1) Dynamic Programming

a) Calculate energy map

sobel_energy converts the image to grayscale (if needed), applies Sobel filters in x/y, and sums the absolute gradients to produce an energy map where edges/high-contrast pixels have higher cost.

```
def sobel_energy(img: np.ndarray) -> np.ndarray:
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY) if img.ndim == 3 else img
    gx = cv2.Sobel(gray, cv2.CV_32F, 1, 0, ksize=3)
    gy = cv2.Sobel(gray, cv2.CV_32F, 0, 1, ksize=3)
    return np.abs(gx) + np.abs(gy)
```

b) Find minimum/optimal seam

min_vertical_seamdp uses forward energy, which calculates the cost of creating NEW edges when a seam is removed, rather than just using existing gradient energy. This produces better visual results by minimizing artifacts.

Key components:

- Takes both cost (energy map) and img (original image) as parameters
- Converts image to grayscale for forward energy computation
- Uses space optimization: only 2 rows of cumulative cost matrix M instead of full H×W
- Stores full predecessor matrix P for backtracking

Forward energy costs computed per pixel:

- c_base = |I(i, j+1) - I(i, j-1)| (cost of joining left/right neighbors)
- c_l = c_base + |I(i-1, j) - I(i, j-1)| (cost from upper-left)
- c_u = c_base (cost from directly above)
- c_r = c_base + |I(i-1, j) - I(i, j+1)| (cost from upper-right)

```
def min_vertical_seamdp(cost: np.ndarray, img: np.ndarray):
    H, W = cost.shape
    M = np.zeros((2, W), dtype=np.float32)  # Space-optimized: only 2 rows
    P = np.zeros((H, W), dtype=np.int8)     # Predecessor matrix

    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY).astype(np.float32)
    M[0, :] = 0

    for i in range(1, H):
        prev = (i - 1) % 2
        curr = i % 2
        for j in range(W):
            # Compute c_l, c_u, c_r based on forward energy
            # Choose minimum: M[curr, j] = min(M[prev, j±1/j] + cost)
            # Store direction in P[i, j]
    # Backtrack from minimum in last row
    j = int(np.argmin(M[(H-1) % 2]))
    seam = [j]
    for i in range(H-1, 0, -1):
        j = j + int(P[i, j])
        seam.append(j)
    seam.reverse()
    return seam
```

c) Remove seam and repeat

carve_height_once demonstrates the complete seam carving cycle: compute energy → find seam → remove seam. It uses the transpose trick to reuse vertical seam code for horizontal carving:

```
def carve_height_once():
    tr = np.transpose(STATE["img"], (1, 0, 2))  # Transpose to reuse vertical code
    E = sobel_energy(tr)
    seam = min_vertical_seamdp(E, img=tr)
    tr = remove_vertical_seamdp(tr, seam)
    STATE["img"] = np.transpose(tr, (1, 0, 2))  # Transpose back
```

The same pattern applies to vertical carving:

```
def carve_width_once():
    E = sobel_energy(STATE["img"])
    seam = min_vertical_seamdp(E, img=STATE["img"])
    STATE["img"] = remove_vertical_seamdp(STATE["img"], seam)
```

2) Question

2ai) Show that the number of such possible seams grows at least exponentially in m, assuming that n > 1.

From the first row, there are n possible starting pixel positions for each seam. For each subsequent row up to row m, there are 2 (n=2) or 3 (n>2) possible successive pixels to pick to form a seam. Thus, the total number of possible seams can be computed as follows.

$$Total\ Possible\ Seams\ =\ col\ *\ (successive\ possible\ pixel\ per\ row)^{row-1}$$

Case 1: n=2

$$Total\ Complexity(possible\ seams)\ =\ n\ *\ 2^{m-1}$$

Case 2: n>2

$$Total\ Complexity(possible\ seams)\ =\ n\ *\ 3^{m-1}$$

From the total number of possible seams, we see that the number of possible seams grows at least (n=2) exponentially in m for any n > 1.

2aii) Suppose now that along with each pixel img[i, j], we have calculated a real valued energy or disruption measure e[i,j], indicating how disruptive or energy effective it would be to remove pixel img[i,j]. Intuitively, the lower a pixel's disruption measure (energy), the more similar the pixel is to its neighbors. Suppose further that we define the disruption measure of a seam to be the sum of the disruption measures of its pixels. In other words, how to relate to subproblems?

Since the disruption measure of any pixel is the total minimum energy of its successive pixels in the path of the least disruptive seam from that pixel. Suppose a pixel at (i,j), the solution to the disruption measure(sum) for that pixel can be computed as follows.

$$\Sigma MinEnergy\,[i,\ j]\ =\ E[i,\ j]\ +\ min(\Sigma MinEnergy\,[i\ +\ 1,\ j\ -\ 1],\ \Sigma MinEnergy\,[i\ +\ 1,\ j],\Sigma MinEnergy\,[i\ +\ 1,\ j\ +\ 1])$$

From the solution to the disruption measure(sum) for any pixel, we can see that pixels have overlapping subproblems with its successive pixels. Using dynamic programming, we can

reduce the cost of recomputing the overlapping subproblems through memoization. This reduces the total complexity from an exhaustive search(O(m*n*3^m-1)) for the sum of minimum energy into O(mn) with a space complexity of O(mn)

2b) Analysis of Divide and Conquer method for computation of convolution filters

Divide and conquer can theoretically improve the efficiency of the computation of filters such as the Sobel filter by breaking the kernel into the product of 2 1D filters on the row and column. When compared to the naive computation of Sobel filter, the complexity(multiplications) of the naive operation is O(3^2) while the divide and conquer method is O(2*3). Theoretically, the divide and conquer method has better efficiency in terms of complexity. However, in practice, the performance of the naive operation of sobel filter may outperform the divide and conquer method as the kernel size is small 3x3 and the cost of recursion outweighs the better time complexity.

3) Greedy Algorithm

From our testing, we found that the greedy algorithm would prematurely cut off the human on the left of the image as it is unable to see the future (right side has low energy and should cut it first).

We tried looking per row to create seams, but that only made the compute time longer, losing the benefit of greedy and still suffering from the shortcoming of greedy, which is unable to see the bigger picture.

Dual gradient energy, which involves looking at Sobel energy values around the point we're at to make a decision helps to delay the inevitable cutting of the human, which can be attributed to it having slightly more information in context, slightly alleviating the inherent downside of greedy

Dynamic Programming vs Greedy Algorithm

Approach	Time Complexity	Space Complexity	Correctness
Dynamic Programming	O(mn)	O(n + m) (store 2 rows)	Global optimum
Greedy Algorithm	O(m)	O(m) (seam)	Local optimum

Greedy algorithm runs significantly faster than DP as well, matching our expected results from analyzing the time complexity.