Dynamic Programming - Avik Das

FIBONACCI PROBLEM

```
def fib(n):
    if n == 0: return  1
    if n == 1: return  1
    return fib(n - 1) + fib(n - 2)
```

```
%time
fib(10)
```

```
CPU times: user 2 µs, sys: 0 ns, total: 2 µs
Wall time: 6.2 µs
```

```
89
```

```
%time
fib(30)
```

```
CPU times: user 3 µs, sys: 0 ns, total: 3 µs
Wall time: 8.58 µs
```

```
1346269
```

```
def fib_memo(n, cache=None):
    if n == 0:
        return 1
    if n == 1:
        return 1
    if cache is None:
        cache = {}
    if n in cache:
        return cache[n]
    result = fib_memo(n - 1, cache) + fib_memo(n - 2, cache)
    cache[n] = result
    return result
```

```
%time
print(fib_memo(100))
```

```
CPU times: user 4 µs, sys: 0 ns, total: 4 µs
Wall time: 8.82 µs
```

```
573147844013817084101
```

```
# Bottom Up Version

def fib_bottom(n):
    a = 1  # f(i - 2)
```

```
    b = 1  # f(i - 1)
    for i in range(2, n + 1):  # end of range is exclusive
      a, b = b, a + b          # the old "a" is no longer accessible after this
    return b
```

```
%time
print(fib_bottom(100))
```

```
CPU times: user 2 µs, sys: 0 ns, total: 2 µs
Wall time: 5.72 µs
573147844013817084101
```

FLOWERBOX PROBLEM

```
# Takes in an array of nutrient values.
# a and b store the results from the last two sub problems
# a and b are initialized by the two base cases

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
```

```
flowerbox([3, 5, 12, 5, 3, 8])
```

```
23
```

```
import math
```

```
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)
```

Real-World Dynamic Programming and Content-Aware Image Resizing

```
!pip install pillow
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: pillow in /usr/local/lib/python3.7/dist-packages (7.1.2)
```

```
from PIL import Image
```

The next three cells are from utils.py, which contains helper functions to read and write images using Pillow.

```
class Color:
    """
    A simple class representing an RGB value.
    """

    def __init__(self, red, green, blue):
        self.red = red
        self.green = green
        self.blue = blue

    def __repr__(self):
        return f'Color({self.red}, {self.green}, {self.blue})'

    def __str__(self):
        return repr(self)
```

```
def read_image_into_array(filename):
    """
    Read the given image into a 2D array of pixels. The result is an array,
    where each element represents a row. Each row is an array, where each
    element is a color.

    See: Color
    """

    img = Image.open(filename, 'r')
    width, height = img.size
```

```
    pixels = list(Color(*pixel) for pixel in img.getdata())
    return [pixels[n:(n + width)] for n in range(0, width * height, width)]
```

```
def write_array_into_image(pixels, filename):
    """
    Write the given 2D array of pixels into an image with the given filename.
    The input pixels are represented as an array, where each element is a row.
    Each row is an array, where each element is a color.

    See: Color
    """

    height = len(pixels)
    width = len(pixels[0])

    img = Image.new('RGB', (width, height))

    output_pixels = img.load()
    for y, row in enumerate(pixels):
        for x, color in enumerate(row):
            output_pixels[x, y] = (color.red, color.green, color.blue)

    img.save(filename)
```

The next cells are from energy.py, which is where we calculate the energy of each pixel.

energy_at()

Compute the energy of the image at the given (x, y) position.

```
    The energy of the pixel is determined by looking at the pixels surrounding
    the requested position. In the case the requested position is at the edge
    of the image, the current position is used whenever a "surrounding position"
    would go out of bounds.

    Expected return value: a single number representing the energy at that point.
```

```
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 - 1 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
```

compute_energy()

Compute the energy of the image at every pixel. Should use the `energy_at` function to actually compute the energy at any single position.

> The input is given as a 2D array of colors, and the output should be a 2D array of numbers, each representing the energy value at the corresponding position.
>
> Expected return value: the 2D grid of energy values.

```
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
```

The second step in the seam carving algorithm: finding the energy of the lowest- energy seam in an image. In this version of the algorithm, only the energy value of the seam is determined. However, this version of the algorithm still forms the basis of overall seam carving process.

compute_vertical_seam_v1()

Find the lowest-energy vertical seam given the energy of each pixel in the input image. The image energy should have been computed before by the `compute_energy` function in the `energy` module.

> This is the first version of the seam finding algorithm. You will implement the recurrence relation directly, outputting the energy of the lowest-energy seam and the x-coordinate where that seam ends.
>
> Expected return value: a tuple with two values:
>
> 1. The x-coordinate where the lowest-energy seam ends.
> 2. The total energy of that seam.

```
def compute_vertical_seam_v1(energy_data):
  min_energy_grid = [[0 for pixel in row]
                     for row in energy_data]
  height = len(energy_data)
  width = len(energy_data[0])

  for x in range(width):  # top row, just copy over
    min_energy_grid[0][x] = energy_data[0][x]

  for y in range(height):
    for x in range(width):
      x_min = x - 1 if x > 0 else 0 # loop to find min-max bounds
      x_max = x + 1 if x < width - 1 else width -1

      min_parent_energy = min(    # choosing the x with lowest
          min_energy_grid[y-1][x_candidate]
          for x_candidate in range(x_min, x_max + 1)
      )
    min_energy_grid[y][x] = energy_data[y][x] + min_parent_energy
  min_end_x, min_seam_energy = min(
      enumerate(min_energy_grid[height - 1]),
      key = lambda m: m[1]
  )
  return (min_end_x, min_seam_energy)
```

visualize_seam_end_on_image()

Draws a red box at the bottom of the image at the specified x-coordinate. This is done to visualize approximately where a vertical seam ends.

```
def visualize_seam_end_on_image(pixels, end_x):
  height = len(pixels)
  width = len(pixels[0])

  new_pixels = [[p for p in row] for row in pixels]

  min_x = max(end_x - 5, 0)
  max_x = min(end_x + 5, width - 1)

  min_y = max(height - 11, 0)
  max_y = height - 1

  for y in range(min_y, max_y + 1):
    for x in range(min_x, max_x + 1):
      new_pixels[y][x] = Color(255, 0, 0)

  return new_pixels
```

class SeamEnergyWithBackPointer

Represents the total energy of a seam along with a back pointer:

    - Stores the total energy of a seam that ends at some position in the
      image. The position is not stored because it can be inferred from where
      in a 2D grid this object appears.

    - Also stores the x-coordinate for the pixel in the previous row that led
      to this particular seam energy. This is the back pointer from which the
      entire seam can be reconstructed.

  This class is part of the second version of the vertical
  seam finding algorithm.

```
class SeamEnergyWithBackPointer:
  def __init__(self, energy, x_coordinate_in_previous_row = None):
    self.energy = energy
    self.x_coordinate_in_previous_row = x_coordinate_in_previous_row
```

compute_vertical_seam_v2()

  Find the lowest-energy vertical seam given the energy of each pixel in the
  input image. The image energy should have been computed before by the
  `compute_energy` function in the `energy` module.

  This is the second version of the seam finding algorithm. In addition to
  storing and finding the lowest-energy value of any seam, you will also store
  back pointers used to reconstruct the lowest-energy seam.

  At the end, you will return a list of x-coordinates where you would have
  returned a single x-coordinate instead.

  You may want to copy over the implementation of the first version as a starting point
  Expected return value: a tuple with two values:

    1. The list of x-coordinates forming the lowest-energy seam, starting at
       the top of the image.
    2. The total energy of that seam.

```
def compute_vertical_seam_v2(energy_data):
  min_energy_grid = [[None for pixel in row]
                     for row in energy_data]
  height = len(energy_data)
  width = len(energy_data[0])

  for x in range(width):
    min_energy_grid[0][x] = SeamEnergyWithBackPointer(energy_data[0][x])

  for y in range(1, height):
```

```
    for x in range(width):
      x_min = x - 1 if x > 0 else 0
      x_max = x + 1 if x < width - 1 else width - 1

      min_x_parent = min(    # choosing the x with lowest
          range(x_min, x_max + 1),
          key = lambda x_candidate: min_energy_grid[y-1][x_candidate].energy
      )   # Looping over x coordinate values while minimizing over energies

      min_energy_grid[y][x] = SeamEnergyWithBackPointer(
          energy_data[y][x] + min_energy_grid[y-1][min_x_parent].energy,
          min_x_parent)

  min_end_x = min(enumerate(min_energy_grid[height - 1]),
  key = lambda m: m[1].energy)[0]
  seam_energy = min_energy_grid[-1][min_end_x].energy
  # last row, picking min energy x value to back-build seam

  seam_xs = []
  last_x = min_end_x
  for y in range(height - 1, -1, -1): # looping backwards, reconstructing seam
    seam_xs.append(last_x)
    last_x = min_energy_grid[y][last_x].x_coordinate_in_previous_row

  seam_xs.reverse()

  return(seam_xs, seam_energy)

  return (min_end_x, min_seam_energy)
```

```
def visualize_seam_on_image(pixels, seam_xs):
    """
    Draws a red line on the image along the given seam. This is done to
    visualize where the seam is.

    This is NOT one of the functions you have to implement.
    """

    height = len(pixels)
    width = len(pixels[0])

    new_pixels = [[p for p in row] for row in pixels]

    for y, seam_x in enumerate(seam_xs):
        min_x = max(seam_x - 2, 0)
        max_x = min(seam_x + 2, width - 1)

        for x in range(min_x, max_x + 1):
            new_pixels[y][x] = Color(255, 0, 0)

    return new_pixels
```

```
def run_this(input_name, output_name):

  input_filename = input_name
  output_filename = output_name

  print(f'Reading {input_filename}...')
  pixels = read_image_into_array(input_filename)

  print('Computing the energy...')
  energy_data = compute_energy(pixels)

  print('Finding the lowest-energy seam...')
  seam_xs, min_seam_energy = compute_vertical_seam_v2(energy_data)

  print(f'Saving {output_filename}')
  visualized_pixels = visualize_seam_on_image(pixels, seam_xs)
  write_array_into_image(visualized_pixels, output_filename)

  print()
  print(f'Minimum seam energy was {min_seam_energy} at x = {seam_xs[-1]}')
```

```
run_this("/content/surfer.jpg", "pic.jpg")
```

```
Reading /content/surfer.jpg...
Computing the energy...
Finding the lowest-energy seam...
Saving pic.jpg

Minimum seam energy was 39703 at x = 1223
```

remove_seam_from_image()

Remove pixels from the given image, as indicated by each of the x-coordinates in the input. The x-coordinates are specified from top to bottom and span the entire height of the image.

Expected return value: the 2D grid of colors. The grid will be smaller than the input by one element in each row, but will have the same number of rows.

```
def remove_seam_from_image(image, seam_xs):
  new_pixels = [
      [
          p for x, p in enumerate(row)
          if x != seam_xs[y]
      ]
      for (y, row) in enumerate(image)
  ]
  return new_pixels
```

remove_n_lowest_seams_from_image()

```
Iteratively:

1. Find the lowest-energy seam in the image.
2. Remove that seam from the image.

Repeat this process `num_seams_to_remove` times, so that the resulting image
has that many pixels removed in each row.

Expected return value: the 2D grid of colors. The grid will be smaller than the input
`num_seams_to_remove` elements in each row, but will have the same number of
rows.
```

```
def remove_n_lowest_seams_from_image(image, num_seams_to_remove):

    for i in range(num_seams_to_remove):
      print(f'Removing seam {i + 1}/{num_seams_to_remove}')

      print('  Computing energy...')
      energy_data = compute_energy(image)
      print('  Finding the lowest-energy seam...')
      seam_xs, _ = compute_vertical_seam_v2(energy_data)

      print(f'  Saving intermediate result to intermediate-{i}.png...')
      visualized_pixels = visualize_seam_on_image(image, seam_xs)
      write_array_into_image(visualized_pixels, f'intermediate-{i}.png')

      print('  Removing the lowest-energy seam...')
      image = remove_seam_from_image(image, seam_xs)

    return image
```

```
def do_all_the_things(input_image, num_seams_to_remove, output_image):

    input_filename = input_image
    num_seams_to_remove = num_seams_to_remove
    output_filename = output_image

    print(f'Reading {input_filename}...')
    pixels = read_image_into_array(input_filename)

    print(f'Saving {output_filename}')
    resized_pixels = \
        remove_n_lowest_seams_from_image(pixels, num_seams_to_remove)
    write_array_into_image(resized_pixels, output_filename)
```

CAUTION: Avik Das says it takes about 90 min to remove about 100 seams. I am gonna say this is for educational purposes only and not run it. Lawd!

```
# do_all_the_things("/content/surfer.jpg", 50, "image_50px.jpg")
```

Hidden Markov Model

```
class HMM:
    """
    A representation of a Hidden Markov Model, consisting of:

      - A set of possible states
      - A set of possible observations
      - The initial probabilities for the states.
      - The transition probabilities between the states.
      - The emission probabilities for the states and observations.
    """

    def __init__(self, p, a, b):
        self.p = p
        self.a = a
        self.b = b

        # For convenience. This way, the definition of the HMM doesn't need to
        # specify all the states separately.
        #
        # Because our algorithms don't need to loop through all the possible
        # observations, we don't need to store the possible observations.
        self.possible_states = p.keys()
```

```

SPEECH_RECOGNITION_HMM = HMM(
    # Initial state probabilities. We can start with 'au' or 'o' with equal
    # probability. All the other syllables only appear in the middle of words.
    {'au': 0.5, 'to': 0, 'o': 0.5, 'tter': 0, 'mo': 0, 'bile': 0},

    # State transition matrix. In this example, we can only form the words
    # 'automobile' and 'otter', and nothing else.
    #
    # For example, 'au' can only be followed by 'to', which can only be
    # followed by 'mo', and so on. Both 'tter' and 'bile' will end their
    # respective words because they can't be followed by anything.
    {
        'au':   {'au': 0, 'to': 1, 'o': 0, 'tter': 0, 'mo': 0, 'bile': 0},
        'to':   {'au': 0, 'to': 0, 'o': 0, 'tter': 0, 'mo': 1, 'bile': 0},

        'o':    {'au': 0, 'to': 0, 'o': 0, 'tter': 1, 'mo': 0, 'bile': 0},
        'tter': {'au': 0, 'to': 0, 'o': 0, 'tter': 0, 'mo': 0, 'bile': 0},

        'mo':   {'au': 0, 'to': 0, 'o': 0, 'tter': 0, 'mo': 0, 'bile': 1},
        'bile': {'au': 0, 'to': 0, 'o': 0, 'tter': 0, 'mo': 0, 'bile': 0},
    },
```

```
# Observation probability matrix.
#
# 'au' can produce both 'sound-au' and 'sound-o', but produces the former
# with higher probability. The same principle applies to 'to', 'o' and
# 'tter', each of which can produce one of two sounds with different
# probabilities.
#
# 'mo' and 'bile' are unambiguous, as each can only produce a single sound.
{
    'au': {
        'sound-au': 0.7,
        'sound-o': 0.3,
        'sound-to': 0,
        'sound-tter': 0,
        'sound-mo': 0,
        'sound-bile': 0
    },

    'to': {
        'sound-au': 0,
        'sound-o': 0,
        'sound-to': 0.7,
        'sound-tter': 0.3,
        'sound-mo': 0,
        'sound-bile': 0
    },

    'o': {
        'sound-au': 0.3,
        'sound-o': 0.7,
        'sound-to': 0,
        'sound-tter': 0,
        'sound-mo': 0,
        'sound-bile': 0
    },

    'tter': {
        'sound-au': 0,
        'sound-o': 0,
        'sound-to': 0.3,
        'sound-tter': 0.7,
        'sound-mo': 0,
        'sound-bile': 0
    },

    'mo': {
        'sound-au': 0,
        'sound-o': 0,
        'sound-to': 0,
        'sound-tter': 0,
        'sound-mo': 1,
```

```
            'sound-bile': 0
        },

        'bile': {
            'sound-au': 0,
            'sound-o': 0,
            'sound-to': 0,
            'sound-tter': 0,
            'sound-mo': 0,
            'sound-bile': 1
        }
    }
)
```

```
def greedy(hmm, observations):
    """
    Given a Hidden Markov Model and a sequence of algorithm, finds the most
    probable sequence of states that produced the observations.

    Uses the greedy algorithm, looking at each observation in isolation and
    ignoring all transition probabilities.
    """

    return [
        max(hmm.possible_states, key=lambda s: hmm.b[s][observation])
        for observation in observations
    ]
```

```
class PathProbabilityWithBackPointer:
    """
    Represents the probability of a path along with a back pointer:

      - Stores the probability of a path ending at a given state at a given
        time. The state and time are not stored because they can be inferred
        from where in the 2D grid this object appears.

      - Also stores the state in the previous time step that led to this
        particular path probability. This is the back pointer from which the
        entire path can be reconstructed.
    """

    def __init__(self, probability, previous_state=None):
        self.probability = probability
        self.previous_state = previous_state
```

```
def viterbi(hmm, observations):
    """
    Given a Hidden Markov Model and a sequence of algorithm, finds the most
    probable sequence of states that produced the observations.
```

```
    Uses the Viterbi algorithm, which incorporates not only the emission
    probabilities, but also the initial and transition probabilities in order
    to construct a realistic path.
    """

    v_grid = [{} for _ in observations]

    for s in hmm.possible_states:
        v_grid[0][s] = PathProbabilityWithBackPointer(
            hmm.p[s] * hmm.b[s][observations[0]]
        )

    for t in range(1, len(observations)):
        for s in hmm.possible_states:
            possible_transition_probabilities = [
                (
                    v_grid[t - 1][r].probability * hmm.a[r][s],
                    r
                )
                for r in hmm.possible_states
            ]

            max_transition_probability, best_previous_state = max(
                possible_transition_probabilities,
                key=lambda probability_and_state: probability_and_state[0]
            )

            v_grid[t][s] = PathProbabilityWithBackPointer(
                max_transition_probability * hmm.b[s][observations[t]],
                best_previous_state
            )

    max_end_state = max(
        hmm.possible_states,
        key=lambda s: v_grid[-1][s].probability
    )

    path_states = []
    last_state = max_end_state
    for t in range(len(observations) - 1, -1, -1):
        path_states.append(last_state)
        last_state = v_grid[t][last_state].previous_state

    path_states.reverse()

    return path_states
```

```
    sound_samples = [
        # Technically, the speaker might have said 'auto', but based on the
        # sounds, it's more likely they said 'otter'. Both the greedy and the
        # Viterbi algorithms will agree.
```

```
        ['sound-o', 'sound-tter'],

        # Here, the speaker definitely said 'automobile', but only the Viterbi
        # algorithm can infer that.
        ['sound-o', 'sound-tter', 'sound-mo', 'sound-bile'],
    ]

    for sample in sound_samples:
        greedy_path = greedy(SPEECH_RECOGNITION_HMM, sample)
        viterbi_path = viterbi(SPEECH_RECOGNITION_HMM, sample)

        print(f"SAMPLE:  {'  '.join(sample)}")
        print(f"  GREEDY :  {'  '.join(greedy_path)}")
        print(f"  VITERBI:  {'  '.join(viterbi_path)}")
        print()
```

```
SAMPLE:  sound-o  sound-tter
  GREEDY :  o  tter
  VITERBI:  o  tter

SAMPLE:  sound-o  sound-tter  sound-mo  sound-bile
  GREEDY :  o  tter  mo  bile
  VITERBI:  au  to  mo  bile
```

```
!pip install jovian --upgrade -q
```

```
     |████████████████████████████████| 68 kB 3.3 MB/s eta 0:00:011
  Building wheel for uuid (setup.py) ... done
```

```
import jovian
```

```
jovian.set_project('dynamic_programming_with_avik_das')
jovian.set_colab_id('dynamic_programming')
```

```
jovian.commit()
```

```
[jovian] Detected Colab notebook...
[jovian] Uploading colab notebook to Jovian...
---------------------------------------------------------------------------
ApiError                                  Traceback (most recent call last)
<ipython-input-154-7f2017c1a3df> in <module>
----> 1 jovian.commit()

/usr/local/lib/python3.7/dist-packages/jovian/utils/commit.py in commit(message, files, outputs, environment, privacy, filename, project, new_project, git_commit, git_message, **kwargs)
    140             return
```

```
    141
--> 142         res = perform_colab_commit(project, privacy)
    143         slug, username, version, title = res['slug'], res['owner']['username'],
res['version'], res['title']
    144

/usr/local/lib/python3.7/dist-packages/jovian/utils/colab.py in
perform_colab_commit(project, privacy)
     48             log(warning, error=True)
     49         return data
---> 50     raise ApiError('Colab commit failed: ' + pretty(res))

ApiError: Colab commit failed: (HTTP 404) No file found for the provided File ID
```