Fundamentals of Dynamic Programming

with Avik Das (LinkedIn Learning)

Chapter 1: What is Dynamic Programming:

⇨ ***Dynamic programming*** is a technique for computing, a recursive algorithm with the highly overlapping sub-problems structure.
⇨ Dynamic programming ***achieves efficiency by solving each sub problem only once***.
⇨ A ***recursive algorithm*** breaks down a larger problem into smaller sub-problems, combining the smaller results into the larger ones. When a problem is broken down into independent chunks with no overlap the technique is known as ***divide and conquer***.
⇨ When sub problems overlap, resulting in the same sub problem being solved again and again, dynamic programming becomes useful
⇨ ***Memoization (top-down approach)*** is a technique for speeding up computations by caching results of repeated calculations or sub problems. This way, when the same calculation has to be performed a second time, the result can be quickly looked up in the ***cache***, instead of performing that calculation again.
⇨ ***Bottom-Up Dynamic Programming*** - earlier sub-problems are solved first so that their results are ready when needed later. Because of a predictable order, this ***allows us to throw away earlier results*** when they will no longer be needed.

→ draw the ***dependencies between sub-problems***

→ ***directed acyclic graph or DAG*** A graph means we have vertices and edges. In this case, the sub-problems and the dependencies between them. The ***edges have direction***. Meaning certain sub-problems depend on others, but not vice versa. And finally, there are ***no cycles***. Meaning we can't come back to a vertex just by following the edges. Otherwise, sub-problems would eventually depend on themselves. ***All DAGs are linearizable. Meaning they can be laid out in an order with the arrows going only in one direction***. This is what defines the order in which we'll solve sub-problems.

⇨ Bottom-up is the best approach. Fast as Memoization but using less space, because it only keeps up with the information you will need each time.

⇨ Bottom-Up Dynamic Programming, represents ***recursive problems as directed acyclic graphs***.

```
def fib_bottom(n):
    a = 1  # f(i - 2)
    b = 1  # f(i - 1)
    for i in range(2, n + 1):
      a, b = b, a + b
    return b
```

⇨ The DAG representation can be ***traversed in order of dependency***, solving sub-problems before they are needed.
⇨ By throwing away under needed results, ***problems can be solved with both minimal time and space***.

CHAPTER 1 QUESTIONS:

⇨ **How does dynamic programming differ from divide and conquer?**

- Divide and conquer is used to solve different problems than dynamic programming, making the two incomparable.
- Divide and conquer deals with non-overlapping subproblems. Dynamic programming deals with highly-overlapping subproblems.

⇨ **How does dynamic programming enable solving certain recursive problems more efficiently**?

- by not repeating the same computations multiple times. Dynamic programming makes sure each subproblem is solved exactly once

⇨ **Why is it important the dependency graph of any problem be acyclic?**

- There is a defined order in which to solve subproblems. All Directed Acyclic Graphs are linearizable, making it possible to solve the subproblems in order of dependency.

⇨ **What is the downside of using memoization**?

- It potentially increases memory usage. Answers to subproblems are kept around even after they are not needed.

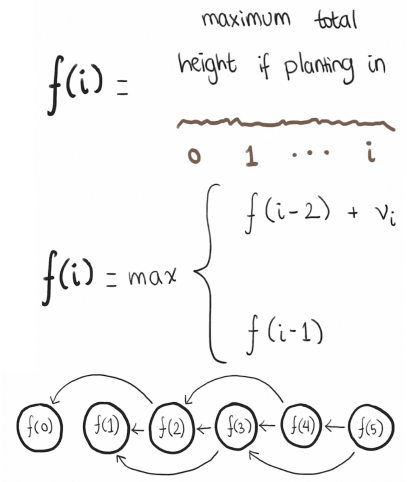
⇨ **Why is the straightforward recursive implementation of the Fibonacci sequence slow**?

- The recursive implementation repeats the same computations many times. Not only are the same computations repeated, but they are repeated more often as the input gets larger.

Chapter 2: Examples of Dynamic Programming:

⇨ ***Flowerbox Problem*** →

We have some soil with a few places to plant flowers. Each place we can plant a flower has some amount of nutrients, which we'll call visa V sub i. For example, the first location has three units of nutrients, the next location has 10 and so on. If you plant a flower in one of these locations the height of the flower will be proportional to the quantity of nutrients at that location. If you plant the flower at the very left and the flower will grow three units. So here are the rules. ***You can plant as many flowers as you want as long as they're not right next to each other. Otherwise the flowers will compete for nutrients. Your goal is to maximize the total height of all the flowers that you plant.*** *What we wanna do is plant flowers where the V sub i are 10 and two. This results in a total height of 12 units.*



⇨ This presents a recurrence relation

⇨ Final answer: ***f(n-1)***

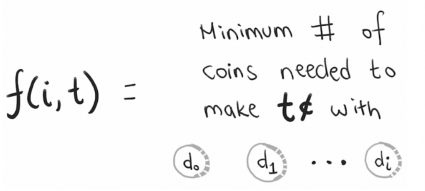
n = total number of spots

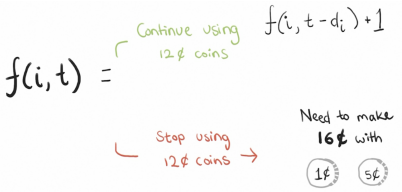
Dependency graph: each subproblem depends on the two before it. (Very similar to fibonacci one.)

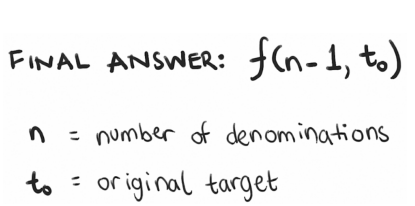
```
def flowerbox(nutrient_values):
    a = 0  # f(i - 2)
    b = 0  # f(i - 1)
    for val in nutrient_values:
        a, b = b, max(a + val, b)
    return b
```

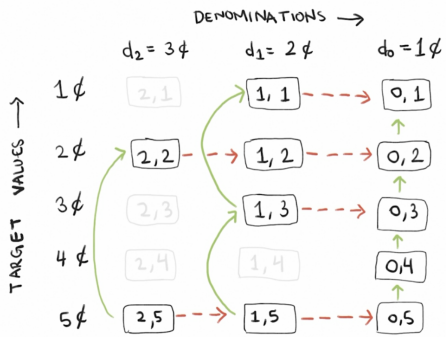
Change-Making Problem →

You're given a target t zero, and a set of denominations d sub i. You can use as many coins of each denomination as you want, as long as the total doesn't go above the target. And he want to minimize the number of coins you need to add up to t zero.









⇨ ***i*** is how many denominations are present, and ***t*** is the goal number of cents

⇨ We must repeatedly choose if we use a coin and up the coin count or if we try the next combination of values

⇨ The function has integer inputs, making it easy to identify subproblems. It also references itself, recurrence relation.

⇨ Dependency graph is more complicated here, because there are 2 inputs

⇨ should use ***memoization*** in this one, since we know we will not need to solve every problem in the dependency graph (we are limited in choices by our goal.)

CHAPTER 2 QUESTIONS:

⇨ **Why is memoization a better option for the change making problem as compared to bottom-up programming?**

Not all subproblems need to be solved.

Because not all the subproblems need to be solved, memoization means only the necessary ones are computed.

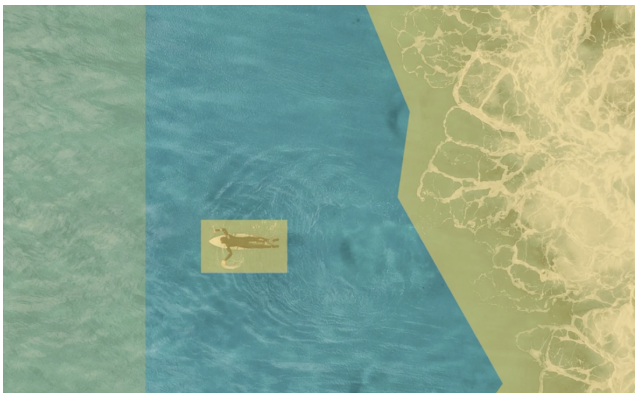
⇨ **Which property of the recurrence relation for the flowerbox problem is not important to solving the problem with dynamic programming?**

The function has integer outputs.

As long as the outputs can be combined, it doesn't matter what the outputs look like.

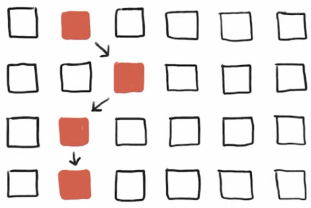
```python
def change_making(denominations, target):
    cache = {}
    def subproblem(i, t):
        if (i, t) in cache: return cache[(i, t)]  # memoization
        # Compute the lowest number of coins we need if choosing
        # to take a coin of the current denomination.
        val = denominations[i]
        if val > t:      # current denomination is too large
            choice_take = math.inf
        elif val == t:  # target reached
            choice_take = 1
        else:           # take and recurse
            choice_take = 1 + subproblem(i, t - val)
        # Compute the lowest number of coins we need if not taking
        # any more coins of the current denomination.
        if i == 0:      # not an option if no more denominations
            choice_leave = math.inf
        else:           # recurse with remaining denominations
            choice_leave = subproblem(i - 1, t)
        optimal = min(choice_take, choice_leave)
        cache[(i, t)] = optimal
        return optimal
    return subproblem(len(denominations) - 1, target)
```

Chapter 3: Real-World Dynamic Programming and Content-Aware Image Resizing:



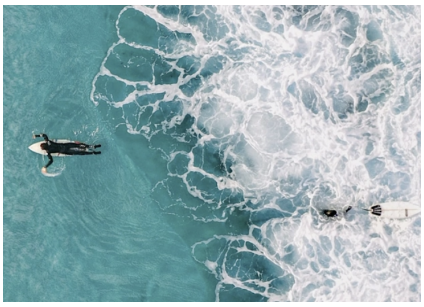
⇨ ***Content-aware image resizing***, is when you change the size of an image, taking into account what's in that image. Different images are resized differently. ***Seam carving*** is one implementation of content aware resizing, in which an uninteresting sequence of pixels is removed from the image.

⇨ Now that we know which part of the photo we can sacrifice, we want to find a string of pixels in the uninteresting region, that goes down the height of the image. The important part is that the pixels are connected. But not necessarily all in a straight line. The disconnected string of pixels is known as a ***seam***. In our photo of the surfer, such a seam might go down the image right next to the waves.

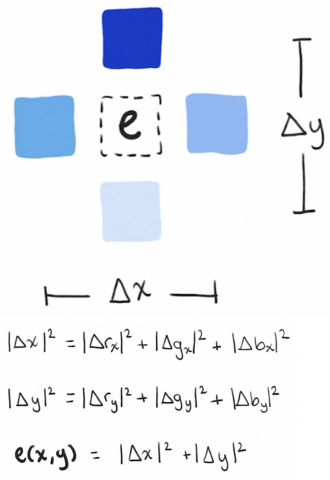


⇨ Since the ***seam***, this string of pixels is in an uninteresting region, we can simply remove the pixels in the seam, the image will be one pixel smaller in width, but it will look like we did anything. Finally, we can repeat this process of finding uninteresting seams and removing them.

⇨ ***Pre-processing*** - This is needed to get the image into a form where we can apply dynamic programming, meaning reading the image and finding the uninteresting parts. Next, we applied the actual ***dynamic programming algorithm***. In this problem, meaning finding the least interesting seam. Finally, we need to do some ***post processing***, to take the results of the dynamic programming algorithm and use them, meaning removing the relevant pixels from the image and maybe saving that image to a file.



⇨ The ***energy*** at a certain pixel in an image is a numerical value that captures ***how much the image is changing around that pixel***. Areas of an image with ***high energy values are interesting,*** and ***areas with low energy values are uninteresting***.



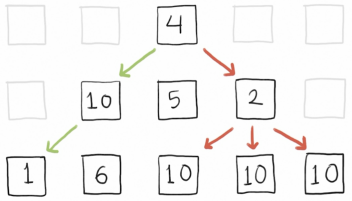
⇨ Taking a pixel and the two surrounding pixels on the left and right, if the color changes significantly between these pixels, then the difference between these pixels is large. If the surrounding pixels have a color similar to the middle one, the

difference is small. To formally define the energy at a pixel we have to look around that pixel both horizontally and vertically.
⇨ Basically the ***Pythagorean Theorem*** applied to the color values of the pixels horizontally and vertically.

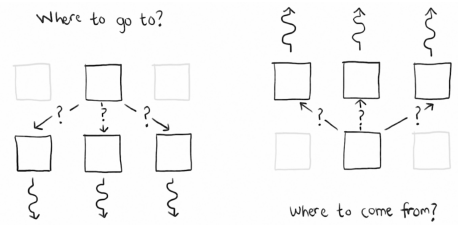
```
def energy_at(pixels, x, y):
  height = len(pixels)
  width = len(pixels)[0]

  x0 = x if x == 0 else x - 1
  x1 = x if x == width-1 else x + 1

  delta_x_red = pixels[y][x0].red - pixels[y][x1].red
  delta_x_green = pixels[y][x0].green - pixels[y][x1].green
  delta_x_blue = pixels[y][x0].blue - pixels[y][x1].blue

  delta_x = ((delta_x_red ** 2) + (delta_x_green ** 2) +
             (delta_x_blue ** 2))

  y0 = y if y == 0 else y - 1
  y1 = y if y == height else y + 1

  delta_y_red = pixels[y0][x].red - pixels[y1][x].red
  delta_y_green = pixels[y0][x].green - pixels[y1][x].green
  delta_y_blue = pixels[y0][x].blue - pixels[y1][x].blue

  delta_y = ((delta_y_red ** 2) + (delta_y_green ** 2) +
             (delta_y_blue ** 2))

  return delta_x + delta_y
```

IMPLEMENTATION:
⇨ the image is represented as a list of lists, so ***len(pixels)*** (length of the outer list) gives height and ***len(pixels)[0]*** (length of any of the inner lists) gives width
⇨ if the current pixel is an edge, we will use the current pixel and not get the surrounding pixels where the edge is. After getting the coordinates, we calculate the energy.

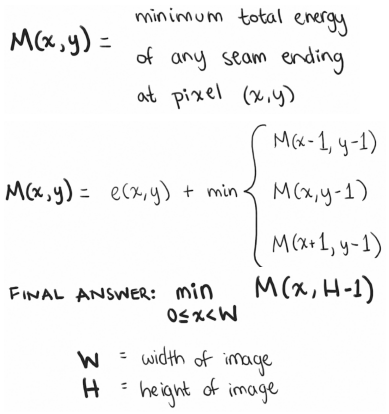
```
def compute_energy(pixels):
  energy = [[0 for pixel in row] for row in pixels]
  for y, row in enumerate(pixels):
    for x, pixel in enumerate(row):
      energy[y][x] = energy_at(pixels, x, y)
  return energy
```



⇨ Goal: minimize the total energy across all pixels in the seam when ***deciding on where a seam should be***.
⇨ But if we do not know the computations for the next pixel groups, so instead of trying to figure out where to go next, if we keep a record of energy levels, we can go backwards and decide the seam path.

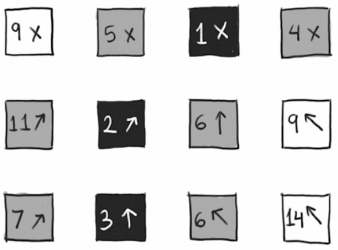


⇨ The very top pixel, ***M(x, 0)*** will have the total energy of just that pixel, ***e(x, 0)***
⇨ Beyond the first pixel, we have to ***calculate the energy of surrounding row above pixels to choose the one with the lowest seam energy*** as we move down the picture / grid of pixels.



⇨ In this problem, the subproblems are located at every pixel, because ***there is a seam ending at every pixel***.

**** See jupyter notebook for full code and explanation of this algorithm and the rest of the project, which is too long to display here.***



⇨ We will use ***back pointer***s to reconstruct the entire seam based on the seam table that was created in ***compute_vertical_seam_v1().***
⇨ At each entry in the dynamic programming table, you get the optimal values that have been computed as well as a ***back pointer*** to the choice made and you got there.
⇨ Since the algorithm goes down the image one row at a time, the y coordinate is always going to be the same. So now, we ***only need to store the x coordinate*** that applies to each iteration to get a back pointer map of the best low-energy seam in the picture.

CHAPTER 3 QUESTIONS:

⇨ **When finding vertical seams, how would you represent a back pointer?**

the x-position of the pixel we came from, The pixel we came from always has the y-position of the previous row, so only the x-position needs to be stored.

⇨ **Which of these properties of an image could not be a useful consideration in the computing the energy of an image**?

pixel brightness, Without considering the surrounding pixels, one bright pixel is considered the same whether it's a single bright spot or part of a large bright region.

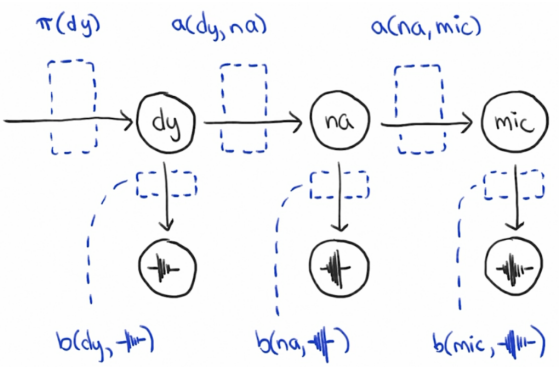
⇨ **When finding the lowest-energy seam, why do we calculate seam energies from the top of the image down to the bottom, as opposed to the other way around**?

Pixels further down the image depend on pixels above them, By performing calculations at the top of the image, the calculated values are available when they are needed below.

⇨ **Why does the greedy approach of always picking lower energy pixels sometimes fail to find the lowest energy seams**?

Choosing only lower energy pixels can get you stuck in a high energy region of the image.
Sometimes, it's better to pick a higher energy pixel in order to gain access to a low energy region of the image.

Chapter 4: Hidden Markov Model



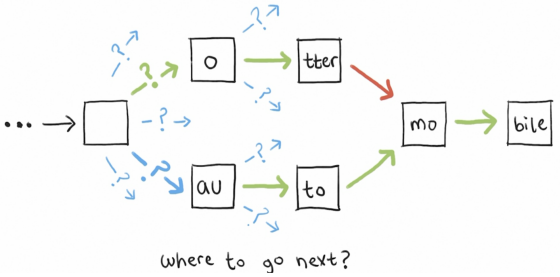
⇨ A mathematical description of real world phenomena, with hidden states, observations, and probabilities, which can be used to answer questions about the phenomena, in this case used for speech recognition

⇨ Transition probability between syllables depends only on the previous and upcoming states, no states earlier

⇨ The syllables and their sounds are the hidden states

⇨ A Hidden Markov Model is a set of possible hidden states, a set of possible observations, and the three probabilities that tie them together.

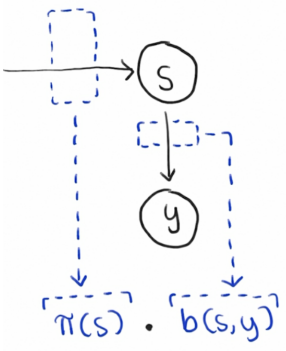
⇨ We want to calculate the most probable sequence of hidden states that resulted in the observations, i.e., the sounds observed, i.e. calculating the most probable sequence of syllables that made those sounds.



⇨ This is challenging, because you cannot just consider the sounds one or two at a time, but you have to take into account the whole. And at every syllable or event in the sequence of observations, there is a fork, and the number of possibilities grows with each fork.

⇨ Because it would be extremely time consuming and take a great deal of computational power to figure out where we are going from one syllable to the next, instead, we will ask where did we come from and use a backwards model

Viterbi Algorithm:

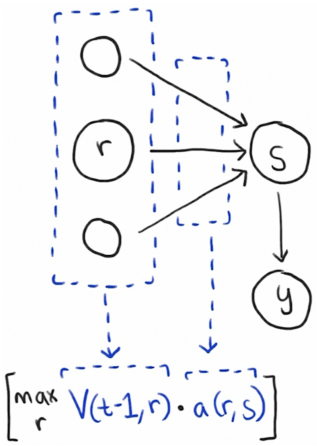


⇨ in order to predict the most probable sequence of hidden states that led to our observations, we will calculate the best path that ends with each and every state, moving ahead with each iteration, deciding which of the paths from the last iteration we want to continue

⇨ ***V(t, s)*** → the maximized probability of ending up at state ***s*** at time ***t***, if we look at just the first ***t + 1*** observations.

⇨ Starting with the initial step of ***t[0],*** there is a probability of ***π(s)*** of ending up at state ***s*** and a probability of ***b(s, y)*** of ending up at state ***y***

→ These probabilities are independent, so they can be multiplied together to get the probability of ending at state ***s*** and producing state ***y***



⇨ After the initial state, we have a series of probabilities for reaching other states we have observed.

$$V(0,s) = \pi(s) \cdot b(s,y)$$

$$V(t,s) = \left[\max_r V(t-1,r) \cdot a(r,s)\right] b(s,y)$$

⇨ We have a recurrent relationship for V with the base case and with subsequent cases.

⇨ This basically follows the same pattern we used in the algorithm for the image editing.

⇨ The greedy function takes in the Markov Model and a list of observation, going through each observation and finding the state that most likely produced that observation, based only on the probability b, not taking into consideration the initial or transitional probabilities, which is why the greedy approach will not produce accurate predictions and results.

→ For the viterbi algorithm, we need to store path probabilities, the values of our V function, but we also need to store backpointers, which state we came from to form our best path.

⇨ we'll use a list of dictionaries. As the second dimension is actually the hidden states, we have a single dictionary for each time step representing path probabilities

CHAPTER 4 QUESTIONS:

⇨ **When computing the probability of ending at a particular state during a particular time step, how many previous states does the Viterbi algorithm consider?**

Any of the states is a suitable candidate to continue from.

⇨ **Why does the greedy approach of picking the most probable hidden state for each observation not work in all cases?**

The greedy approach looks at each observation in isolation, not accounting for other sections in the sequence. Even though a particular hidden state may be more likely in isolation, in the context of the other observations, another hidden state may be more likely.

⇨ **When speech recognition is represented by a Hidden Markov Model, what are the hidden states and observations?**

Hidden states: syllables Observations: sound waves, Sound waves are directly observed, while syllables are not.

⇨ **Why must a Hidden Markov Model be trained**?

to determine the correct probabilities that will later be used for inference, Even though the probabilities can be estimated based on domain knowledge, the exact values should be learned from existing data.

⇨ **Why is pre-processing such an important part of applying Hidden Markov Models to a real-world problem?**

for reducing the raw data into a manageable number of possible observations

The raw data is already machine-readable, but it may not be suitable for a Hidden Markov Model