, leave index2 where it is
        path1 = longest_subsequence_recursive(sequence1, sequence2,
                                             index1+1, index2)
        path2 = longest_subsequence_recursive(sequence1, sequence2,
                                             index1, index2+1)

        # return whichever had the better result, path1 or path2, which is the longest
        # subsequence once we have made our way all the way through the two sequences.
        return max(path1, path2)
```

```
print("SUBSEQUENCE: ", subsequence)
```

```
evaluate_test_cases(longest_subsequence_recursive2, longest_subsequence_tests)
```

TEST CASE #0

Input:

```
{'sequence1': 'serendipitous', 'sequence2': 'precipitation'}
```

Expected Output:

7

Actual Output:

7

Execution Time:

257.263 ms

Test Result:

PASSED

TEST CASE #1

Input:

```
{'sequence1': [1, 3, 5, 6, 7, 2, 5, 2, 3], 'sequence2': [6, 2, 4, 7, 1, 5, 6, 2, 3]}
```

Expected Output:

5

Actual Output:

5

Execution Time:

4.02 ms

Test Result:

PASSED

TEST CASE #2

Input:

```
{'sequence1': 'longest', 'sequence2': 'stone'}
```

Expected Output:

3

Actual Output:

3

Execution Time:

0.203 ms

Test Result:

PASSED

TEST CASE #3

Input:

```
{'sequence1': 'asdfwevad', 'sequence2': 'opkpoiklk1j'}
```

Expected Output:

0

Actual Output:

0

Execution Time:

77.236 ms

Test Result:

PASSED

TEST CASE #4

Input:

```
{'sequence1': 'dense', 'sequence2': 'condensed'}
```

Expected Output:

5

Actual Output:

5

Execution Time:

0.148 ms

Test Result:

PASSED

TEST CASE #5

Input:

```
{'sequence1': '', 'sequence2': 'opkpoiklk1j'}
```

Expected Output:

0

Actual Output:

0

Execution Time:

0.003 ms

Test Result:

PASSED

TEST CASE #6

Input:

```
{'sequence1': '', 'sequence2': ''}
```

Expected Output:

0

Actual Output:

0

Execution Time:

0.002 ms

Test Result:

PASSED

TEST CASE #7

Input:

{'sequence1': 'abcdef', 'sequence2': 'badcfe'}

Expected Output:

3

Actual Output:

3

Execution Time:

0.05 ms

Test Result:

PASSED

TEST CASE #8

Input:

{'sequence1': 'blasphemous', 'sequence2': 'contagious'}

Expected Output:

4

Actual Output:

4

Execution Time:

74.064 ms

Test Result:

PASSED

SUMMARY

TOTAL: 9, PASSED: 9, FAILED: 0

```
[(7, True, 257.263),  
(5, True, 4.02),  
(3, True, 0.203),  
(0, True, 77.236),  
(5, True, 0.148),  
(0, True, 0.003),  
(0, True, 0.002),  
(3, True, 0.05),  
(4, True, 74.064)]
```

```
%%time  
longest_subsequence_recursive2(T8['input']['sequence1'], T8['input']['sequence2'])
```

CPU times: user 73.3 ms, sys: 52 µs, total: 73.3 ms

Wall time: 71.8 ms

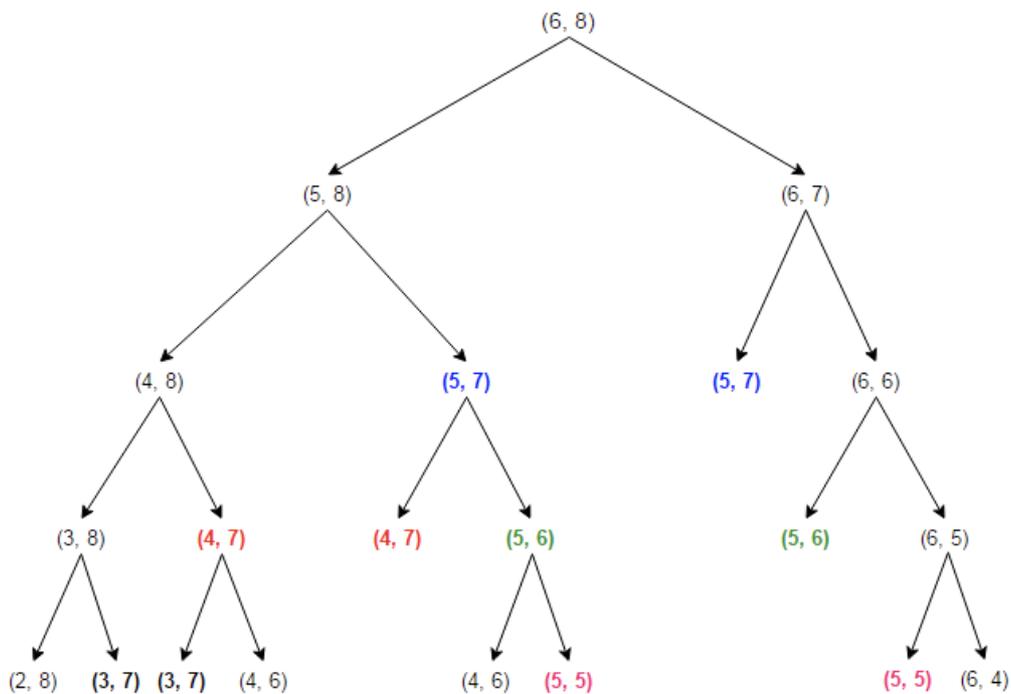
4

Complexity Analysis

Worst case occurs when each time we have to try 2 subproblems i.e. when the sequences have no common elements.

```
l  x  y  c  t  z  
  
a  b  d  e  f  g  h  j
```

Here's what the tree looks like in such a case (source - Techie Delight):



All the leaf nodes are $(0, 0)$. Can you count the number of leaf nodes?

HINT: Count the number of unique paths from root to leaf. The length of each path is $m+n$ and at each level there are 2 choices.

Based on the above can you infer that the time complexity is $O(2^{m+n})$.

```
# Now for the version using memoization to save on time complexity and to
# create more efficiency.
```

```
def longest_subsequence_memoized(sequence1, sequence2):
```

```
# dictionary to save us from so many element calls and cut down on
# time complexity
```

```
memo = {}
```

```
# recursive function, which will have the index trackers
```

```
def recurse(index1=0, index2=0):
```

```
# Create key for keeping track of elements in sequences
```

```
key = (index1, index2)
```

```
# if the key already exists in memo, return it
```

```
if key in memo:
```

```
    return memo[key]
```

```
# else if we are at the end of a sequence, return 0, because there
# is nothing left to cycle through
```

```
elif index1 == len(sequence1) or index2 == len(sequence2):
```

```
    memo[key] = 0
```

```
# else if we have a match between sequences, we recurse over the
# the next two index numbers.
```

```
elif sequence1[index1] == sequence2[index2]:
```

```
memo[key] = 1 + recurse(index1+1, index2+1)

# if the two are not equal, we make our path choice
else:
    memo[key] = max(recurse(index1+1, index2), recurse(index1, index2+1))

# return the memo entry for the key, ending the unnecessary recursion
return memo[key]

# (0,0) = the whole string, over which we will recurse
return recurse(0,0)
```

```
evaluate_test_cases(longest_subsequence_memoized, longest_subsequence_tests)
```

TEST CASE #0

Input:

```
{'sequence1': 'serendipitous', 'sequence2': 'precipitation'}
```

Expected Output:

7

Actual Output:

7

Execution Time:

0.24 ms

Test Result:

PASSED

TEST CASE #1

Input:

```
{'sequence1': [1, 3, 5, 6, 7, 2, 5, 2, 3], 'sequence2': [6, 2, 4, 7, 1, 5, 6, 2, 3]}
```

Expected Output:

5

Actual Output:

5

Execution Time:

0.092 ms

Test Result:

PASSED

TEST CASE #2

Input:

```
{'sequence1': 'longest', 'sequence2': 'stone'}
```

Expected Output:

3

Actual Output:

3

Execution Time:

0.052 ms

Test Result:

PASSED

TEST CASE #3

Input:

```
{'sequence1': 'asdfwevad', 'sequence2': 'opkpoiklk1j'}
```

Expected Output:

0

Actual Output:

0

Execution Time:

0.163 ms

Test Result:

PASSED

TEST CASE #4

Input:

```
{'sequence1': 'dense', 'sequence2': 'condensed'}
```

Expected Output:

5

Actual Output:

5

Execution Time:

0.043 ms

Test Result:

PASSED

TEST CASE #5

Input:

```
{'sequence1': '', 'sequence2': 'opkpoiklk1j'}
```

Expected Output:

0

Actual Output:

0

Execution Time:

0.003 ms

Test Result:

PASSED

TEST CASE #6

Input:

```
{'sequence1': '', 'sequence2': ''}
```

Expected Output:

0

Actual Output:

0

Execution Time:

0.003 ms

Test Result:

PASSED

TEST CASE #7

Input:

```
{'sequence1': 'abcdef', 'sequence2': 'badcfe'}
```

Expected Output:

3

Actual Output:

3

Execution Time:

0.037 ms

Test Result:

PASSED

TEST CASE #8

Input:

```
{'sequence1': 'blasphemous', 'sequence2': 'contagious'}
```

Expected Output:

4

Actual Output:

4

Execution Time:

0.152 ms

Test Result:

PASSED

SUMMARY

TOTAL: 9, PASSED: 9, FAILED: 0

- (7, True, 0.24),
- (5, True, 0.092),
- (3, True, 0.052),
- (0, True, 0.163),
- (5, True, 0.043),
- (0, True, 0.003),
- (0, True, 0.003),
- (3, True, 0.037),
- (4, True, 0.152)]

Dynamic programming

1. Create a table of size $(n_1+1) * (n_2+1)$ initialized with 0s, where n_1 and n_2 are the lengths of the sequences. $table[i][j]$ represents the longest common subsequence of $seq1[:i]$ and $seq2[:j]$. Here's what the table looks like (source: Kevin Mavani, Medium).

	-	A	G	A	C	T	G	T	C
-	0	0	0	0	0	0	0	0	0
T	0	0	0	0	0	1	1	1	1
A	0	1	1	1	1	1	1	1	1
G	0	1	2	2	2	2	2	2	2
T	0	1	2	2	2	3	3	3	3
C	0	1	2	2	3	3	3	3	4
A	0	1	2	3	3	3	3	3	4
C	0	1	2	3	4	4	4	4	4
G	0	1	2	3	4	4	5	5	5

2. If $seq1[i]$ and $seq2[j]$ are equal, then $table[i+1][j+1] = 1 + table[i][j]$

3. If $seq1[i]$ and $seq2[j]$ are equal, then $table[i+1][j+1] = \max(table[i][j+1], table[i+1][j])$

Verify that the complexity of the dynamic programming approach is $O(N1 * N2)$.

```
# Creating a table for our function  
# We will create an extra row and column so that we get rid of the zero index to make c  
len1, len2 = 5, 7  
[[0 for x in range(len2)] for x in range(len1)]
```

```
[[0, 0, 0, 0, 0, 0, 0],  
 [0, 0, 0, 0, 0, 0, 0],  
 [0, 0, 0, 0, 0, 0, 0],  
 [0, 0, 0, 0, 0, 0, 0],  
 [0, 0, 0, 0, 0, 0, 0]]
```

```
def longest_subsequence_dynamic(sequence1, sequence2):  
  
    # Creating the length variables for our table  
    len1, len2 = len(sequence1), len(sequence2)  
    # Create table, populating with zeros, adding 1 for our unused row, column  
    table = [[0 for x in range(len2+1)] for x in range(len1+1)]  
  
    # Iterate over rows  
    for index1 in range(len1):  
        # Iterate over our columns  
        for index2 in range(len2):  
            if sequence1[index1] == sequence2[index2]:  
                # Go to the value diagonally right, which will be 1 + the current  
                table[index1+1][index2+1] = 1 + table[index1][index2]  
  
            else:  
                # go to whichever is larger, back one row or back one column  
                table[index1+1][index2+1] = max(table[index1][index2+1], table[index1+1][index2])  
  
    # Return the bottom right cell, which will have the count of the longest  
    # common subsequence.  
    return table[-1][-1]
```

```
evaluate_test_cases(longest_subsequence_dynamic, longest_subsequence_tests)
```

TEST CASE #0

Input:

```
{'sequence1': 'serendipitous', 'sequence2': 'precipitation'}
```

Expected Output:

7

Actual Output:

7

Execution Time:

0.112 ms

Test Result:

PASSED

TEST CASE #1

Input:

{'sequence1': [1, 3, 5, 6, 7, 2, 5, 2, 3], 'sequence2': [6, 2, 4, 7, 1, 5, 6, 2, 3]}

Expected Output:

5

Actual Output:

5

Execution Time:

0.064 ms

Test Result:

PASSED

TEST CASE #2

Input:

{'sequence1': 'longest', 'sequence2': 'stone'}

Expected Output:

3

Actual Output:

3

Execution Time:

0.03 ms

Test Result:

PASSED

TEST CASE #3

Input:

```
{'sequence1': 'asdfwevad', 'sequence2': 'opkpoiklk1j'}
```

Expected Output:

0

Actual Output:

0

Execution Time:

0.066 ms

Test Result:

PASSED

TEST CASE #4

Input:

```
{'sequence1': 'dense', 'sequence2': 'condensed'}
```

Expected Output:

5

Actual Output:

5

Execution Time:

0.032 ms

Test Result:

PASSED

TEST CASE #5

Input:

```
{'sequence1': '', 'sequence2': 'opkpoiklk1j'}
```

Expected Output:

0

Actual Output:

0

Execution Time:

0.006 ms

Test Result:

PASSED

TEST CASE #6

Input:

```
{'sequence1': '', 'sequence2': ''}
```

Expected Output:

0

Actual Output:

0

Execution Time:

0.004 ms

Test Result:

PASSED

TEST CASE #7

Input:

```
{'sequence1': 'abcdef', 'sequence2': 'badcfe'}
```

Expected Output:

3

Actual Output:

3

Execution Time:

0.029 ms

Test Result:

PASSED

TEST CASE #8

Input:

```
{'sequence1': 'blasphemous', 'sequence2': 'contagious'}
```

Expected Output:

4

Actual Output:

4

Execution Time:

0.073 ms

Test Result:

PASSED

SUMMARY

TOTAL: 9, PASSED: 9, FAILED: 0

```
[(7, True, 0.112),  
(5, True, 0.064),  
(3, True, 0.03),  
(0, True, 0.066),  
(5, True, 0.032),  
(0, True, 0.006),
```

```
(0, True, 0.004),  
(3, True, 0.029),  
(4, True, 0.073)]
```

```
# complexity of the dynamic programming approach is  $O(N1*N2)$   
# 2 for-loops in which we are doing one comparison and one addition and taking a max
```

0-1 Knapsack Problem

Problem statement

You're in charge of selecting a football (soccer) team from a large pool of players. Each player has a cost, and a rating. You have a limited budget. What is the highest total rating of a team that fits within your budget. Assume that there's no minimum or maximum team size.

General problem statement:

Given n elements, each of which has a weight and a profit, determine the maximum profit that can be obtained by selecting a subset of the elements weighing no more than w .

Profit	[2, 3, 1, 5, 4, 7]
Weight	[4, 5, 1, 3, 2, 5]
Capacity	15

Test cases:

1. Some generic test cases
2. All the elements can be included
3. None of the elements can be included
4. Only one of the elements can be included
5. You do not use complete capacity, i.e. a solution with a lower capacity has high profit.

Inputs, Outputs, and Defining the Problem

```
# > input = weights (or can be viewed as cost), represented as a list of numbers  
# > input = profit (can be viewed as benefit), represented as a list of numbers  
# > input = capacity, the max weight you are allowed  
# > output = max profit, the maximum profit that can be obtained from selecting of total
```

Knapsack test cases:

```
test0 = {
  'input': {
    'capacity': 165,
    'weights': [23, 31, 29, 44, 53, 38, 63, 85, 89, 82],
    'profits': [92, 57, 49, 68, 60, 43, 67, 84, 87, 72]
  },
  'output': 309
}

test1 = {
  'input': {
    'capacity': 3,
    'weights': [4, 5, 6],
    'profits': [1, 2, 3]
  },
  'output': 0
}

test2 = {
  'input': {
    'capacity': 4,
    'weights': [4, 5, 1],
    'profits': [1, 2, 3]
  },
  'output': 3
}

test3 = {
  'input': {
    'capacity': 170,
    'weights': [41, 50, 49, 59, 55, 57, 60],
    'profits': [442, 525, 511, 593, 546, 564, 617]
  },
  'output': 1735
}

test4 = {
  'input': {
    'capacity': 15,
    'weights': [4, 5, 6],
    'profits': [1, 2, 3]
  },
  'output': 6
}

test5 = {
  'input': {
    'capacity': 15,
    'weights': [4, 5, 1, 3, 2, 5],
    'profits': [2, 3, 1, 5, 4, 7]
  },
  'output': 15
}
```

```
'output': 19
}
```

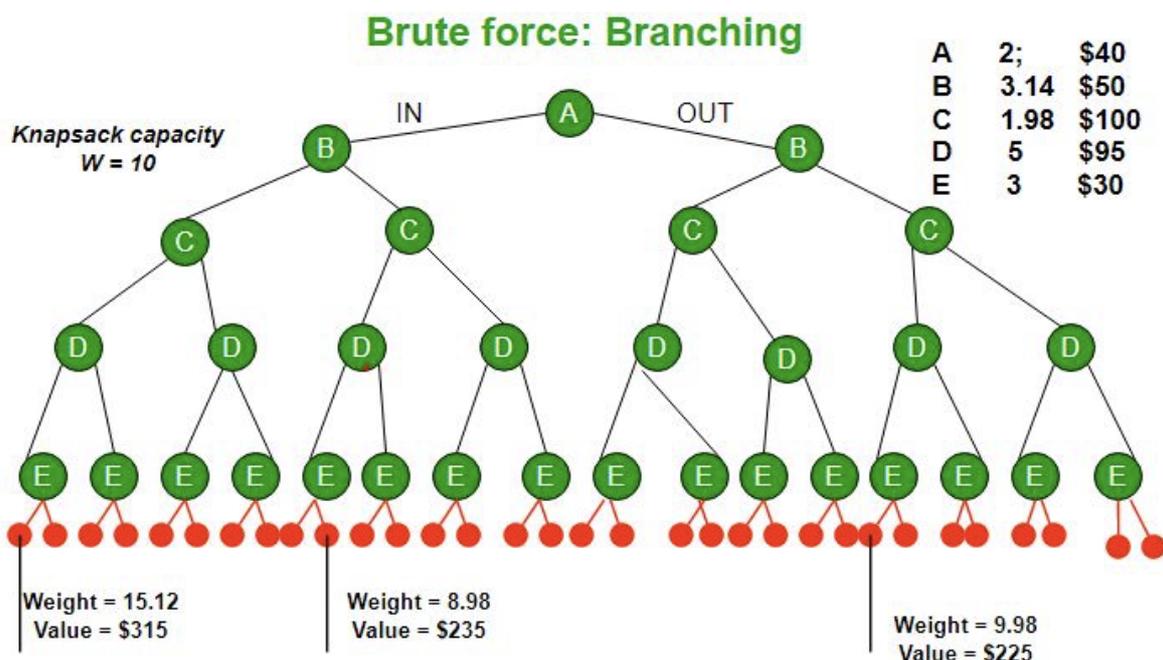
```
tests = [test0, test1, test2, test3, test4, test5]
```

Recursion

Profit [2, 3, 1, 5, 4, 7]
Weight [4, 5, 1, 3, 2, 5]
Capacity 15

1. We'll write a recursive function that computes `max_profit(weights[idx:], profits[idx:], capacity)`, with `idx` starting from 0.
2. If `weights[idx] > capacity`, the current element is cannot be selected, so the maximum profit is the same as `max_profit(weights[idx+1:], profits[idx+1:], capacity)`.
3. Otherwise, there are two possibilities: we either pick `weights[idx]` or don't. We can recursively compute the maximum
 - A. If we don't pick `weights[idx]`, once again the maximum profit for this case is `max_profit(weights[idx+1:], profits[idx+1:], capacity)`
 - B. If we pick `weights[idx]`, the maximum profit for this case is `profits[idx] + max_profit(weights[idx+1:], profits[idx+1:], capacity - weights[idx])`
4. If `weights[idx:]` is empty, the maximum profit for this case is 0.

Here's a visualization of the recursion tree:



Verify that the time complexity of the recursive algorithm is $O(2^N)$

```

def max_profits_recursive(weights, profits, capacity, index = 0):

    # If our index = weights, we are at the end of our options
    if index == len(weights):
        return 0

    # If the weight of the current index is too great, more than the capacity
    # available, then we recursively consider the next option in line.
    elif weights[index] > capacity:
        return max_profits_recursive(weights, profits, capacity, index+1)

    # 2 choices:
    else:
        # Although it can within capacity, we do not take it, because it is not
        # a part of the optimal solution, so perform same operation as above
        path1 = max_profits_recursive(weights, profits, capacity, index+1)

        # Add the element to our "bag", take profit of the index item, call
        # function again with weights, profits, and lowered capacity, and
        # index moves forward one spot.
        path2 = profits[index] + max_profits_recursive(weights, profits,
                                                    capacity - weights[index],
                                                    index + 1)

    return max(path1, path2)

```

```

evaluate_test_cases(max_profits_recursive, tests)

```

TEST CASE #0

Input:

```

{'capacity': 165, 'weights': [23, 31, 29, 44, 53, 38, 63, 85, 89, 82], 'profits': [92,
57, 49, 68, 6...

```

Expected Output:

309

Actual Output:

309

Execution Time:

0.163 ms

Test Result:

PASSED

TEST CASE #1

Input:

```
{'capacity': 3, 'weights': [4, 5, 6], 'profits': [1, 2, 3]}
```

Expected Output:

0

Actual Output:

0

Execution Time:

0.006 ms

Test Result:

PASSED

TEST CASE #2

Input:

```
{'capacity': 4, 'weights': [4, 5, 1], 'profits': [1, 2, 3]}
```

Expected Output:

3

Actual Output:

3

Execution Time:

0.008 ms

Test Result:

PASSED

TEST CASE #3

Input:

```
{'capacity': 170, 'weights': [41, 50, 49, 59, 55, 57, 60], 'profits': [442, 525, 511, 593, 546, 564, ...]}
```

Expected Output:

1735

Actual Output:

1735

Execution Time:

0.069 ms

Test Result:

PASSED

TEST CASE #4

Input:

```
{'capacity': 15, 'weights': [4, 5, 6], 'profits': [1, 2, 3]}
```

Expected Output:

6

Actual Output:

6

Execution Time:

0.01 ms

Test Result:

PASSED

TEST CASE #5

Input:

```
{'capacity': 15, 'weights': [4, 5, 1, 3, 2, 5], 'profits': [2, 3, 1, 5, 4, 7]}
```

Expected Output:

19

Actual Output:

19

Execution Time:

0.055 ms

Test Result:

PASSED

SUMMARY

TOTAL: 6, PASSED: 6, FAILED: 0

```
[(309, True, 0.163),  
(0, True, 0.006),  
(3, True, 0.008),  
(1735, True, 0.069),  
(6, True, 0.01),  
(19, True, 0.055)]
```

Dynamic Programming

1. Create a table of size $(n+1) * (capacity+1)$ consisting of all 0s, where n is the number of elements. $table[i][c]$ represents the maximum profit that can be obtained using the first i elements if the maximum capacity is c . Here's a visual representation of a filled table (source - geeksforgeeks):

	0	1	2	3	4	5	6	7	8	9	10
2(1)	0	0	1	1	1	1	1	1	1	1	1
3(2)	0	0	1	2	2	3	3	3	3	3	3
3(5)	0	0	1	5	5	6	7	7	8	8	8
4(9)	0	0	1	5	9	9	10	14	14	15	16
6(4)	0	0	1	5	9	9	10	14	14	15	16

(The 0th row will contain all zeros and is not shown above.)

2. We'll fill the table row by row and column by column. `table[i][c]` can be filled using some values in the row above it.
3. If `weights[i] > c` i.e. if the current element can is larger than capacity, then `table[i+1][c]` is simply equal to `table[i][c]` (since there's no way we can pick this element).
4. If `weights[i] <= c` then we have two choices: to either pick the current element or not to get the value of `table[i+1][c]`. We can compare the maximum profit for both these options and pick the better one as the value of `table[i][c]`.
 - A. If we don't pick the element with weight `weights[i]`, then once again the maximum profit is `table[i][c]`
 - B. If we pick the element with weight `weights[i]`, then the maximum profit is `profits[i] + table[i][c-weights[i]]`, since we have used up some capacity.

Verify that the complexity of the dynamic programming solution is $O(N * W)$.

```
n, capacity = 5, 10
[[0 for x in range(capacity+1)] for x in range(n+1)]
```

```
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

```
def max_profits_dynamic(weights, profits, capacity):

    length = len(weights)

    # Create a table of size (n+1) * (capacity+1) consisting of all 0s, where
    # is n is the number of elements. table[i][c] represents the maximum profit
    # that can be obtained using the first i elements if the maximum capacity is c.
    table = [[0 for x in range(capacity+1)] for x in range(length+1)]

    # Loop through the rows of weights
    for index1 in range(length):

        # Loop through the columns considering capacity, from col-1, since first is
        # zeroed out column
        for index2 in range(1, capacity+1):

            # if the weight of the current index is greater than the capacity left
            if weights[index1] > index2:
                # the current index is copied from the item directly above it
                table[index1+1][index2] = table[index1][index2]

            # if the weight is NOT too much, then we have two options
            else:
                # we choose the higher of the two, either do not use the current element
```

```
# or we do and we also take the profits of index and subtract from our  
# capacity  
table[index1+1][index2] = max(table[index1][index2], profits[index1] +  
                               table[index1][index2 - weights[index1]])  
  
# return the bottom right element of the table, which will be the highest profit  
return table[-1][-1]
```

```
evaluate_test_cases(max_profits_dynamic, tests)
```

TEST CASE #0

Input:

```
{'capacity': 165, 'weights': [23, 31, 29, 44, 53, 38, 63, 85, 89, 82], 'profits': [92,  
57, 49, 68, 6...
```

Expected Output:

309

Actual Output:

309

Execution Time:

0.857 ms

Test Result:

PASSED

TEST CASE #1

Input:

```
{'capacity': 3, 'weights': [4, 5, 6], 'profits': [1, 2, 3]}
```

Expected Output:

0

Actual Output:

0

Execution Time:

0.014 ms

Test Result:

PASSED

TEST CASE #2

Input:

```
{'capacity': 4, 'weights': [4, 5, 1], 'profits': [1, 2, 3]}
```

Expected Output:

3

Actual Output:

3

Execution Time:

0.014 ms

Test Result:

PASSED

TEST CASE #3

Input:

```
{'capacity': 170, 'weights': [41, 50, 49, 59, 55, 57, 60], 'profits': [442, 525, 511, 593, 546, 564, ...]}
```

Expected Output:

1735

Actual Output:

1735

Execution Time:

0.624 ms

Test Result:

PASSED

TEST CASE #4

Input:

```
{'capacity': 15, 'weights': [4, 5, 6], 'profits': [1, 2, 3]}
```

Expected Output:

6

Actual Output:

6

Execution Time:

0.037 ms

Test Result:

PASSED

TEST CASE #5

Input:

```
{'capacity': 15, 'weights': [4, 5, 1, 3, 2, 5], 'profits': [2, 3, 1, 5, 4, 7]}
```

Expected Output:

19

Actual Output:

19

Execution Time:

0.063 ms

Test Result:

PASSED

SUMMARY

TOTAL: 6, PASSED: 6, FAILED: 0

```
[(309, True, 0.857),
 (0, True, 0.014),
 (3, True, 0.014),
 (1735, True, 0.624),
 (6, True, 0.037),
 (19, True, 0.063)]
```

DEFAULT SOLUTIONS

```
def lcq_recursive(seq1, seq2, idx1=0, idx2=0):
    # Check if either of the sequences is exhausted
    if idx1 == len(seq1) or idx2 == len(seq2):
        return 0

    # Check if the current characters are equal
    if seq1[idx1] == seq2[idx2]:
        return 1 + lcq_recursive(seq1, seq2, idx1+1, idx2+1)
    # Skip one element from each sequence
    else:
        return max(lcq_recursive(seq1, seq2, idx1+1, idx2),
                   lcq_recursive(seq1, seq2, idx1, idx2+1))
```

```
from jovian.pythondsa import evaluate_test_cases
```

```
evaluate_test_cases(lcq_recursive, lcq_tests)
```

```
-----
NameError                                Traceback (most recent call last)
/tmp/ipykernel_38/865804860.py in <module>
----> 1 evaluate_test_cases(lcq_recursive, lcq_tests)
```

NameError: name 'lcq_tests' is not defined

```
%%time
lcq_recursive('seredipitous', 'precipitation')
```

```
%%time
lcq_recursive('Asdfsfafssess', 'oypiououuiuo')
```

```
def lcq_memoized(seq1, seq2):
    memo = {}

    def recurse(idx1, idx2):
        key = idx1, idx2

        if key in memo:
            return memo[key]
```

```

if idx1 == len(seq1) or idx2 == len(seq2):
    memo[key] = 0
elif seq1[idx1] == seq2[idx2]:
    memo[key] = 1 + recurse(idx1+1, idx2+1)
else:
    memo[key] = max(recurse(idx1+1, idx2),
                   recurse(idx1, idx2+1))
return memo[key]

return recurse(0, 0)

```

```
evaluate_test_cases(lcq_memoized, lcq_tests)
```

```
%%time
lcq_memoized('Asdfsfafssess', 'oypiououuiuo')
```

```
%%time
lcq_memoized('seredipitous', 'precipitation')
```

```
%%time
lcq_memoized('longest', 'stone')
```

Dynamic programming:

```

def lcq_dp(seq1, seq2):
    n1, n2 = len(seq1), len(seq2)
    results = [[0 for _ in range(n2+1)] for _ in range(n1+1)]
    for idx1 in range(n1):
        for idx2 in range(n2):
            if seq1[idx1] == seq2[idx2]:
                results[idx1+1][idx2+1] = 1 + results[idx1][idx2]
            else:
                results[idx1+1][idx2+1] = max(results[idx1][idx2+1], results[idx1+1][idx2])
    return results[-1][-1]

```

```
evaluate_test_cases(lcq_dp, lcq_tests)
```

```
%%time
lcq_dp('Asdfsfafssess', 'oypiououuiuo')
```

```
%%time
lcq_dp('seredipitous', 'precipitation')
```

```
%%time
lcq_dp('longest', 'stone')
```

Knapsack Solutions

```
from jovian.pythondsa import evaluate_test_cases
```

```
def max_profit_recursive(capacity, weights, profits, idx=0):  
    if idx == len(weights):  
        return 0  
    if weights[idx] > capacity:  
        return max_profit_recursive(capacity, weights, profits, idx+1)  
    else:  
        return max(max_profit_recursive(capacity, weights, profits, idx+1),  
                   profits[idx] + max_profit_recursive(capacity-weights[idx], weights,
```

```
evaluate_test_cases(max_profit_recursive, tests)
```

Memoized:

```
def knapsack_memo(capacity, weights, profits):  
    memo = {}  
  
    def recurse(idx, remaining):  
        key = (idx, remaining)  
        if key in memo:  
            return memo[key]  
        elif idx == len(weights):  
            memo[key] = 0  
        elif weights[idx] > remaining:  
            memo[key] = recurse(idx+1, remaining)  
        else:  
            memo[key] = max(recurse(idx+1, remaining),  
                           profits[idx] + recurse(idx+1, remaining-weights[idx]))  
        return memo[key]  
  
    return recurse(0, capacity)
```

```
evaluate_test_cases(knapsack_memo, tests)
```

Dynamic programming:

```
def knapsack_dp(capacity, weights, profits):  
    n = len(weights)  
    results = [[0 for _ in range(capacity+1)] for _ in range(n+1)]  
  
    for idx in range(n):  
        for c in range(capacity+1):  
            if weights[idx] > c:  
                results[idx+1][c] = results[idx][c]  
            else:
```

```
        results[idx+1][c] = max(results[idx][c], profits[idx] + results[idx][c]-  
return results[-1][-1]
```

```
evaluate_test_cases(knapsack_dp, tests)
```

```
import jovian
```

```
jovian.commit(filename="dynamic-programming-problems")
```

Graph Algorithms (BFS, DFS, Shortest Paths) using Python

Part 5 of "Data Structures and Algorithms in Python"

[Data Structures and Algorithms in Python](#) is a beginner-friendly introduction to common data structures (linked lists, stacks, queues, graphs) and algorithms (search, sorting, recursion, dynamic programming) in Python, designed to help you prepare for coding interviews and assessments.

Ask questions, get help & participate in discussions on the [course community forum](#). Earn a verified certificate of accomplishment for this course by signing up here: <http://pythonsa.com>.

How to Run the Code

The best way to learn the material is to execute the code and experiment with it yourself. This tutorial is an executable [Jupyter notebook](#). You can *run* this tutorial and experiment with the code examples in a couple of ways: *using free online resources* (recommended) or *on your computer*.

Option 1: Running using free online resources (1-click, recommended)

The easiest way to start executing the code is to click the **Run** button at the top of this page and select **Run on Binder**. You can also select "Run on Colab" or "Run on Kaggle", but you'll need to create an account on [Google Colab](#) or [Kaggle](#) to use these platforms.

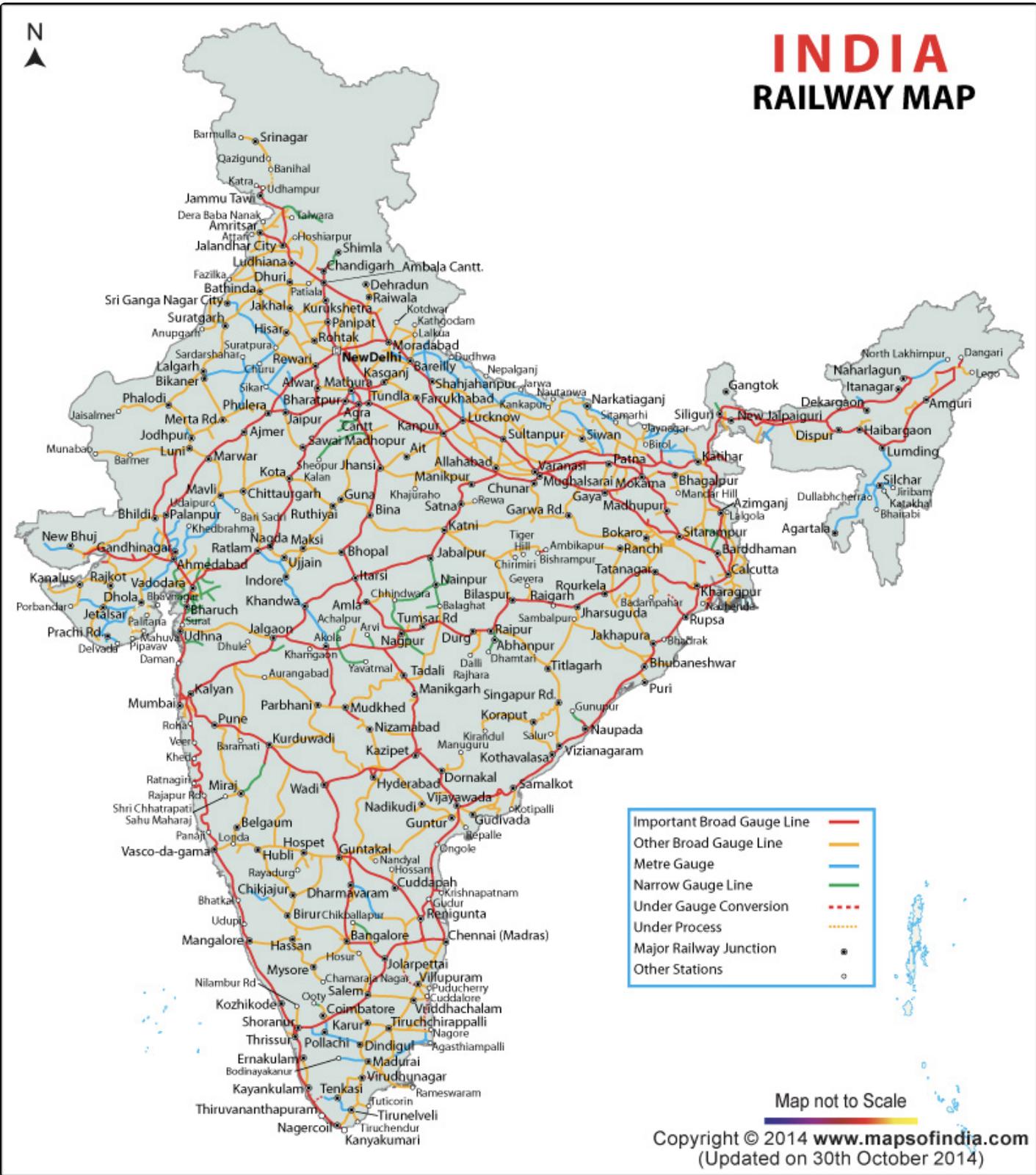
Option 2: Running on your computer locally

To run the code on your computer locally, you'll need to set up [Python](#), download the notebook and install the required libraries. We recommend using the [Conda](#) distribution of Python. Click the **Run** button at the top of this page, select the **Run Locally** option, and follow the instructions.

Graphs in the Real World

Railway network

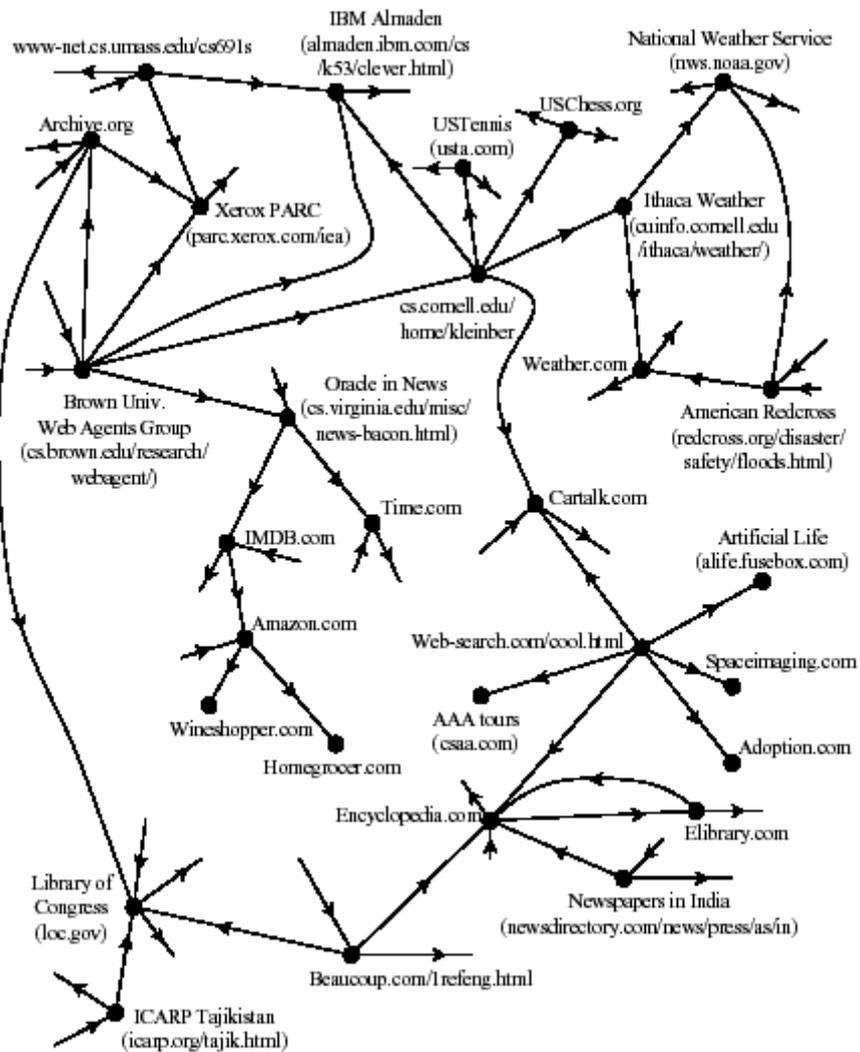
INDIA RAILWAY MAP



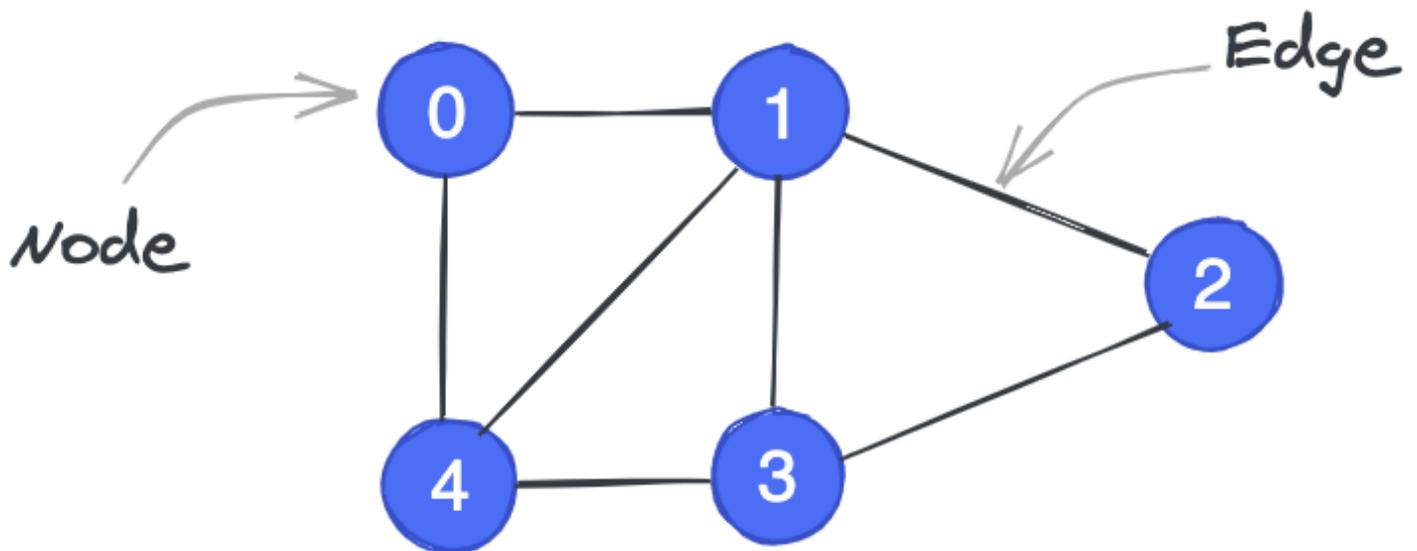
Flight routes



Hyperlinks



Graph Data Structure



*# Simplest way to represent the above node/edge relationships
But not very efficient. Must loop through all to get info
Will be more efficient to store information within nodes, edges*

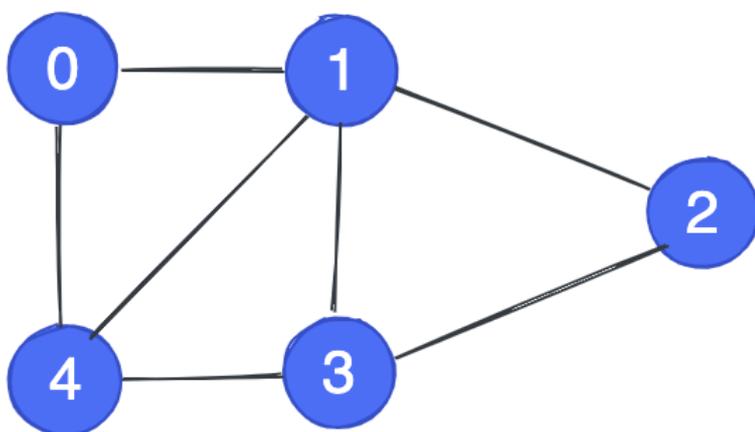
```
number_nodes = 5
```

```
# Represented using a list of bi-directional pairs  
edges = [(0,1), (0,4), (1,4), (1,2), (2,3), (1,3), (3,4)]
```

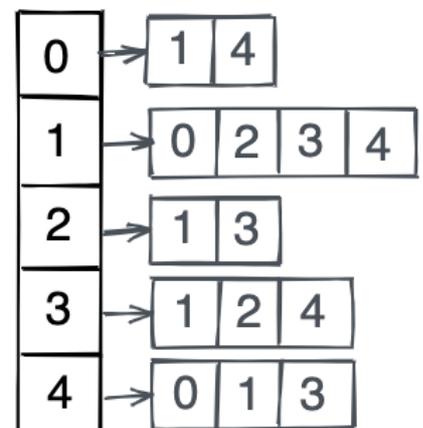
```
number_nodes, len(edges)
```

```
(5, 7)
```

Adjacency Lists



Adjacency List



Question: Create a class to represent a graph as an adjacency list in Python

```
[[],[],[],[],[]] # We will need a list of empty lists for our adjacency list
```

```
[[], [], [], [], []]
```

```
# cannot do this because it will be the same list repeated and will all be the same  
list1 = [[]] * 10  
list1
```

```
[[], [], [], [], [], [], [], [], [], []]
```

```
# Can do this though:  
list2 = [[] for x in range(10)]
```

```
list2[0].append(1)  
list2
```

```
[[1], [], [], [], [], [], [], [], [], []]
```

```
# Our edges as defined above:  
for edge in edges:  
    print(edge)
```

```
(0, 1)
```

```
(0, 4)
```

```
(1, 4)
```

```
(1, 2)
```

```
(2, 3)
```

```
(1, 3)
```

```
(3, 4)
```

```
# Can also isolate each node associated with the edge:  
for node1, node2 in edges:  
    print("node 1: ", node1, " ", "node 2: ", node2)
```

```
node 1: 0    node 2: 1
```

```
node 1: 0    node 2: 4
```

```
node 1: 1    node 2: 4
```

```
node 1: 1    node 2: 2
```

```
node 1: 2    node 2: 3
```

```
node 1: 1    node 2: 3
```

```
node 1: 3    node 2: 4
```

```
class Graph:  
    def __init__(self, number_nodes, edges):  
        self.number_nodes = number_nodes  
        self.data = [[] for x in range(number_nodes)]  
        for node1, node2 in edges:
```

```

    # Set up the graph:
    self.data[node1].append(node2)
    self.data[node2].append(node1)

# So we can make pretty printings of our data
# outputs an object rather than the string form
def __repr__(self):
    # See what enumerate on a list does below
    # for each node and neighbor in the data, creates a string with placeholders
    # set for node and neighbors belonging to it, separated by a colon (as a list)
    return "\n".join("{}: {}".format(node1, neighbors) for node1, neighbors in enumerate(self.data))

# This is called when we print(), str(), insert() for graph1, etc.
def __str__(self):
    return self.__repr__()

# Question: Write a function to add an edge to a graph represented as an adjacency list
def add_edge(self, node1, node2):
    self.data[node1].append(node2)
    self.data[node2].append(node1)

# Question: Write a function to remove an edge from a graph represented as an adjacency list
def remove_edge(self, node1, node2):
    self.data[node1].remove(node2)
    self.data[node2].remove(node1)

```

```

enumerate([4, 5, 6, 7, 8]) # Creates an iterable object

```

```

<enumerate at 0x7f5eb0689b00>

```

```

# So we can iterate over it and get the index and the value:
for x in enumerate([4, 5, 6, 7, 8]):
    print(x)

```

```

(0, 4)
(1, 5)
(2, 6)
(3, 7)
(4, 8)

```

```

graph1 = Graph(number_nodes, edges)

```

```
graph1.data
```

```
[[1, 4], [0, 4, 2, 3], [1, 3], [2, 1, 4], [0, 1, 3]]
```

```
for x in enumerate(graph1.data):  
    print(x)
```

```
(0, [1, 4])  
(1, [0, 4, 2, 3])  
(2, [1, 3])  
(3, [2, 1, 4])  
(4, [0, 1, 3])
```

```
# for each node and neighbor in the data, creates a string with placeholders  
# set for node and neighbors belonging to it, separated by a colon (as a list)
```

```
["{}:{}".format(node1, neighbors) for node1, neighbors in enumerate(graph1.data)]
```

```
['0:[1, 4]', '1:[0, 4, 2, 3]', '2:[1, 3]', '3:[2, 1, 4]', '4:[0, 1, 3]']
```

```
# To join the above strings together as one string representing the adjacency list  
# we have put this into the __repr__
```

```
"\n".join(["{}:{}".format(node1, neighbors) for node1, neighbors in enumerate(graph1.data)])
```

```
'0:[1, 4]\n1:[0, 4, 2, 3]\n2:[1, 3]\n3:[2, 1, 4]\n4:[0, 1, 3]'
```

```
print(graph1)
```

```
0: [1, 4]  
1: [0, 4, 2, 3]  
2: [1, 3]  
3: [2, 1, 4]  
4: [0, 1, 3]
```

Question: Write a function to add an edge to a graph represented as an adjacency list.

Question: Write a function to remove an edge from a graph represented as a adjacency list.

```
graph1.add_edge(3, 0)
```

```
for x in enumerate(graph1.data):  
    print(x)
```

```
(0, [1, 4, 3])
```

```
(1, [0, 4, 2, 3])
(2, [1, 3])
(3, [2, 1, 4, 0])
(4, [0, 1, 3])
```

```
graph1.remove_edge(3, 0)
```

```
for x in enumerate(graph1.data):
    print(x)
```

```
(0, [1, 4])
(1, [0, 4, 2, 3])
(2, [1, 3])
(3, [2, 1, 4])
(4, [0, 1, 3])
```

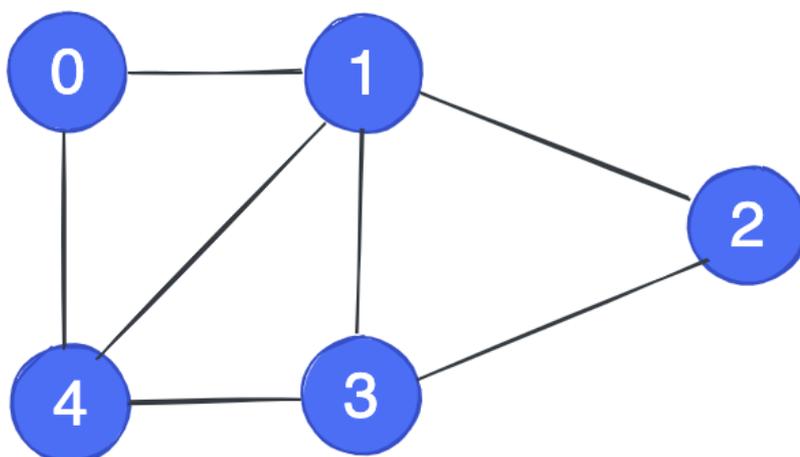
```
!pip install jovian --upgrade --quiet
```

```
import jovian
```

```
jovian.commit()
```

```
[jovian] Updating notebook "evanmarie/python-graph-algorithms" on https://jovian.ai
[jovian] Committed successfully! https://jovian.ai/evanmarie/python-graph-algorithms
'https://jovian.ai/evanmarie/python-graph-algorithms'
```

Adjacency Matrix



Adjacency Matrix

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Question: Represent a graph as an adjacency matrix in Python

```
nodes = 5
```

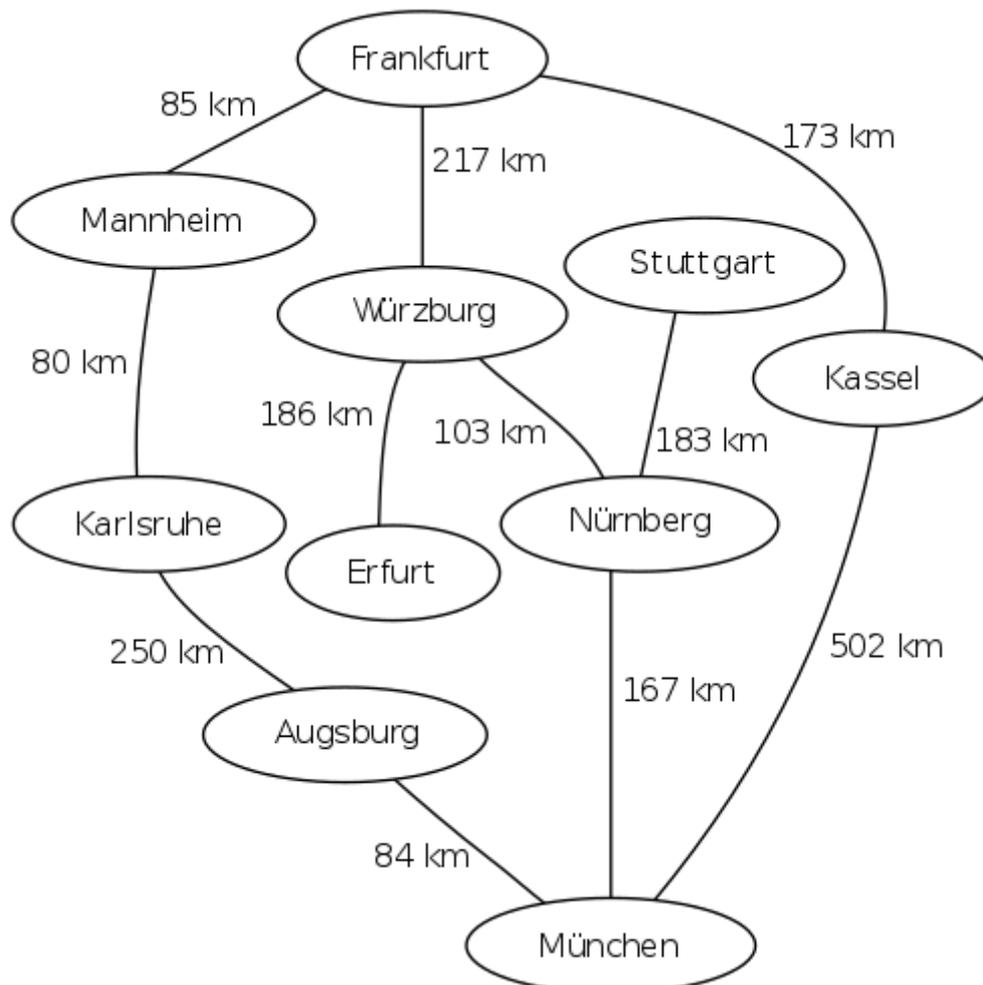
```
graph2 = [[ [0] for x in range(number_nodes) ] for x in range(number_nodes) ]  
graph2
```

```
[[ [0], [0], [0], [0], [0] ],  
 [ [0], [0], [0], [0], [0] ],  
 [ [0], [0], [0], [0], [0] ],  
 [ [0], [0], [0], [0], [0] ],  
 [ [0], [0], [0], [0], [0] ] ]
```

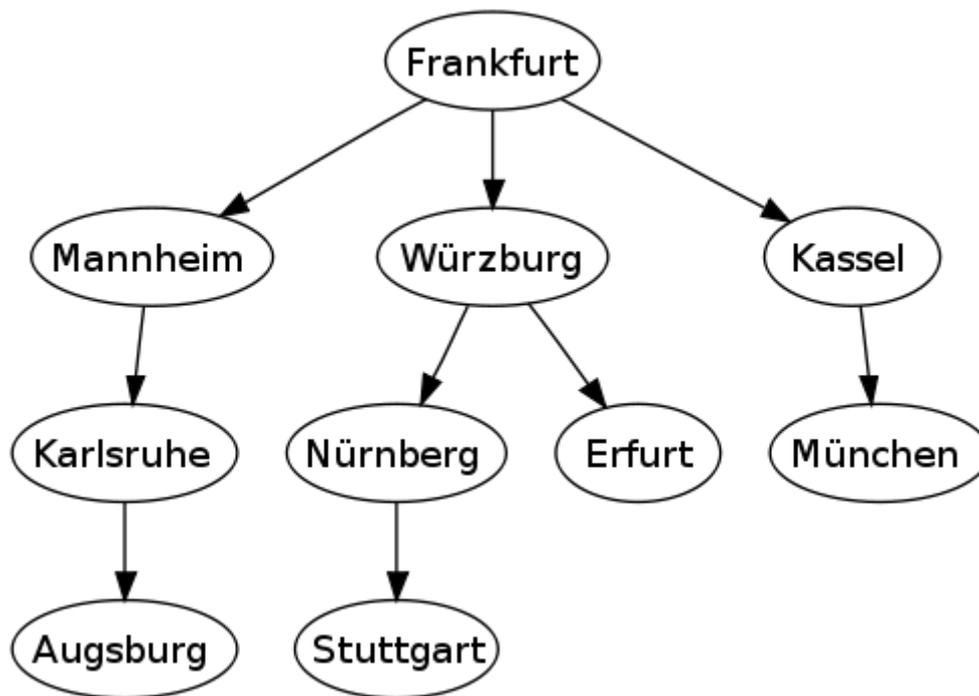
Graph Traversal

Breadth-First Search

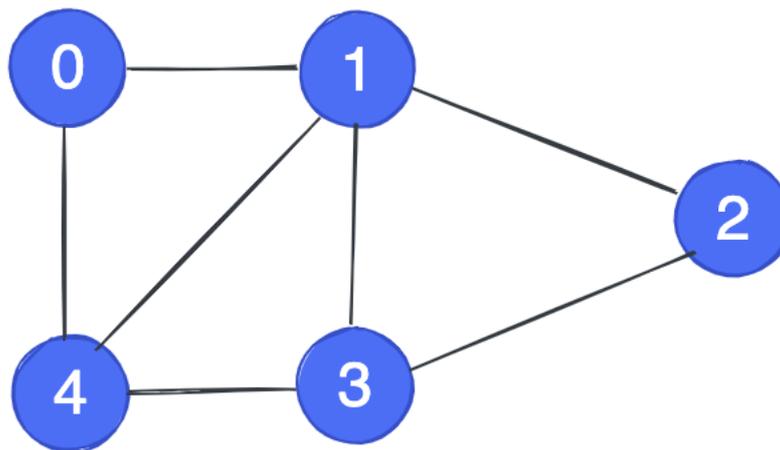
A real-world graph:



Breadth-first search tree (starting from Frankfurt):



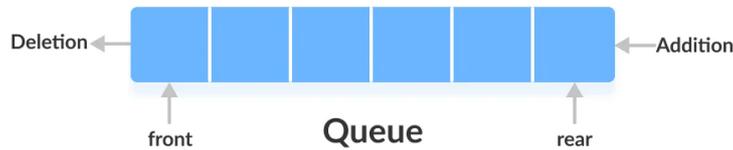
Question: Implement breadth-first search given a source node in a graph using Python.



BFS pseudocode (Wikipedia):

```

1  procedure BFS(G, root) is
2    let Q be a queue
3    label root as discovered
4    Q.enqueue(root)
5    while Q is not empty do
6      v := Q.dequeue()
7      if v is the goal then
8        return v
9      for all edges from v to w in G.adjacentEdges(v) do
10       if w is not labeled as discovered then
11         label w as discovered
12         Q.enqueue(w)
  
```



```

def breadth_first(graph, root):

    # create queue list
    queue = []
    # start with all nodes as a list marked undiscovered for the entirety
    # of the graph
    discovered = [False] * len(graph.data)

    # track distance (number of edges) for each node
    distance = [None] * len(graph.data)

    # dictionary, parent, keeps track of which nodes caused others to
    # be discovered (allows for easy backtracking as well)
    parent = [None] * len(graph.data)

    # mark the root node as discovered and add to queue
    discovered[root] = True
    queue.append(root)
    # in the beginning, starting with root, no edges are known
    distance[root] = 0
    # Set up index to track the first available item in the queue, FIFO
    index = 0

    while index < len(queue):
        # acquire the first-in element, which we will call current, dequeue
        current = queue[index]
        # update the index
        index += 1

        # check edges of current node (contained in self.data for current)
        for node in graph.data[current]:
            # if this node has not yet been discovered (is False in discovered list)
            if not discovered[node]:
                # the distance for this node is 1 more than that of the current
                # node which caused it to be discovered
                distance[node] = 1 + distance[current]
                # setting parent as node which caused this one to be discovered
                parent[node] = current
                # mark as discovered and add to the queue
                discovered[node] = True
                queue.append(node)

    return queue, distance, parent

```

```
breadth_first(graph1, 3)
```

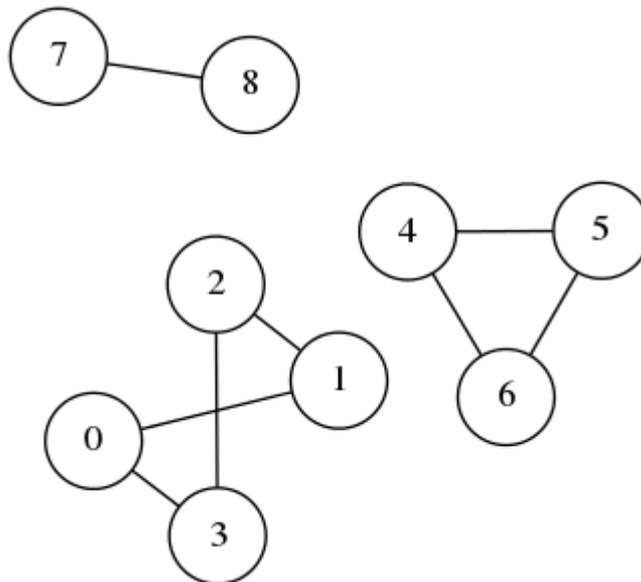
```
([3, 2, 1, 4, 0], [2, 1, 1, 0, 1], [1, 3, 3, None, 3])
```

```
import jovian
```

```
jovian.commit()
```

```
[jovian] Updating notebook "evanmarie/python-graph-algorithms" on https://jovian.ai  
[jovian] Committed successfully! https://jovian.ai/evanmarie/python-graph-algorithms  
'https://jovian.ai/evanmarie/python-graph-algorithms'
```

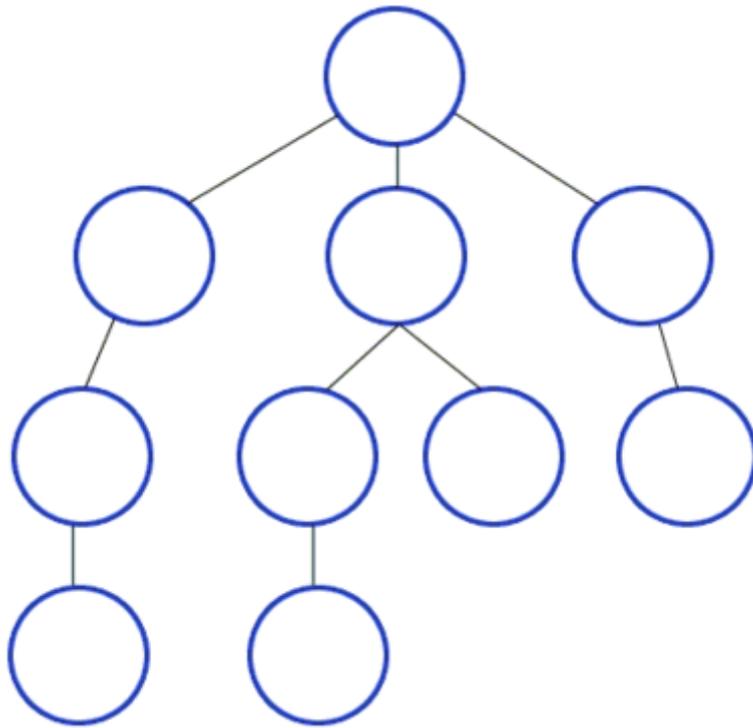
Question: Write a program to check if all the nodes in a graph are connected



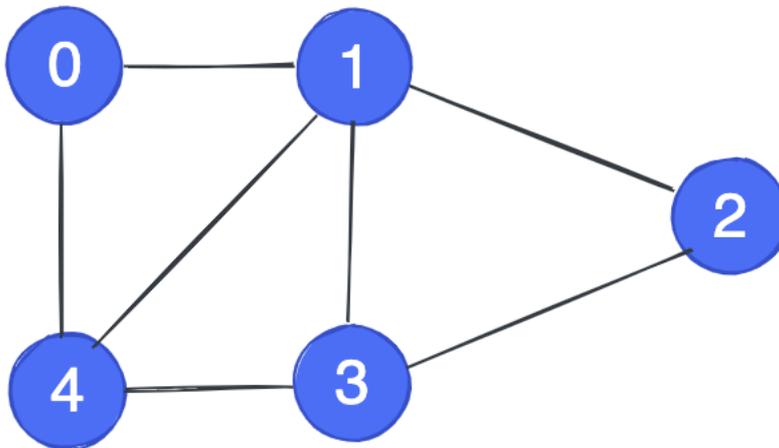
```
num_nodes3 = 9  
edges3 = [(0, 1), (0, 3), (1, 2), (2, 3), (4, 5), (4, 6), (5, 6), (7, 8)]  
num_nodes3, len(edges3)
```

```
(9, 8)
```

Depth-first search



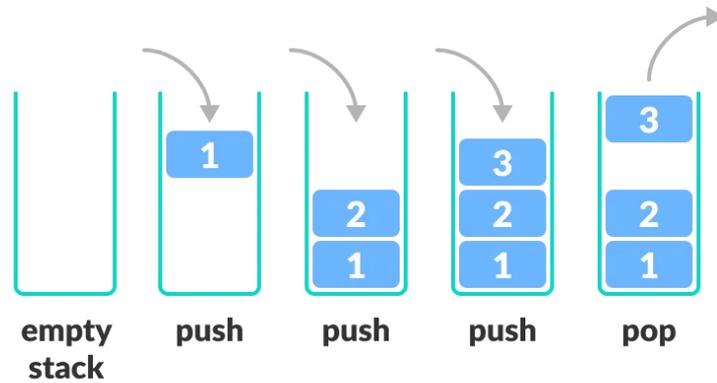
Question: Implement depth first search from a given node in a graph using Python.



DFS pseudocode (Wikipedia):

```

procedure DFS_iterative(G, v) is
  let S be a stack
  S.push(v)
  while S is not empty do
    v = S.pop()
    if v is not labeled as discovered then
      label v as discovered
      for all edges from v to w in G.adjacentEdges(v) do
        S.push(w)
  
```



```
def depth_first(graph, root):

    # create stack
    stack = []

    # the entirety of the graph starts out undiscovered
    discovered = [False] * len(graph.data)

    # track parents of nodes
    parent = [None] * len(graph.data)

    # result list = store of the results of nodes that have been popped
    results = []

    # add root to the stack
    stack.append(root)
    # do not mark discovered until removing from stack

    # if there is anything in the stack
    while len(stack) > 0:

        # the current node is the last node of the stack
        # which we pop out
        current = stack.pop()

        # will get duplicate values in result if we do not
        # check now whether or not the current has been discovered
        if not discovered[current]:

            # mark current as discovered
            discovered[current] = True

            # add current to the result list
            results.append(current)

            # loop adding all nodes to stack if they have not been discovered
            for node in graph.data[current]:
```

```
if not discovered[node]:  
    # setting parent as node which cased this one to be discovered  
    parent[node] = current  
    stack.append(node)
```

```
return results
```

```
depth_first(graph1, 3)
```

```
[3, 4, 1, 2, 0]
```

```
graph1
```

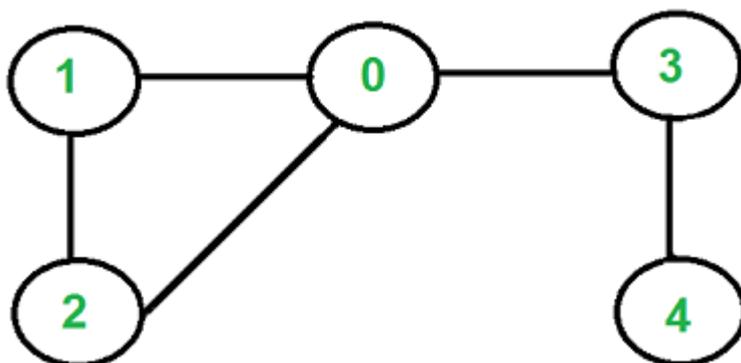
```
0: [1, 4]  
1: [0, 4, 2, 3]  
2: [1, 3]  
3: [2, 1, 4]  
4: [0, 1, 3]
```

```
import jovian
```

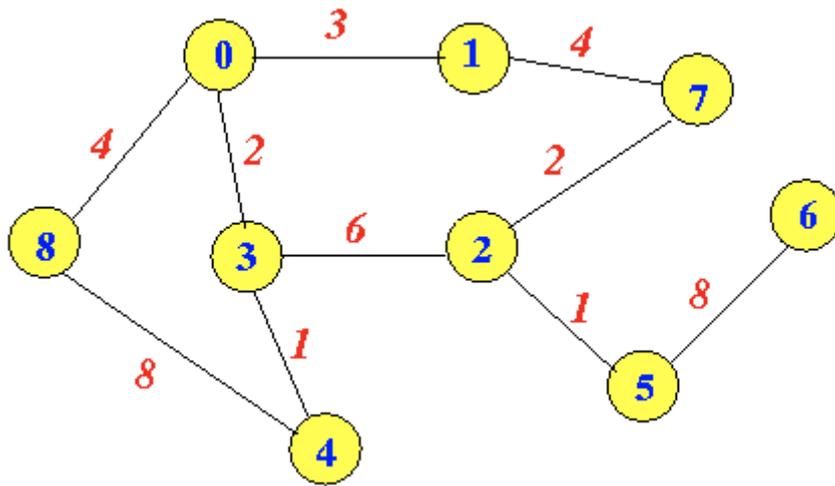
```
jovian.commit()
```

```
[jovian] Updating notebook "evanmarie/python-graph-algorithms" on https://jovian.ai  
[jovian] Committed successfully! https://jovian.ai/evanmarie/python-graph-algorithms  
'https://jovian.ai/evanmarie/python-graph-algorithms'
```

Question: Write a function to detect a cycle in a graph



Weighted Graphs



Graph with weights

```
num_nodes5 = 9
```

```
edges5 = [(0, 1, 3), (0, 3, 2), (0, 8, 4), (1, 7, 4), (2, 7, 2), (2, 3, 6),
          (2, 5, 1), (3, 4, 1), (4, 8, 8), (5, 6, 8)]
```

```
graph2 = Graph(number_nodes, edges)
```

```
graph2
```

```
0: [1, 4]
```

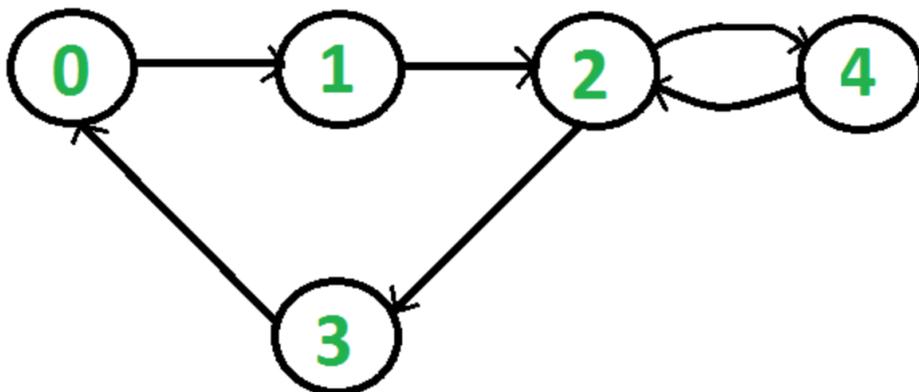
```
1: [0, 4, 2, 3]
```

```
2: [1, 3]
```

```
3: [2, 1, 4]
```

```
4: [0, 1, 3]
```

Directed Graphs



```
num_nodes6 = 5
```

```
edges6 = [(0, 1), (1, 2), (2, 3), (2, 4), (4, 2), (3, 0)]
```

```
num_nodes6, len(edges6)
```

```
(5, 6)
```

Question: Define a class to represent weighted and directed graphs in Python.

```

class Graph2:
    def __init__(self, number_nodes, edges, directed=False, weighted=False):
        self.number_nodes = number_nodes
        self.directed = directed
        self.weighted = weighted
        self.data = [[] for x in range(number_nodes)]

        # for each corresponding element in the adjacency list
        # the following will store the weight for the edges.
        self.weight = [[] for x in range(number_nodes)]
        for edge in edges:
            if self.weighted:
                # include weights
                node1, node2, weight = edge
                self.data[node1].append(node2)

                # The following stores the weight of the edge from node1
                # to node2
                self.weight[node1].append(weight)

                # We may want to store the other direction, if bi-directional
                # so we will store the reciprocal direction data
                if not directed:
                    self.data[node2].append(node1)
                    self.weight[node2].append(weight)

            else:
                # work without weights
                node1, node2 = edge
                self.data[node1].append(node2)
                if not directed:
                    self.data[node2].append(node1)

    def __repr__(self):
        result = ""
        if self.weighted:
            for i, (nodes, weights) in enumerate(zip(self.data, self.weight)):
                result += "{}: {}\n".format(i, list(zip(nodes, weights)))
        else:
            for i, nodes in enumerate(self.data):
                result += "{}: {}\n".format(i, nodes)
        return result

```

graph1

```

0: [1, 4]
1: [0, 4, 2, 3]
2: [1, 3]
3: [2, 1, 4]
4: [0, 1, 3]

```

```
graph1 = Graph2(number_nodes, edges)
graph1
```

```
0: [1, 4]
1: [0, 4, 2, 3]
2: [1, 3]
3: [2, 1, 4]
4: [0, 1, 3]
```

```
# Graph with weights
```

```
num_nodes2 = 9
edges2 = [(0, 1, 3), (0, 3, 2), (0, 8, 4), (1, 7, 4), (2, 7, 2), (2, 3, 6),
          (2, 5, 1), (3, 4, 1), (4, 8, 8), (5, 6, 8)]
```

```
graph3 = Graph2(num_nodes2, edges2, weighted = True)
graph3
```

```
0: [(1, 3), (3, 2), (8, 4)]
1: [(0, 3), (7, 4)]
2: [(7, 2), (3, 6), (5, 1)]
3: [(0, 2), (2, 6), (4, 1)]
4: [(3, 1), (8, 8)]
5: [(2, 1), (6, 8)]
6: [(5, 8)]
7: [(1, 4), (2, 2)]
8: [(0, 4), (4, 8)]
```

```
num_nodes3 = 5
edges3 = [(0, 1), (1, 2), (2, 3), (2, 4), (4, 2), (3, 0)]
```

```
graph4 = Graph2(num_nodes3, edges3, directed = True)
graph4
```

```
0: [1]
1: [2]
2: [3, 4]
3: [0]
4: [2]
```

```
# Aakash's Version:
```

```
class Graph:
    def __init__(self, num_nodes, edges, directed=False):
        self.data = [[] for _ in range(num_nodes)]
        self.weight = [[] for _ in range(num_nodes)]

        self.directed = directed
        self.weighted = len(edges) > 0 and len(edges[0]) == 3
```

```

for e in edges:
    self.data[e[0]].append(e[1])
    if self.weighted:
        self.weight[e[0]].append(e[2])

    if not directed:
        self.data[e[1]].append(e[0])
        if self.weighted:
            self.data[e[1]].append(e[2])

def __repr__(self):
    result = ""
    for i in range(len(self.data)):
        pairs = list(zip(self.data[i], self.weight[i]))
        result += "{}: {}\n".format(i, pairs)
    return result

def __str__(self):
    return repr(self)

```

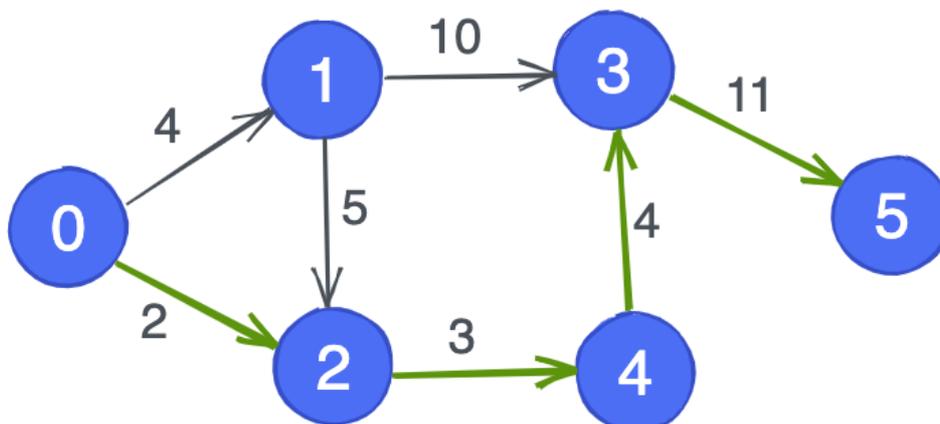
```
import jovian
```

```
jovian.commit()
```

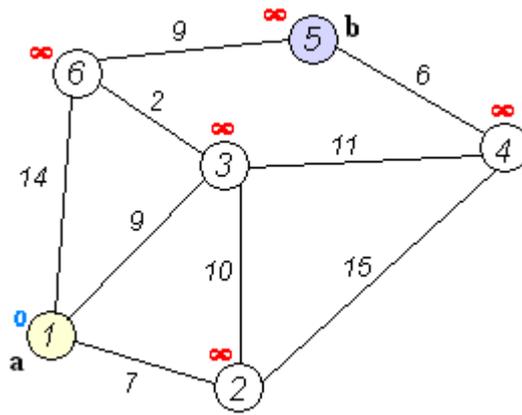
[jovian] Updating notebook "evanmarie/python-graph-algorithms" on <https://jovian.ai>
[jovian] Committed successfully! <https://jovian.ai/evanmarie/python-graph-algorithms>
'<https://jovian.ai/evanmarie/python-graph-algorithms>'

Shortest Paths

Question: Write a function to find the length of the shortest path between two nodes in a weighted directed graph.



Dijkstra's algorithm (Wikipedia):



1. Mark all nodes unvisited. Create a set of all the unvisited nodes called the unvisited set.
2. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes. Set the initial node as current.[16]
3. For the current node, consider all of its unvisited neighbours and calculate their tentative distances through the current node. Compare the newly calculated tentative distance to the current assigned value and assign the smaller one. For example, if the current node A is marked with a distance of 6, and the edge connecting it with a neighbour B has length 2, then the distance to B through A will be $6 + 2 = 8$. If B was previously marked with a distance greater than 8 then change it to 8. Otherwise, the current value will be kept.
4. When we are done considering all of the unvisited neighbours of the current node, mark the current node as visited and remove it from the unvisited set. A visited node will never be checked again.
5. If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the unvisited set is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes), then stop. The algorithm has finished.
6. Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new "current node", and go back to step 3.

```
def shortest_path(graph, start, goal):

    # Set all nodes as unvisited to start
    visited = [False] * len(graph.data)
    # Keep track of parents for backtracking and tracing paths
    parent = [None] * len(graph.data)
    # Set all distances to infinity to start
    distance = [float('inf')] * len(graph.data)
    # Create the list that will serve as our queue of all nodes in the order
    # of their distance from start/root
    queue = []

    # Start with the distance from start to itself as 0
    distance[start] = 0
    # Add the root/start node to our queue
    queue.append(start)
    # What is the next element we need to dequeue
    index = 0

    # While the index is less than our queue and we have not
```

```

# found and marked our target as visited
while index < len(queue) and not visited[goal]:
    # Get an element to serve as "current" from the queue
    current = queue[index]
    # Mark current as visited
    visited[current] = True
    # Increment our index counter
    index += 1

    # Update distances of all neighbors (using helper function below)
    update_distances(graph, current, distance, parent)

    # Find the first unvisited node with the smallest distance (helper)
    next_node = pick_next_node(distance, visited)
    # If there is a next node and have not visited all that there is to
    # visit, append the best next node to the queue
    if next_node:
        queue.append(next_node)

    # Mark this node as visited
    visited[current] = True

return distance[goal], parent

```

```

def update_distances(graph, current, distance, parent=None):
    """Update the distances of the current node's neighbors"""

    # Get neighbors of current node using graph.data
    neighbors = graph.data[current]
    # Get the weights of the edges connecting current to its
    # neighbors
    weights = graph.weight[current]

    # Go through each of the neighbors
    for i, node in enumerate(neighbors):
        # Acquire weight, so now we have node and its weight
        # for all neighbors
        weight = weights[i]

        # Distance checking for shortest (i.e. lowest weight)
        # If distance from start to current plus weight for the
        # distance from current to next node is less than the distance
        # of a neighbor (which is infinity to start),
        if distance[current] + weight < distance[node]:
            # Update the distance between node and start/root
            # (no updating if there is already a distance and it is
            # shorter than our current distance from start.)
            distance[node] = distance[current] + weight

```

```

# By which node did we arrive here and why are we updating?
if parent:
    parent[node] = current

```

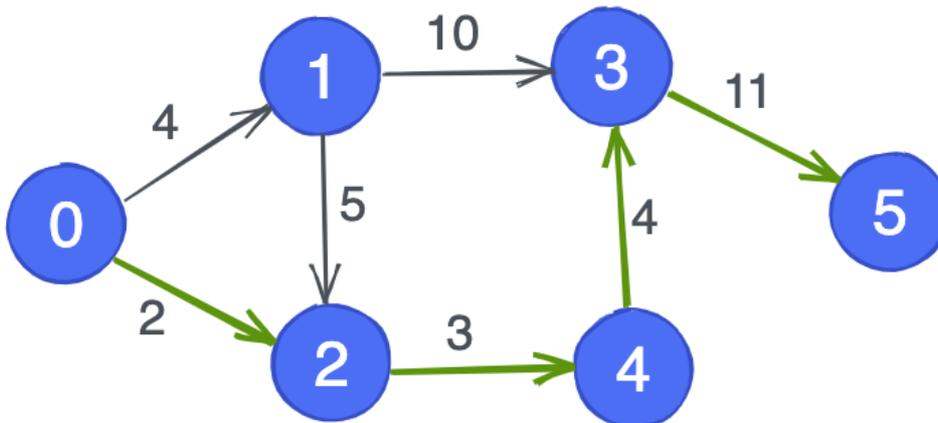
```

def pick_next_node(distance, visited):
    """Pick the next unvisited node by which has the shortest distance/weight"""

    # Tracking the minimum distance
    min_distance = float('inf')
    # Node with the minimum distance is set to none
    min_node = None

    # Loop through nodes and check:
    for node in range(len(distance)):
        # If it has not been visited and the distance from it back
        # to start/root is less than current min,
        if not visited[node] and distance[node] < min_distance:
            # update minimum to this node
            min_node = node
            # and update the minimum distance currently.
            min_distance = distance[node]
    return min_node

```



```

# Graph representing the directional node map above
num_nodes7 = 6
edges7 = [(0, 1, 4), (0, 2, 2), (1, 2, 5), (1, 3, 10), (2, 4, 3), (4, 3, 4), (3, 5, 11)]
num_nodes7, len(edges7)

```

```
(6, 7)
```

```

# New weighted and directed graph to try out shortest_path
graph7 = Graph2(num_nodes7, edges7, weighted = True, directed = True)
graph7

```

```

0: [(1, 4), (2, 2)]
1: [(2, 5), (3, 10)]
2: [(4, 3)]
3: [(5, 11)]

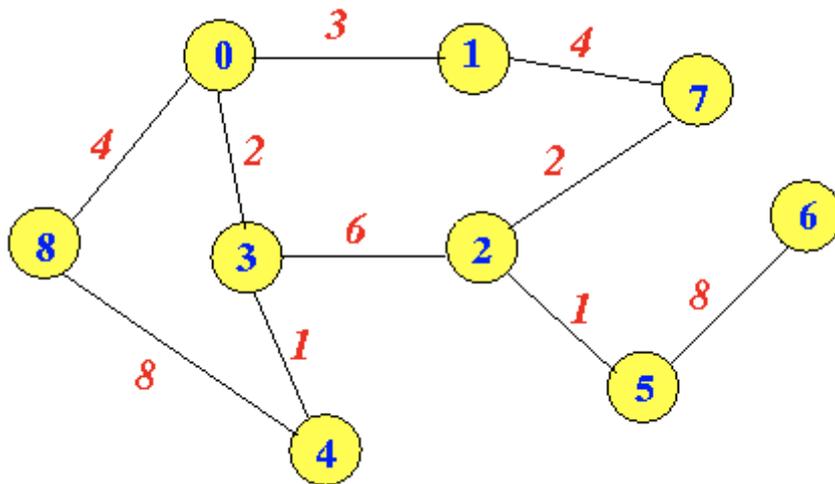
```

```
4: [(3, 4)]
```

```
5: []
```

```
shortest_path(graph7, 0, 5)
```

```
(20, [None, 0, 0, 4, 2, 3])
```



```
# Now for shortest path with this graph (above)  
# Undirected, but weighted
```

```
shortest_path(graph3, 0, 7) # getting shortest from node 0 to node 7
```

```
(7, [None, 0, 3, 0, 3, None, None, 1, 0])
```

```
shortest_path(graph3, 2, 8) # getting shortest from node 2 to node 8
```

```
(15, [3, 7, None, 2, 3, 2, 5, 2, 4])
```

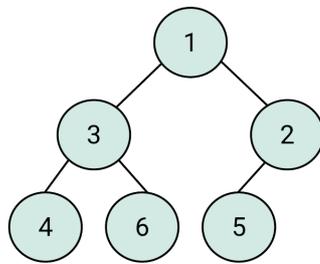
```
import jovian
```

```
jovian.commit()
```

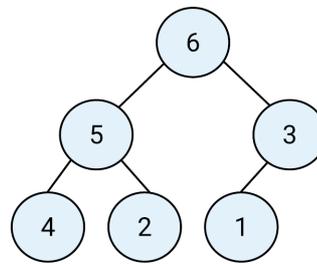
```
[jovian] Updating notebook "evanmarie/python-graph-algorithms" on https://jovian.ai  
[jovian] Committed successfully! https://jovian.ai/evanmarie/python-graph-algorithms  
'https://jovian.ai/evanmarie/python-graph-algorithms'
```

Binary Heap

A data structure to maintain the running minimum/maximum of a set of numbers, supporting efficient addition/removal.



Min heap



Max Heap

Heap operations:

- Insertion - $O(\log N)$
- Min/Max - $O(1)$ (depending on type of heap)
- Deletion - $O(\log N)$
- Convert a list to a heap - $O(n)$

Python's built-in heap: <https://docs.python.org/3/library/heapq.html>

Question: Implement Dijkstra's shortest path algorithm using the `heap` module from Python. What is the complexity of the algorithm?

More Problems

Solve more graph problems here: <https://leetcode.com/tag/graph/>

```
import jovian
```

```
jovian.commit()
```

```
[jovian] Attempting to save notebook..
```

```
[jovian] Updating notebook "aakashns/python-graph-algorithms" on https://jovian.ai/
```

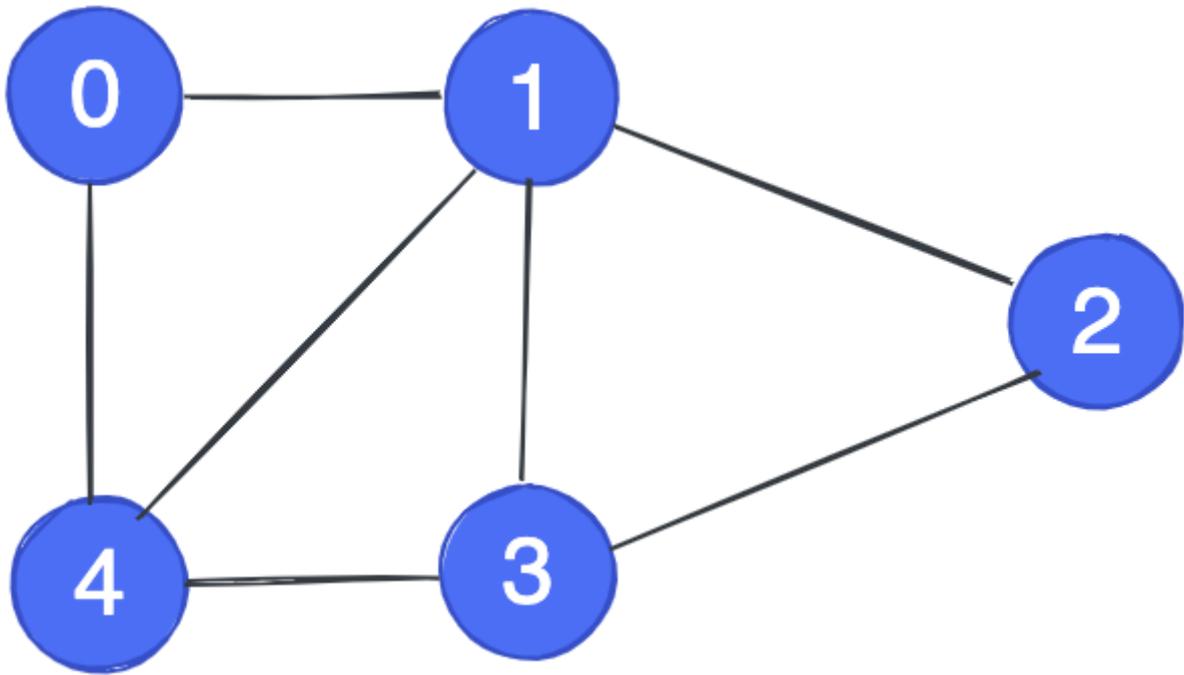
```
[jovian] Uploading notebook..
```

```
[jovian] Capturing environment..
```

```
[jovian] Committed successfully! https://jovian.ai/aakashns/python-graph-algorithms
```

```
'https://jovian.ai/aakashns/python-graph-algorithms'
```

PROVIDED SOLUTIONS



Input Data

```

num_nodes1 = 5
edges1 = [(0, 1), (1, 2), (2, 3), (3, 4), (4, 0), (1, 4), (1, 3)]
num_nodes1, len(edges1)

```

(5, 7)

```

num_nodes3 = 9
edges3 = [(0, 1), (0, 3), (1, 2), (2, 3), (4, 5), (4, 6), (5, 6), (7, 8)]
num_nodes3, len(edges3)

```

(9, 8)

```

num_nodes5 = 9
edges5 = [(0, 1, 3), (0, 3, 2), (0, 8, 4), (1, 7, 4), (2, 7, 2), (2, 3, 6),
          (2, 5, 1), (3, 4, 1), (4, 8, 8), (5, 6, 8)]
num_nodes5, len(edges5)

```

(9, 10)

```

# Directed graph
num_nodes6 = 5
edges6 = [(0, 1), (1, 2), (2, 3), (2, 4), (4, 2), (3, 0)]
num_nodes6, len(edges6)

```

(5, 6)

```

num_nodes7 = 6
edges7 = [(0, 1, 4), (0, 2, 2), (1, 2, 5), (1, 3, 10), (2, 4, 3), (4, 3, 4), (3, 5, 11)]
num_nodes7, len(edges7)

```

(6, 7)

Adjacency List

```
class Graph:
    def __init__(self, num_nodes, edges):
        self.data = [[] for _ in range(num_nodes)]
        for v1, v2 in edges:
            self.data[v1].append(v2)
            self.data[v2].append(v1)

    def __repr__(self):
        return "\n".join("{} : {}".format(i, neighbors) for (i, neighbors) in enumerate(self.data))

    def __str__(self):
        return repr(self)
```

```
g1 = Graph(num_nodes1, edges1)
```

```
g1
```

```
0 : [1, 4]
1 : [0, 2, 4, 3]
2 : [1, 3]
3 : [2, 4, 1]
4 : [3, 0, 1]
```

Adjacency Matrix

Breadth First Search

Complexity $O(m + n)$

```
def bfs(graph, source):
    visited = [False] * len(graph.data)
    queue = []

    visited[source] = True
    queue.append(source)
    i = 0

    while i < len(queue):
        for v in graph.data[queue[i]]:
            if not visited[v]:
                visited[v] = True
                queue.append(v)
        i += 1

    return queue
```

```
bfs(g1, 3)
```

```
[3, 2, 4, 1, 0]
```

Depth First Search

```
def dfs(graph, source):
    visited = [False] * len(graph.data)
    stack = [source]
    result = []

    while len(stack) > 0:
        current = stack.pop()
        if not visited[current]:
            result.append(current)
            visited[current] = True
            for v in graph.data[current]:
                stack.append(v)

    return result
```

```
dfs(g1, 0)
```

```
[0, 4, 1, 3, 2]
```

Directed and Weighted Graph

```
class Graph:
    def __init__(self, num_nodes, edges, directed=False):
        self.data = [[] for _ in range(num_nodes)]
        self.weight = [[] for _ in range(num_nodes)]

        self.directed = directed
        self.weighted = len(edges) > 0 and len(edges[0]) == 3

        for e in edges:
            self.data[e[0]].append(e[1])
            if self.weighted:
                self.weight[e[0]].append(e[2])

            if not directed:
                self.data[e[1]].append(e[0])
                if self.weighted:
                    self.data[e[1]].append(e[2])

    def __repr__(self):
        result = ""
        for i in range(len(self.data)):
            pairs = list(zip(self.data[i], self.weight[i]))
```

```
        result += "{}: {}\n".format(i, pairs)
    return result

def __str__(self):
    return repr(self)
```

```
g7 = Graph(num_nodes7, edges7, directed=True)
```

```
g7
```

```
0: [(1, 4), (2, 2)]
1: [(2, 5), (3, 10)]
2: [(4, 3)]
3: [(5, 11)]
4: [(3, 4)]
5: []
```

```
g7.weight
```

```
[2, 10, 3, 11, 4, []]
```

Shortest Path - Dijkstra's Algorithm

```
def update_distances(graph, current, distance, parent=None):
    """Update the distances of the current node's neighbors"""
    neighbors = graph.data[current]
    weights = graph.weight[current]
    for i, node in enumerate(neighbors):
        weight = weights[i]
        if distance[current] + weight < distance[node]:
            distance[node] = distance[current] + weight
            if parent:
                parent[node] = current

def pick_next_node(distance, visited):
    """Pick the next unvisited node at the smallest distance"""
    min_distance = float('inf')
    min_node = None
    for node in range(len(distance)):
        if not visited[node] and distance[node] < min_distance:
            min_node = node
            min_distance = distance[node]
    return min_node

def shortest_path(graph, source, dest):
    """Find the length of the shortest path between source and destination"""
    visited = [False] * len(graph.data)
    distance = [float('inf')] * len(graph.data)
    parent = [None] * len(graph.data)
    queue = []
```

```

idx = 0

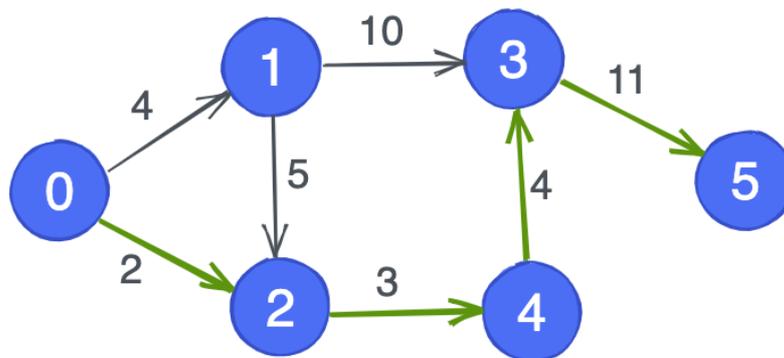
queue.append(source)
distance[source] = 0
visited[source] = True

while idx < len(queue) and not visited[dest]:
    current = queue[idx]
    update_distances(graph, current, distance, parent)

    next_node = pick_next_node(distance, visited)
    if next_node is not None:
        visited[next_node] = True
        queue.append(next_node)
    idx += 1

return distance[dest], distance, parent

```



```
shortest_path(g7, 0, 5)
```

```
(20, [0, 4, 2, 9, 5, 20], [None, 0, 0, 4, 2, 3])
```

```
import jovian
```

```
jovian.commit()
```

```
[jovian] Updating notebook "evanmarie/python-graph-algorithms" on https://jovian.ai
[jovian] Committed successfully! https://jovian.ai/evanmarie/python-graph-algorithms
'https://jovian.ai/evanmarie/python-graph-algorithms'
```

Subarray with Given Sum

The following question was asked during a coding interview for Amazon:

You are given an array of numbers (non-negative). Find a continuous subarray of the list which adds up to a given sum.

[1, 7, 4, 2, 1, 3, 11, 5]
10

```
# Working with lists / arrays of numbers  
list_01 = [1, 7, 4, 2, 1, 3]  
i, j = 2, 6  
print(list_01[2:6])  
print(sum(list_01))
```

[4, 2, 1, 3]

18

```
# Sample input and outputs:  
array = [1, 7, 4, 2, 1, 3]  
target = 10  
output = [2, 6]
```

```
# Function signature  
def sub_array_sum(array, target):  
    pass
```

PROBLEM DESCRIPTION:

- Given an array of integers and a target number
- Must write a function that finds a subarray of consecutive integers within the input array that add up to the target number input
- No negative numbers
- Return the index numbers in the array of the beginning and end integers in the subarray

QUESTIONS About Constraints/Implementation:

- Might the subarray wrap around after reaching the end of the array?
- What should be returned if array is empty?
- What should be returned if the subarray sum does not exist in the array?

- Should the function plan for multiple subarrays or simply return the first to appear?
- If one or more zeros are present at the beginning of the subarray, should they be included?

Steps in PLAIN ENGLISH for brute force approach:

1. Run two loops over the integers
2. First loop tracks beginning of subarray
3. Second loop tracks end
4. When target is reached, return beginning and end

Possible EDGE CASES to Prepare and Test For:

- test00 - Subarray is at end
- test01 - Subarray is at beginning
- test02 - Subarray is in the middle
- test03 - There is no subarray that sums to target
- test04 - Array contains zero
- test05 - Array contains two that sum to target
- test06 - Array is empty
- test07 - Entire array does not sum target

TEST CASES

```
# Edge Test Cases# General Test Cases:
```

```
test00 = {  
  'input': {  
    'array': [1, 4, 3, 6, 2, 2],  
    'target': 10  
  },  
  'output': [3, 6]  
}
```

```
test01 = {  
  'input': {  
    'array': [2, 3, 5, 6, 1, 9],  
    'target': 10  
  },  
  'output': [0, 3]  
}
```

```
test02 = {  
  'input': {  
    'array': [1, 2, 3, 4, 1, 9],
```

```
        'target': 8
    },
    'output': [2, 5]
}

test03 = {
    'input': {
        'array': [1, 2, 3, 5, 6, 8, 7],
        'target': 10
    },
    'output': [1, 4]
}

test04 = {
    'input': {
        'array': [0, 2, 3, 6, 8, 1, 2, 5],
        'target': 11
    },
    'output': [0, 4]
}

test05 = {
    'input': {
        'array': [1, 2, 3, 2, 6, 4, 1],
        'target': 5
    },
    'output': [1, 3]
}

test06 = {
    'input': {
        'array': [],
        'target': 12
    },
    'output': None
}

test07 = {
    'input': {
        'array': [1, 2, 3, 4, 5],
        'target': 16
    },
    'output': None
}
```

General Test Cases:

```

test08 = {
    'input': {
        'array': [1, 2, 3, 4, 5, 6, 7],
        'target': 10
    },
    'output': [0, 4]
}

test09 = {
    'input': {
        'array': [3, 4, 5, 6, 7, 8, 9, 1, 2],
        'target': 24
    },
    'output': [4, 7]
}

test10 = {
    'input': {
        'array': [8, 4, 11, 45, 22, 32, 14, 64, 44],
        'target': 78
    },
    'output': [2, 5]
}

```

```
tests = [test00, test01, test02, test03, test04, test05, test06, test07, test08, test09]
```

TIME FOR SOME CODE!

```

# Always do brute force first, and optimize after.
# I have a tendency to jump to optimized and get in over my head.
# In time, I will learn to swim. Not bad for only 2 months 11 days of coding.
# These comments are my self-pep-talk. Enjoy!

def sum_subarray_brute(array, target):
    n = len(array)          # start_index goes 0 - len(array)

    for start_index in range(0, n):          # Adds n time comp
        for end_index in range(start_index, n+1): # end_index goes from start_index to
                                                    # Mult by approx. n time comp
            if sum(array[start_index:
                          end_index]) == target: # if the sum of the numbers from end_
                                                    # one number index just before end_ir
                return [start_index, end_index] # check if the sum of numbers == targ
                                                    # Complexity- at most mult times n ag
                                                    # Time Comp of brute force: n^3

    return None

```

```
from jovian.pythondsa import evaluate_test_cases
```

```
evaluate_test_cases(sum_subarray_brute, tests)
```

TEST CASE #0

Input:

```
{'array': [1, 4, 3, 6, 2, 2], 'target': 10}
```

Expected Output:

```
[3, 6]
```

Actual Output:

```
[3, 6]
```

Execution Time:

```
0.016 ms
```

Test Result:

PASSED

TEST CASE #1

Input:

```
{'array': [2, 3, 5, 6, 1, 9], 'target': 10}
```

Expected Output:

```
[0, 3]
```

Actual Output:

```
[0, 3]
```

Execution Time:

```
0.007 ms
```

Test Result:

PASSED

TEST CASE #2

Input:

```
{'array': [1, 2, 3, 4, 1, 9], 'target': 8}
```

Expected Output:

```
[2, 5]
```

Actual Output:

```
[2, 5]
```

Execution Time:

```
0.01 ms
```

Test Result:

```
PASSED
```

TEST CASE #3

Input:

```
{'array': [1, 2, 3, 5, 6, 8, 7], 'target': 10}
```

Expected Output:

```
[1, 4]
```

Actual Output:

```
[1, 4]
```

Execution Time:

```
0.009 ms
```

Test Result:

```
PASSED
```

TEST CASE #4

Input:

```
{'array': [0, 2, 3, 6, 8, 1, 2, 5], 'target': 11}
```

Expected Output:

```
[0, 4]
```

Actual Output:

[0, 4]

Execution Time:

0.005 ms

Test Result:

PASSED

TEST CASE #5

Input:

{'array': [1, 2, 3, 2, 6, 4, 1], 'target': 5}

Expected Output:

[1, 3]

Actual Output:

[1, 3]

Execution Time:

0.007 ms

Test Result:

PASSED

TEST CASE #6

Input:

{'array': [], 'target': 12}

Expected Output:

None

Actual Output:

None

Execution Time:

0.002 ms

Test Result:

PASSED

TEST CASE #7

Input:

```
{'array': [1, 2, 3, 4, 5], 'target': 16}
```

Expected Output:

None

Actual Output:

None

Execution Time:

0.011 ms

Test Result:

PASSED

TEST CASE #8

Input:

```
{'array': [1, 2, 3, 4, 5, 6, 7], 'target': 10}
```

Expected Output:

[0, 4]

Actual Output:

[0, 4]

Execution Time:

0.005 ms

Test Result:

PASSED

TEST CASE #9

Input:

```
{'array': [3, 4, 5, 6, 7, 8, 9, 1, 2], 'target': 24}
```

Expected Output:

```
[4, 7]
```

Actual Output:

```
[4, 7]
```

Execution Time:

```
0.019 ms
```

Test Result:

PASSED

TEST CASE #10

Input:

```
{'array': [8, 4, 11, 45, 22, 32, 14, 64, 44], 'target': 78}
```

Expected Output:

```
[2, 5]
```

Actual Output:

```
[2, 5]
```

Execution Time:

```
0.014 ms
```

Test Result:

PASSED

SUMMARY

TOTAL: 11, PASSED: 11, FAILED: 0

```
[[3, 6], True, 0.016),
 [0, 3], True, 0.007),
 [2, 5], True, 0.01),
 [1, 4], True, 0.009),
 [0, 4], True, 0.005),
 [1, 3], True, 0.007),
 (None, True, 0.002),
 (None, True, 0.011),
 [0, 4], True, 0.005),
 [4, 7], True, 0.019),
 [2, 5], True, 0.014]
```

```
import jovian
```

```
jovian.commit()
```

[jovian] Updating notebook "evanmarie/python-subarray-with-given-sum" on <https://jovian.ai>

[jovian] Committed successfully! <https://jovian.ai/evanmarie/python-subarray-with-given-sum>

'<https://jovian.ai/evanmarie/python-subarray-with-given-sum>'

Optimization

```
# Optimizing the brute force: Keep a running sum while incrementing
# Discontinue summing once sum is more than target or equal to target
# I tried to do this all first, and I got confused with my calculations.
# Brute force first all the way!

def sum_subarray_optimized(array, target):
    n = len(array)

    for start_index in range(0, n):
        running_sum = 0

        for end_index in range(start_index, n+1):
            if running_sum == target:
                return [start_index, end_index]
            elif running_sum > target:
                break

            if end_index < n:
                running_sum += array[end_index]

    return None
```

```
evaluate_test_cases(sum_subarray_optimized, tests)
```

TEST CASE #0

Input:

```
{'array': [1, 4, 3, 6, 2, 2], 'target': 10}
```

Expected Output:

```
[3, 6]
```

Actual Output:

```
[3, 6]
```

Execution Time:

```
0.01 ms
```

Test Result:

```
PASSED
```

TEST CASE #1

Input:

```
{'array': [2, 3, 5, 6, 1, 9], 'target': 10}
```

Expected Output:

```
[0, 3]
```

Actual Output:

```
[0, 3]
```

Execution Time:

```
0.005 ms
```

Test Result:

```
PASSED
```

TEST CASE #2

Input:

```
{'array': [1, 2, 3, 4, 1, 9], 'target': 8}
```

Expected Output:

```
[2, 5]
```

Actual Output:

```
[2, 5]
```

Execution Time:

```
0.006 ms
```

Test Result:

PASSED

TEST CASE #3

Input:

```
{'array': [1, 2, 3, 5, 6, 8, 7], 'target': 10}
```

Expected Output:

```
[1, 4]
```

Actual Output:

```
[1, 4]
```

Execution Time:

```
0.006 ms
```

Test Result:

PASSED

TEST CASE #4

Input:

```
{'array': [0, 2, 3, 6, 8, 1, 2, 5], 'target': 11}
```

Expected Output:

```
[0, 4]
```

Actual Output:

[0, 4]

Execution Time:

0.005 ms

Test Result:

PASSED

TEST CASE #5

Input:

{'array': [1, 2, 3, 2, 6, 4, 1], 'target': 5}

Expected Output:

[1, 3]

Actual Output:

[1, 3]

Execution Time:

0.005 ms

Test Result:

PASSED

TEST CASE #6

Input:

{'array': [], 'target': 12}

Expected Output:

None

Actual Output:

None

Execution Time:

0.003 ms

Test Result:

PASSED

TEST CASE #7

Input:

```
{'array': [1, 2, 3, 4, 5], 'target': 16}
```

Expected Output:

None

Actual Output:

None

Execution Time:

0.008 ms

Test Result:

PASSED

TEST CASE #8

Input:

```
{'array': [1, 2, 3, 4, 5, 6, 7], 'target': 10}
```

Expected Output:

[0, 4]

Actual Output:

[0, 4]

Execution Time:

0.004 ms

Test Result:

PASSED

TEST CASE #9

Input:

```
{'array': [3, 4, 5, 6, 7, 8, 9, 1, 2], 'target': 24}
```

Expected Output:

```
[4, 7]
```

Actual Output:

```
[4, 7]
```

Execution Time:

```
0.008 ms
```

Test Result:

PASSED

TEST CASE #10

Input:

```
{'array': [8, 4, 11, 45, 22, 32, 14, 64, 44], 'target': 78}
```

Expected Output:

```
[2, 5]
```

Actual Output:

```
[2, 5]
```

Execution Time:

```
0.008 ms
```

Test Result:

PASSED

SUMMARY

TOTAL: 11, PASSED: 11, FAILED: 0

```
[[3, 6], True, 0.01),
 [0, 3], True, 0.005),
 [2, 5], True, 0.006),
 [1, 4], True, 0.006),
 [0, 4], True, 0.005),
 [1, 3], True, 0.005),
 (None, True, 0.003),
 (None, True, 0.008),
 [0, 4], True, 0.004),
 [4, 7], True, 0.008),
 [2, 5], True, 0.008]
```

MEGA Optimization

```
# One more optimization is possible, again the one I tried from
# the beginning and had my little epic fail.
# Slide the window closed more from the left by moving start_index
# up one spot, then trying the next number after end_index for target.

def sum_subarray_mega_optimized(array, target):
    start_index, end_index, running_sum = 0, 0, 0
    n = len(array)

    while i < n and end_index < n+1:           # Total number of iterations = 2n+1
        if running_sum == target:
            return [start_index, end_index]    # O(n) Time Complexity

        elif running_sum < target:
            if end_index < n:
                running_sum += array[end_index]
                end_index += 1
        elif running_sum > target:
            running_sum -= array[start_index]
            start_index += 1

    return None
```

```
evaluate_test_cases(sum_subarray_mega_optimized, tests)
```

TEST CASE #0

Input:

```
{'array': [1, 4, 3, 6, 2, 2], 'target': 10}
```

Expected Output:

[3, 6]

Actual Output:

[3, 6]

Execution Time:

0.008 ms

Test Result:

PASSED

TEST CASE #1

Input:

{'array': [2, 3, 5, 6, 1, 9], 'target': 10}

Expected Output:

[0, 3]

Actual Output:

[0, 3]

Execution Time:

0.004 ms

Test Result:

PASSED

TEST CASE #2

Input:

{'array': [1, 2, 3, 4, 1, 9], 'target': 8}

Expected Output:

[2, 5]

Actual Output:

[2, 5]

Execution Time:

0.006 ms

Test Result:

PASSED

TEST CASE #3

Input:

{'array': [1, 2, 3, 5, 6, 8, 7], 'target': 10}

Expected Output:

[1, 4]

Actual Output:

[1, 4]

Execution Time:

0.004 ms

Test Result:

PASSED

TEST CASE #4

Input:

{'array': [0, 2, 3, 6, 8, 1, 2, 5], 'target': 11}

Expected Output:

[0, 4]

Actual Output:

[0, 4]

Execution Time:

0.004 ms

Test Result:

PASSED

TEST CASE #5

Input:

```
{'array': [1, 2, 3, 2, 6, 4, 1], 'target': 5}
```

Expected Output:

```
[1, 3]
```

Actual Output:

```
[1, 3]
```

Execution Time:

```
0.004 ms
```

Test Result:

PASSED

TEST CASE #6

Input:

```
{'array': [], 'target': 12}
```

Expected Output:

```
None
```

Actual Output:

```
None
```

Execution Time:

```
0.002 ms
```

Test Result:

PASSED

TEST CASE #7

Input:

```
{'array': [1, 2, 3, 4, 5], 'target': 16}
```

Expected Output:

None

Actual Output:

None

Execution Time:

0.005 ms

Test Result:

PASSED

TEST CASE #8

Input:

```
{'array': [1, 2, 3, 4, 5, 6, 7], 'target': 10}
```

Expected Output:

[0, 4]

Actual Output:

[0, 4]

Execution Time:

0.003 ms

Test Result:

PASSED

TEST CASE #9

Input:

```
{'array': [3, 4, 5, 6, 7, 8, 9, 1, 2], 'target': 24}
```

Expected Output:

[4, 7]

Actual Output:

[4, 7]

Execution Time:

0.006 ms

Test Result:

PASSED

TEST CASE #10

Input:

{'array': [8, 4, 11, 45, 22, 32, 14, 64, 44], 'target': 78}

Expected Output:

[2, 5]

Actual Output:

[2, 5]

Execution Time:

0.007 ms

Test Result:

PASSED

SUMMARY

TOTAL: 11, PASSED: 11, FAILED: 0

[[3, 6], True, 0.008),
[0, 3], True, 0.004),
[2, 5], True, 0.006),
[1, 4], True, 0.004),
[0, 4], True, 0.004),
[1, 3], True, 0.004),
(None, True, 0.002),
(None, True, 0.005),
[0, 4], True, 0.003),

```
([4, 7], True, 0.006),  
([2, 5], True, 0.007)]
```

Default Solutions:

Test case:

```
arr1 = [1, 7, 4, 2, 1, 3, 11, 5]  
target1 = 10
```

Brute force $O(n^3)$:

```
def subarray_sum1(arr, target):  
    n = len(arr)  
    for i in range(n):  
        for j in range(i, n+1):  
            if sum(arr[i:j]) == target:  
                return i, j  
    return None, None
```

```
i, j = subarray_sum1(arr1, target1)  
i, j, arr1[i:j]
```

(2, 6, [4, 2, 1, 3])

Better brute force $O(n^2)$:

```
def subarray_sum2(arr, target):  
    n = len(arr)  
    for i in range(n):  
        s = 0  
        for j in range(i, n+1):  
            if s == target:  
                return i, j  
            s += arr[j]  
    return None, None
```

```
i, j = subarray_sum2(arr1, target1)  
i, j, arr1[i:j]
```

IndexError

Traceback (most recent call last)

/tmp/ipykernel_38/3584318455.py in <module>

```
----> 1 i, j = subarray_sum2(arr1, target1)  
      2 i, j, arr1[i:j]
```

/tmp/ipykernel_38/1661949270.py in subarray_sum2(arr, target)

```
6         if s == target:  
7             return i, j
```

```
-----> 8         s += arr[j]
          9         return None, None
```

IndexError: list index out of range

Greedy algorithm $O(n)$:

```
def subarray_sum3(arr, target):
    n = len(arr)
    i, j, s = 0, 0, 0
    while i < n and j <= n:
        if s == target:
            return i, j
        elif s < target:
            s += arr[j]
            j += 1
        elif s > target:
            s -= arr[i]
            i += 1
    return None, None
```

```
i, j = subarray_sum3(arr1, target1)
i, j, arr1[i:j]
```

(2, 6, [4, 2, 1, 3])

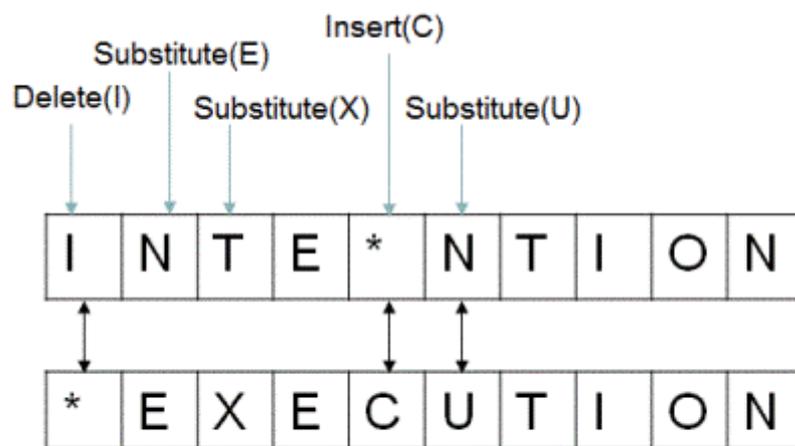
Minimum Edit Distance

The following interview was asked during a coding interview at Google:

Given two strings A and B, find the minimum number of steps required to convert A to B. (each operation is counted as 1 step.) You have the following 3 operations permitted on a word:

- Insert a character
- Delete a character
- Replace a character

Here's a visual representation (source: iDeserve)



Own words: given two strings, we need to perform a series of operations to the first string to convert it to the second string. The operations possible are insert, delete, and replace.

Inputs: two strings, such as string_1 = "intention", string_2 = "execution"

Output: the number of operations it takes to convert string_1 to string_2, i.e. 5

```
# Function signature:  
def minimum_steps(string_1, string_2):  
    pass
```

Test Cases:

test00 - General case

test01 - No change is required

test03 - All characters need to be changed

test04 - Both strings are equal length

test05 - Strings are unequal length

test06 - One or both strings are empty

test07 - Words only require one of the operations repeated

TEST CASES:

Edge Test Cases

```
test00 = {  
  'input': {  
    'string_1': "intention",  
    'string_2': "execution"  
  },  
  'output': 5  
}
```

```
test01 = {  
  'input': {  
    'string_1': "cat",  
    'string_2': "cat"  
  },  
  'output': 0  
}
```

```
test02 = {  
  'input': {  
    'string_1': "supper",  
    'string_2': "tactful"  
  },  
  'output': 7  
}
```

```
test03 = {  
  'input': {  
    'string_1': "sugar",  
    'string_2': "lover"  
  },  
  'output': 4  
}
```

```
test04 = {  
  'input': {  
    'string_1': "chester",  
    'string_2': "plus"  
  },  
  'output': 6  
}
```

```
test05 = {  
  'input': {  
    'string_1': "",  
    'string_2': "toaster"  
  },  
  'output': 7  
}
```

```
test06 = {
```

```

    'input': {
        'string_1': "scooter",
        'string_2': "travels"
    },
    'output': 7
}

test07 = {
    'input': {
        'string_1': "chicken",
        'string_2': "kitchen"
    },
    'output': 4
}

test08 = {
    'input': {
        'string_1': "kitten",
        'string_2': "sitting"
    },
    'output': 3
}

test09 = {
    'input': {
        'string_1': "sunday",
        'string_2': "saturday"
    },
    'output': 3
}

```

```
tests = [test00, test01, test02, test03, test04, test05, test06, test07, test08, test09]
```

```
from jovian.pythondsa import evaluate_test_cases
```

STEPS:

- if string_1 becomes empty first, add all of string_2 to it, done
- if string_2 becomes empty first, delete all characters from string_1, done
- if the characters are equal, ignore both, move on
- if the characters are not equal, it must be deleted, swapped, or shifted down
- if deleted, recursively solve after ignoring first character of string_1
- if swapped, move both cursors forward, recursively solve after ignoring the current character of each
- if shifting, shift string_1 right a spot, and recursively solve the problem with the new version of string_1 and normal string_2

BRUTE FORCE

- When in doubt, question if it is possible to solve the problem recursively
- Can you find one or more subproblems that can be repeated to solve the overall

```
def minimum_steps(string_1, string_2, index_1 = 0, index_2 = 0):
    if index_1 == len(string_1):
        return len(string_2) - index_2

    elif index_2 == len(string_1):
        return len(string_1) - index_1

    elif string_1[index_1] == string_2[index_2]:
        return minimum_steps(string_1, string_2, index_1+1, index_2+1)

    else: # performing 1) delete 2) swap 3) shift/insert
        return 1 + min(minimum_steps(string_1, string_2, index_1+1, index_2),
                        minimum_steps(string_1, string_2, index_1+1, index_2+1),
                        minimum_steps(string_1, string_2, index_1, index_2+1))
```

```
print("intention and execution: ", minimum_steps('intention', 'execution'))
print("chicken and kitchen: ", minimum_steps('chicken', 'kitchen'))
print("kitten and sitting: ", minimum_steps('kitten', 'sitting'))
print("sunday and saturday: ", minimum_steps('sunday', 'saturday'))
print("chester and plus: ", minimum_steps('chester', 'plus'))
print("please and choose: ", minimum_steps('please', 'choose'))
```

```
intention and execution: 5
chicken and kitchen: 4
kitten and sitting: 3
sunday and saturday: 5
```

```
-----
IndexError                                Traceback (most recent call last)
/tmp/ipykernel_50/315974634.py in <module>
      3 print("kitten and sitting: ", minimum_steps('kitten', 'sitting'))
      4 print("sunday and saturday: ", minimum_steps('sunday', 'saturday'))
----> 5 print("chester and plus: ", minimum_steps('chester', 'plus'))
      6 print("please and choose: ", minimum_steps('please', 'choose'))

/tmp/ipykernel_50/1719044234.py in minimum_steps(string_1, string_2, index_1, index_2)
     10
     11     else: # performing 1) delete 2) swap 3) shift/insert
----> 12     return 1 + min(minimum_steps(string_1, string_2, index_1+1, index_2),
     13                     minimum_steps(string_1, string_2, index_1+1,
index_2+1),
     14                     minimum_steps(string_1, string_2, index_1, index_2+1))
```

```
/tmp/ipykernel_50/1719044234.py in minimum_steps(string_1, string_2, index_1, index_2)
  10
  11     else: # performing 1) delete 2) swap 3) shift/insert
----> 12     return 1 + min(minimum_steps(string_1, string_2, index_1+1, index_2),
  13                       minimum_steps(string_1, string_2, index_1+1,
index_2+1)),
  14                       minimum_steps(string_1, string_2, index_1, index_2+1))
```

```
/tmp/ipykernel_50/1719044234.py in minimum_steps(string_1, string_2, index_1, index_2)
  10
  11     else: # performing 1) delete 2) swap 3) shift/insert
----> 12     return 1 + min(minimum_steps(string_1, string_2, index_1+1, index_2),
  13                       minimum_steps(string_1, string_2, index_1+1,
index_2+1)),
  14                       minimum_steps(string_1, string_2, index_1, index_2+1))
```

```
/tmp/ipykernel_50/1719044234.py in minimum_steps(string_1, string_2, index_1, index_2)
  10
  11     else: # performing 1) delete 2) swap 3) shift/insert
----> 12     return 1 + min(minimum_steps(string_1, string_2, index_1+1, index_2),
  13                       minimum_steps(string_1, string_2, index_1+1,
index_2+1)),
  14                       minimum_steps(string_1, string_2, index_1, index_2+1))
```

```
/tmp/ipykernel_50/1719044234.py in minimum_steps(string_1, string_2, index_1, index_2)
  10
  11     else: # performing 1) delete 2) swap 3) shift/insert
----> 12     return 1 + min(minimum_steps(string_1, string_2, index_1+1, index_2),
  13                       minimum_steps(string_1, string_2, index_1+1,
index_2+1)),
  14                       minimum_steps(string_1, string_2, index_1, index_2+1))
```

```
/tmp/ipykernel_50/1719044234.py in minimum_steps(string_1, string_2, index_1, index_2)
  10
  11     else: # performing 1) delete 2) swap 3) shift/insert
----> 12     return 1 + min(minimum_steps(string_1, string_2, index_1+1, index_2),
  13                       minimum_steps(string_1, string_2, index_1+1,
index_2+1)),
  14                       minimum_steps(string_1, string_2, index_1, index_2+1))
```

```
/tmp/ipykernel_50/1719044234.py in minimum_steps(string_1, string_2, index_1, index_2)
  12     return 1 + min(minimum_steps(string_1, string_2, index_1+1, index_2),
  13                       minimum_steps(string_1, string_2, index_1+1,
index_2+1)),
----> 14                       minimum_steps(string_1, string_2, index_1, index_2+1))
  15
```

```
/tmp/ipykernel_50/1719044234.py in minimum_steps(string_1, string_2, index_1, index_2)
  12     return 1 + min(minimum_steps(string_1, string_2, index_1+1, index_2),
```

```
13             minimum_steps(string_1, string_2, index_1+1,
index_2+1),
---> 14             minimum_steps(string_1, string_2, index_1, index_2+1))
15
```

```
/tmp/ipykernel_50/1719044234.py in minimum_steps(string_1, string_2, index_1, index_2)
12         return 1 + min(minimum_steps(string_1, string_2, index_1+1, index_2),
13                         minimum_steps(string_1, string_2, index_1+1,
index_2+1),
---> 14                         minimum_steps(string_1, string_2, index_1, index_2+1))
15
```

```
/tmp/ipykernel_50/1719044234.py in minimum_steps(string_1, string_2, index_1, index_2)
12         return 1 + min(minimum_steps(string_1, string_2, index_1+1, index_2),
13                         minimum_steps(string_1, string_2, index_1+1,
index_2+1),
---> 14                         minimum_steps(string_1, string_2, index_1, index_2+1))
15
```

```
/tmp/ipykernel_50/1719044234.py in minimum_steps(string_1, string_2, index_1, index_2)
6             return len(string_1) - index_1
7
----> 8         elif string_1[index_1] == string_2[index_2]:
9             return minimum_steps(string_1, string_2, index_1+1, index_2+1)
10
```

IndexError: string index out of range

```
evaluate_test_cases(minimum_steps, tests)
```

TEST CASE #0

Input:

```
{'string_1': 'intention', 'string_2': 'execution'}
```

Expected Output:

5

Actual Output:

5

Execution Time:

90.121 ms

Test Result:

PASSED

TEST CASE #1

Input:

```
{'string_1': 'cat', 'string_2': 'cat'}
```

Expected Output:

0

Actual Output:

0

Execution Time:

0.006 ms

Test Result:

PASSED

TEST CASE #2

Input:

```
{'string_1': 'supper', 'string_2': 'tactful'}
```

Expected Output:

7

Actual Output:

7

Execution Time:

2.951 ms

Test Result:

PASSED

TEST CASE #3

Input:

```
{'string_1': 'sugar', 'string_2': 'lover'}
```

Expected Output:

4

Actual Output:

4

Execution Time:

0.409 ms

Test Result:

PASSED

TEST CASE #4

```
-----  
IndexError                                Traceback (most recent call last)  
/tmp/ipykernel_50/1409213795.py in <module>  
----> 1 evaluate_test_cases(minimum_steps, tests)  
  
/opt/conda/lib/python3.9/site-packages/jovian/pythonsa/__init__.py in  
evaluate_test_cases(function, test_cases, error_only, summary_only)  
    85     if not error_only:  
    86         print("\n\033[1mTEST CASE #{}\033[0m".format(i))  
----> 87     result = evaluate_test_case(function, test_case, display=False)  
    88     results.append(result)  
    89     if error_only and not result[1]:  
  
/opt/conda/lib/python3.9/site-packages/jovian/pythonsa/__init__.py in  
evaluate_test_case(function, test_case, display)  
    63  
    64     start = timer()  
----> 65     actual_output = function(**inputs)  
    66     end = timer()  
    67  
  
/tmp/ipykernel_50/2643043875.py in minimum_steps(string_1, string_2, index_1, index_2)  
    10  
    11     else: # performing 1) delete 2) swap 3) shift/insert  
----> 12     return 1 + min(minimum_steps(string_1, string_2, index_1+1, index_2),  
    13                     minimum_steps(string_1, string_2, index_1+1,  
index_2+1),  
    14                     minimum_steps(string_1, string_2, index_1, index_2+1))
```

```

/tmp/ipykernel_50/2643043875.py in minimum_steps(string_1, string_2, index_1, index_2)
    10
    11     else: # performing 1) delete 2) swap 3) shift/insert
----> 12     return 1 + min(minimum_steps(string_1, string_2, index_1+1, index_2),
    13                     minimum_steps(string_1, string_2, index_1+1,
index_2+1)),
    14                     minimum_steps(string_1, string_2, index_1, index_2+1))

/tmp/ipykernel_50/2643043875.py in minimum_steps(string_1, string_2, index_1, index_2)
    10
    11     else: # performing 1) delete 2) swap 3) shift/insert
----> 12     return 1 + min(minimum_steps(string_1, string_2, index_1+1, index_2),
    13                     minimum_steps(string_1, string_2, index_1+1,
index_2+1)),
    14                     minimum_steps(string_1, string_2, index_1, index_2+1))

/tmp/ipykernel_50/2643043875.py in minimum_steps(string_1, string_2, index_1, index_2)
    10
    11     else: # performing 1) delete 2) swap 3) shift/insert
----> 12     return 1 + min(minimum_steps(string_1, string_2, index_1+1, index_2),
    13                     minimum_steps(string_1, string_2, index_1+1,
index_2+1)),
    14                     minimum_steps(string_1, string_2, index_1, index_2+1))

/tmp/ipykernel_50/2643043875.py in minimum_steps(string_1, string_2, index_1, index_2)
    10
    11     else: # performing 1) delete 2) swap 3) shift/insert
----> 12     return 1 + min(minimum_steps(string_1, string_2, index_1+1, index_2),
    13                     minimum_steps(string_1, string_2, index_1+1,
index_2+1)),
    14                     minimum_steps(string_1, string_2, index_1, index_2+1))

/tmp/ipykernel_50/2643043875.py in minimum_steps(string_1, string_2, index_1, index_2)
    10
    11     else: # performing 1) delete 2) swap 3) shift/insert
----> 12     return 1 + min(minimum_steps(string_1, string_2, index_1+1, index_2),
    13                     minimum_steps(string_1, string_2, index_1+1,
index_2+1)),
    14                     minimum_steps(string_1, string_2, index_1, index_2+1))

/tmp/ipykernel_50/2643043875.py in minimum_steps(string_1, string_2, index_1, index_2)
    12     return 1 + min(minimum_steps(string_1, string_2, index_1+1, index_2),
    13                     minimum_steps(string_1, string_2, index_1+1,
index_2+1)),
----> 14                     minimum_steps(string_1, string_2, index_1, index_2+1))
    15
    16 print("intention and execution: ", minimum_steps_memo('intention', 'execution'))

/tmp/ipykernel_50/2643043875.py in minimum_steps(string_1, string_2, index_1, index_2)

```

```

12         return 1 + min(minimum_steps(string_1, string_2, index_1+1, index_2),
13                         minimum_steps(string_1, string_2, index_1+1,
index_2+1)),
---> 14                 minimum_steps(string_1, string_2, index_1, index_2+1))
15
16 print("intention and execution: ", minimum_steps_memo('intention', 'execution'))

/tmp/ipykernel_50/2643043875.py in minimum_steps(string_1, string_2, index_1, index_2)
12         return 1 + min(minimum_steps(string_1, string_2, index_1+1, index_2),
13                         minimum_steps(string_1, string_2, index_1+1,
index_2+1)),
---> 14                 minimum_steps(string_1, string_2, index_1, index_2+1))
15
16 print("intention and execution: ", minimum_steps_memo('intention', 'execution'))

/tmp/ipykernel_50/2643043875.py in minimum_steps(string_1, string_2, index_1, index_2)
12         return 1 + min(minimum_steps(string_1, string_2, index_1+1, index_2),
13                         minimum_steps(string_1, string_2, index_1+1,
index_2+1)),
---> 14                 minimum_steps(string_1, string_2, index_1, index_2+1))
15
16 print("intention and execution: ", minimum_steps_memo('intention', 'execution'))

/tmp/ipykernel_50/2643043875.py in minimum_steps(string_1, string_2, index_1, index_2)
6             return len(string_1) - index_1
7
----> 8         elif string_1[index_1] == string_2[index_2]:
9             return minimum_steps(string_1, string_2, index_1+1, index_2+1)
10

```

IndexError: string index out of range

BRUTE FORCE TIME COMPLEXITY:

- Total number of recursions = lengths of the two strings combined.
- After that, complexity should be calculated on adding the total length of the two strings combined minus 1 for each additional substring comparison and computation. Thus reaching the time complexity below.

Time Complexity: $O(3(n_1+n_2))$

MEMOIZATION:

There are many repetitions that can be reduced by using a cache.

Before doing any computations, check the memo dictionary for solutions for the changing variables, if so, return it, if not, compute it and add it, and return the value from the memo.

```

def minimum_steps_memo(string_1, string_2):
    memo = dict()

```

```

def recursive_memo(index_1, index_2):
    key = index_1, index_2

    if key in memo:                                     # if the index 1 to 2 comparison is in memo
        return memo[key]

    elif index_1 == len(string_1):                     # if string_1 becomes empty first
        memo[key] = len(string_2) - index_2

    elif index_2 == len(string_2):                     # if string_2 becomes empty first
        memo[key] = len(string_1) - index_1

    elif string_1[index_1] == string_2[index_2]:      # if strings are identical
        memo[key] = recursive_memo(index_1 + 1, index_2 + 1) # continue recursion

    else: # performing 1) delete 2) swap 3) shift/insert
        memo[key] = 1 + min(
            recursive_memo(index_1 + 1, index_2),
            recursive_memo(index_1 + 1, index_2 + 1),
            recursive_memo(index_1, index_2 + 1))

    return memo[key]

return recursive_memo(0, 0)

```

```

print("intention and execution: ", minimum_steps_memo('intention', 'execution'))
print("chicken and kitchen: ", minimum_steps_memo('chicken', 'kitchen'))
print("kitten and sitting: ", minimum_steps_memo('kitten', 'sitting'))
print("sunday and saturday: ", minimum_steps_memo('sunday', 'saturday'))
print("chester and plus: ", minimum_steps_memo('chester', 'plus'))
print("please and choose: ", minimum_steps_memo('please', 'choose'))

```

```

intention and execution: 5
chicken and kitchen: 4
kitten and sitting: 3
sunday and saturday: 3
chester and plus: 6
please and choose: 4

```

```

evaluate_test_cases(minimum_steps_memo, tests)

```

TEST CASE #0

Input:

```
{'string_1': 'intention', 'string_2': 'execution'}
```

Expected Output:

5

Actual Output:

5

Execution Time:

0.099 ms

Test Result:

PASSED

TEST CASE #1

Input:

```
{'string_1': 'cat', 'string_2': 'cat'}
```

Expected Output:

0

Actual Output:

0

Execution Time:

0.008 ms

Test Result:

PASSED

TEST CASE #2

Input:

```
{'string_1': 'supper', 'string_2': 'tactful'}
```

Expected Output:

7

Actual Output:

7

Execution Time:

0.079 ms

Test Result:

PASSED

TEST CASE #3

Input:

```
{'string_1': 'sugar', 'string_2': 'lover'}
```

Expected Output:

4

Actual Output:

4

Execution Time:

0.054 ms

Test Result:

PASSED

TEST CASE #4

Input:

```
{'string_1': 'chester', 'string_2': 'plus'}
```

Expected Output:

6

Actual Output:

6

Execution Time:

0.059 ms

Test Result:

PASSED

TEST CASE #5

Input:

```
{'string_1': '', 'string_2': 'toaster'}
```

Expected Output:

7

Actual Output:

7

Execution Time:

0.004 ms

Test Result:

PASSED

TEST CASE #6

Input:

```
{'string_1': 'scooter', 'string_2': 'travels'}
```

Expected Output:

7

Actual Output:

7

Execution Time:

0.07 ms

Test Result:

PASSED

TEST CASE #7

Input:

```
{'string_1': 'chicken', 'string_2': 'kitchen'}
```

Expected Output:

4

Actual Output:

4

Execution Time:

0.076 ms

Test Result:

PASSED

TEST CASE #8

Input:

```
{'string_1': 'kitten', 'string_2': 'sitting'}
```

Expected Output:

3

Actual Output:

3

Execution Time:

0.072 ms

Test Result:

PASSED

TEST CASE #9

Input:

```
{'string_1': 'sunday', 'string_2': 'saturday'}
```

Expected Output:

3

Actual Output:

3

Execution Time:

0.059 ms

Test Result:

PASSED

SUMMARY

TOTAL: 10, PASSED: 10, FAILED: 0

```
[(5, True, 0.099),  
(0, True, 0.008),  
(7, True, 0.079),  
(4, True, 0.054),  
(6, True, 0.059),  
(7, True, 0.004),  
(7, True, 0.07),  
(4, True, 0.076),  
(3, True, 0.072),  
(3, True, 0.059)]
```

MEMOIZATION TIME COMPLEXITY:

- Memoization method only needs to compute the solution for a key once, a fixed number of comparisons and an addition.
- The time required is constant, and the upper bound is some constant multiple of total number of memoizations necessary.
- The time complexity is therefore equal to the sum of the lengths of the two strings.
Time Complexity: $O(n_1+n_2)$

```
import jovian
```

```
jovian.commit()
```

```
[jovian] Updating notebook "evanmarie/python-minimum-edit-distance-41118" on  
https://jovian.ai
```

```
[jovian] Committed successfully! https://jovian.ai/evanmarie/python-minimum-edit-distance-41118
```

```
'https://jovian.ai/evanmarie/python-minimum-edit-distance-41118'
```

Brute force (recursion - exponential):

```
def min_edit_distance(str1, str2, i1=0, i2=0):
    if i1 == len(str1):
        return len(str2) - i2
    if i2 == len(str2):
        return len(str1) - i1
    if str1[i1] == str2[i2]:
        return min_edit_distance(str1, str2, i1+1, i2+1)
    return 1 + min(min_edit_distance(str1, str2, i1, i2+1), # Insert at beginning of str1
                  min_edit_distance(str1, str2, i1+1, i2), # Remove from beginning of str1
                  min_edit_distance(str1, str2, i1+1, i2+1)) # Swap first character of str1 and str2
```

```
min_edit_distance('wednesday', 'thursday')
```

5

```
min_edit_distance('intention', 'execution')
```

5

Improved (memoization - $O(n_1 * n_2)$):

```
def min_edit_distance2(str1, str2):
    memo = {}
    def recurse(i1, i2):
        key = (i1, i2)
        if key in memo:
            return memo[key]
        elif i1 == len(str1):
            memo[key] = len(str2) - i2
        elif i2 == len(str2):
            memo[key] = len(str1) - i1
        elif str1[i1] == str2[i2]:
            memo[key] = recurse(i1+1, i2+1)
        else:
            memo[key] = 1 + min(recurse(i1, i2+1),
                               recurse(i1+1, i2),
                               recurse(i1+1, i2+1))
    return memo[key]
    return recurse(0, 0)
```

```
min_edit_distance2('intention', 'execution')
```

5

Best (Dynamic programming - $O(n_1 * n_2)$):

```
# left as an exercise
```

