Lab 10

Problem 1

You are tasked with helping a maintenance worker move between floors in a building. Each floor of the building has a cost associated with moving to it. The worker can either go up one floor at a time or skip one floor and go up two floors at once.

You need to determine the minimum cost for the worker to reach the top of the building, given that they can either start at the first floor or the second floor.

Input:

- A list `cost` of integers, where `cost[i]` is the cost for moving to floor `i`. The worker can either move one floor up (paying the cost of the current floor) or skip one floor and go two floors up (paying the cost of the skipped floor and the next one).

Output:

- The minimum total cost for the worker to reach the top of the building.

Task:

1. Explain how you would use dynamic programming to calculate the minimum cost for the worker to reach the top of the building.
2. With the given example `cost = [1,100,1,1,1,100,1,1,100,1]`, explain step-by-step how the dynamic programming approach works to calculate the minimum cost of reaching the top.

<u>Answer</u>

Algorithm Approach

To solve this problem efficiently, dynamic programming (DP) is an ideal approach because the problem involves optimal substructure and overlapping subproblems. Let's break it down:

Why Dynamic Programming?

1. **Subproblems:**
   The problem involves calculating the minimum cost to reach the top of the building, but instead of calculating the total cost in one go, we can break it into smaller steps. The worker can either:

    - Move one floor up and pay the cost of the current floor, or
    - Skip a floor and move two floors up, paying the cost of the skipped floor and the next one.

2. For each floor, we need to make a decision based on the costs of the two previous floors, so each floor's minimum cost depends on the minimum costs of the two previous floors.

3. **Optimal Substructure:**
   The problem exhibits optimal substructure, meaning that the minimum cost to reach a given floor is the minimum of the costs to reach the previous two floors. This allows us to break the problem into smaller subproblems that are easier to solve.

4. **Overlapping Subproblems:**
   The cost to reach any given floor will be recalculated multiple times if solved naively. By using dynamic programming, we can store the results of each subproblem (the minimum cost to reach each floor) in the `cost` array, which prevents redundant calculations and improves efficiency.

Dynamic Programming Approach:

1. **Initialization:**
   We start by defining the `cost[]` array, where `cost[i]` will store the minimum cost to reach floor `i`. This is done in-place, so we update the original `cost[]` array.
2. **Recurrence Relation:**
   For each floor `i` (starting from the 2nd step), the minimum cost can be computed as:
   `cost[i] += min(cost[i-1], cost[i-2])`

   This means that the cost to reach floor `i` is the current floor's cost `cost[i]` plus the minimum of the costs to reach the previous two floors.

3. **Final Step:**
   The worker can either reach the top from the last floor (`cost[n-1]`) or from the second-to-last floor (`cost[n-2]`). Thus, the final result is:
   `min(cost[n-1], cost[n-2])`

Step by step example

Let's walk through the dynamic programming approach with the example `cost = [1,100,1,1,1,100,1,1,100,1]`:

1. **Initialization:**

- Start by initializing the `cost` array, where `cost[i]` represents the minimum cost to reach the `i`th floor. The `cost[]` array will be updated in-place.
  `cost = [1, 100, 1, 1, 1, 100, 1, 1, 100, 1]`

2. **Base Cases:**
   - The cost of the first floor (`cost[0]`) is already given as `1`.
   - The cost of the second floor (`cost[1]`) is `100`, as given.

3. **Iterating through the floors and updating the cost array:**
   - **Floor 2 (i=2):** The worker can either come from floor 1 or floor 0. The cost will be:
     `cost[2] += min(cost[1], cost[0])  # cost[2] = 1 + min(100, 1) = 2`

     Updated `cost` array: `[1, 100, 2, 1, 1, 100, 1, 1, 100, 1]`

   - **Floor 3 (i=3):** The worker can either come from floor 2 or floor 1:
     `cost[3] += min(cost[2], cost[1])  # cost[3] = 1 + min(2, 100) = 3`

     Updated `cost` array: `[1, 100, 2, 3, 1, 100, 1, 1, 100, 1]`

   - **Floor 4 (i=4):** The worker can either come from floor 3 or floor 2:
     `cost[4] += min(cost[3], cost[2])  # cost[4] = 1 + min(3, 2) = 3`

     Updated `cost` array: `[1, 100, 2, 3, 3, 100, 1, 1, 100, 1]`

   - **Floor 5 (i=5):** The worker can either come from floor 4 or floor 3:
     `cost[5] += min(cost[4], cost[3])  # cost[5] = 100 + min(3, 3) = 103`

     Updated `cost` array: `[1, 100, 2, 3, 3, 103, 1, 1, 100, 1]`

   - **Floor 6 (i=6):** The worker can either come from floor 5 or floor 4:
     `cost[6] += min(cost[5], cost[4])  # cost[6] = 1 + min(103, 3) = 4`

     Updated `cost` array: `[1, 100, 2, 3, 3, 103, 4, 1, 100, 1]`

   - **Floor 7 (i=7):** The worker can either come from floor 6 or floor 5:
     `cost[7] += min(cost[6], cost[5])  # cost[7] = 1 + min(4, 103) = 5`

Updated `cost` array: `[1, 100, 2, 3, 3, 103, 4, 5, 100, 1]`

- **Floor 8 (i=8):** The worker can either come from floor 7 or floor 6:
  `cost[8] += min(cost[7], cost[6])  # cost[8] = 100 + min(5, 4) = 104`

  Updated `cost` array: `[1, 100, 2, 3, 3, 103, 4, 5, 104, 1]`

- **Floor 9 (i=9):** The worker can either come from floor 8 or floor 7:
  `cost[9] += min(cost[8], cost[7])  # cost[9] = 1 + min(104, 5) = 6`

  Updated `cost` array: `[1, 100, 2, 3, 3, 103, 4, 5, 104, 6]`

4. **Final Step:** After processing all the floors, the minimum cost to reach the top of the building is:
   `min(cost[9], cost[8])  # min(6, 104) = 6`

   The minimum cost to reach the top is **6**.

Problem 2

You are working on a network monitoring tool that tracks the number of active devices in a network at any given time. Each device in the network has a unique identifier, and the identifiers are represented as integers. These identifiers can be seen as a sequence of numbers, where each number's binary representation shows which devices are active at specific time intervals.

Your task is to write a function that, for each integer from `0` to `n`, returns the number of devices that are active, which corresponds to the number of `1`s in the binary representation of each number.

The function should generate a list where each index `i` (from `0` to `n`) tells you how many devices are active (i.e., how many `1`s are in the binary representation of `i`).

Input:

- An integer `n` which represents the highest device identifier you want to track.

Output:

- A list `ans` of length `n + 1`, where `ans[i]` is the number of active devices (the number of `1`s) in the binary representation of the number `i`.

Example:

Input: `n = 2`

Output: `[0, 1, 1]`

Explanation:

- The binary representation of 0 is `0`, which has 0 `1`s (no active devices).
- The binary representation of 1 is `1`, which has 1 `1` (one active device).
- The binary representation of 2 is `10`, which has 1 `1` (one active device).

Task:

1. How can you solve this problem efficiently using dynamic programming that runs in **O(n)** time?
2. For the given example `n = 5`, explain how you would calculate the binary representations of each number from `0` to `5` and count the number of `1`s in each binary number to build the final output array.

<u>Answer</u>

How can you solve this problem efficiently in O(n) time? Explain how to reach that approach.

We are asked to count how many `1`s appear in the binary representation of each number from `0` to `n`. A straightforward way is to convert each number to binary and count the `1`s one by one. However, that takes more time—about `O(n log n)` because converting to binary and counting takes log time per number.

To make it faster, we can use **dynamic programming** by recognizing a simple pattern in binary numbers.

Let's think step-by-step:

1. **Observe the pattern**:
    - Every time you divide a number by 2 (which in binary means shifting bits to the right), you remove the least significant bit (LSB).
    - If the number is **even**, that bit is `0`, so the number of `1`s stays the same.

  - If the number is **odd**, that bit is `1`, so the number of `1`s is **1 more** than its half.

2. **Example**:
   - `2` in binary is `10` → same number of `1`s as `1` (`1` in binary is `1`)
   - `3` in binary is `11` → one more `1` than `1`
   - `5` is `101` → one more `1` than `2` (`10`)

   So we see:
   ```
   countBits(2) = countBits(1) → because 2 is even
   countBits(3) = countBits(1) + 1 → because 3 is odd
   countBits(4) = countBits(2) → even
   countBits(5) = countBits(2) + 1 → odd
   ```

3. **From this pattern**, we can define the formula:
   `countBits(i) = countBits(i // 2) + (i % 2)`
   - If `i` is **even** → `i % 2 = 0`, so it reuses the same count as `i // 2`
   - If `i` is **odd** → `i % 2 = 1`, so we add one more to the count of `i // 2`

4. **Dynamic programming idea**:

   - We build an array `t` where `t[i]` stores the number of `1`s for number `i`
   - Start from `t[0] = 0` and build up to `t[n]` using the formula above
   - Since we reuse already computed values, this gives us **O(n)** time complexity

Using the input `n = 5`, explain step-by-step how to calculate the result.

Let's build the result step-by-step using the rule:

`t[i] = t[i // 2] + (i % 2)`

We initialize `t[0] = 0` and go up to `t[5]`:

- `t[0] = 0` (binary `0`)

- `t[1] = t[0] + 1 = 0 + 1 = 1` (binary `1`)

- `t[2] = t[1] + 0 = 1 + 0 = 1` (binary `10`)

- `t[3] = t[1] + 1 = 1 + 1 = 2` (binary `11`)

- `t[4] = t[2] + 0 = 1 + 0 = 1` (binary `100`)
- `t[5] = t[2] + 1 = 1 + 1 = 2` (binary `101`)

So the final result is:

```
[0, 1, 1, 2, 1, 2]
```

Problem 3

Imagine you are programming a robot to move across a factory floor. The floor is divided into checkpoints, and each checkpoint indicates how far the robot can jump forward. The robot starts at the first checkpoint and aims to reach the last one.

Each checkpoint has a number that represents the **maximum steps** the robot can jump from there. The robot can jump from 1 up to the maximum number of steps allowed at each checkpoint. You need to determine if the robot can reach the last checkpoint starting from the first one.

Input:

- A list `nums` of integers where `nums[i]` is the maximum number of steps the robot can take from checkpoint `i`.

Output:

- A boolean value: `True` if the robot can reach the last checkpoint, otherwise `False`.

Task:

1. Explain how you would solve this problem using dynamic programming.
2. With the given examples `nums = [2, 3, 1, 1, 4]` and `nums = [3, 2, 1, 0, 4]`, explain step-by-step how the approach works to determine if the robot can reach the last checkpoint.

Answer

This problem is about determining if a robot can reach the last checkpoint, starting from the first, where each checkpoint specifies how far the robot can jump.

Why Use Dynamic Programming?

1. **Optimal Substructure**:
   The ability to reach a checkpoint depends on whether you can reach any earlier

checkpoint. Each subproblem (checkpoint) contributes to the overall solution (reaching the last checkpoint).

2. **Overlapping Subproblems**:
   Instead of recalculating whether each checkpoint is reachable multiple times, we store the farthest index (`max_reach`) the robot can reach at any point. This avoids redundant work, similar to DP where results of subproblems are stored and reused.

How It Relates to Greedy:

- **Greedy Aspect**:
  At each checkpoint, the robot chooses to jump the maximum possible steps. This is a greedy choice that aims to reach the farthest point, maximizing the chance of reaching the last checkpoint.

- **DP Aspect**:
  While we don't use a traditional DP table, we keep track of the maximum reachable index (`max_reach`), which acts like storing subproblem results. By updating `max_reach` as we go, we avoid recomputing reachability for each checkpoint.

How the Code Works:

In the code, we track the farthest point the robot can reach using the variable `max_reach`. Let's break down the approach:

1. **Initialization**:
   We start with `max_reach = 0`, meaning we begin at the first checkpoint.
2. **Iterate through the list**: We loop through each checkpoint (`i`), and at each checkpoint, we check if it is reachable:
   - If `i > max_reach`, it means we can't reach this checkpoint, so we return `False`.
3. **Update `max_reach`**:
   At each checkpoint `i`, we update `max_reach`:
   - `max_reach = max(max_reach, i + nums[i])`: This line means that from the current checkpoint `i`, the robot can jump as far as `i + nums[i]`, and we update `max_reach` to reflect the farthest point we can now reach.
4. **Check if we can reach the last checkpoint**:
   If `max_reach >= len(nums) - 1`, it means we can reach or exceed the last checkpoint, so we return `True`.
5. **Final Return**:
   If after iterating through all the checkpoints we never reach the last index, we return `False`.

Step-by-step Explanation Using the Given Examples `nums = [2, 3, 1, 1, 4]` and `nums = [3, 2, 1, 0, 4]`:

Example 1: `nums = [2, 3, 1, 1, 4]`

1. **Initialization**:
    - `max_reach = 0` (starting at the first checkpoint).
2. **Step 1 (i = 0)**:
    - The robot is at checkpoint 0, and it can jump 2 steps.
    - Update `max_reach = max(0, 0 + nums[0]) = 2`.
    - The robot can now reach up to checkpoint 2.
3. **Step 2 (i = 1)**:
    - The robot is at checkpoint 1, and it can jump 3 steps.
    - Update `max_reach = max(2, 1 + nums[1]) = 4`.
    - The robot can now reach up to checkpoint 4, which is the last checkpoint.
4. **Early Termination**:
    - Since `max_reach = 4` is greater than or equal to the last index, return `True`.

Example 2: `nums = [3, 2, 1, 0, 4]`

1. **Initialization**:
    - `max_reach = 0` (starting at the first checkpoint).
2. **Step 1 (i = 0)**:
    - The robot is at checkpoint 0, and it can jump 3 steps.
    - Update `max_reach = max(0, 0 + nums[0]) = 3`.

- The robot can now reach up to checkpoint 3.

3. **Step 2 (i = 1)**:
    - The robot is at checkpoint 1, and it can jump 2 steps.
    - Update `max_reach = max(3, 1 + nums[1]) = 3`.
    - The robot can still reach up to checkpoint 3.
4. **Step 3 (i = 2)**:
    - The robot is at checkpoint 2, and it can jump 1 step.
    - Update `max_reach = max(3, 2 + nums[2]) = 3`.
    - The robot can still reach up to checkpoint 3.
5. **Step 4 (i = 3)**:
    - The robot is at checkpoint 3, and it can jump 0 steps.
    - Since `i = 3` is the farthest the robot can reach and `max_reach` is 3, the robot is stuck and cannot move forward.
6. **Final Check**:
    - Since `max_reach` never exceeds or equals the last index (`4`), return `False`.

Problem 4

You are managing a rectangular garden represented as a 2D grid. Each cell in the grid is either:

- `1` (indicating that the plot is **healthy and green**), or
- `0` (indicating a **damaged or unusable** plot).

Your goal is to find out how many **square plots** (subgrids) of any size can be formed such that **all cells in the square are green (1)**.

A square plot must be fully green — all cells in it must be 1s — and can be of any size `1×1`, `2×2`, ..., up to the size that fits within the garden.

Example:

Input:

```
garden = [
  [0, 1, 1, 1],
  [1, 1, 1, 1],
  [0, 1, 1, 1]
]
```

Output: `15`

Explanation:

- There are 10 squares of size 1×1 (each individual 1).
- 4 squares of size 2×2.
- 1 square of size 3×3.
- Total = **10 + 4 + 1 = 15**

Task:

1. How would you solve this problem using dynamic programming?
2. Use the example `garden = [[1, 0, 1], [1, 1, 0], [1, 1, 0]]` to walk through how the DP table is built step-by-step and how the final result is calculated.

Answer

Understanding the Common Approach

Let's begin by understanding what we're trying to do.

We are given a **binary matrix** — that is, a 2D list of `0`s and `1`s — for example:

```
matrix = [
  [1, 1, 1],
  [1, 1, 1],
  [0, 1, 1]
]
```

Each cell represents a tile of land (or garden plot, image pixel, etc.). A `1` means the cell is *usable*, and a `0` means it is *not usable*.

Our Goal

We want to count how many **square submatrices** (small square-shaped blocks) we can find in the matrix such that **all elements in the square are 1**.

These squares can be of size:

- 1×1 (single cell),
- 2×2 (block of 4),
- 3×3 (block of 9), etc.

What does it mean for a square to "end at (i, j)"?

This is the key idea in the DP approach, so let's clarify:

Example:

If we say **a square ends at position `(2, 2)`**, we mean:

- The **bottom-right corner** of the square is at cell `(2, 2)` (i.e., `matrix[2][2]`).
- The rest of the square stretches **upwards** and **to the left** from that cell.

For example, a 2×2 square ending at `(2, 2)` looks like this:

```
[1, 1]
[1, 1]  ← ends here at (2, 2)
```

This square includes:

- `(1, 1)`, `(1, 2)`
- `(2, 1)`, `(2, 2)`

So we say: **a square of size 2×2 ends at (2, 2)**.

Core DP Idea

We define a DP table where:

> `dp[i][j]` represents the **size of the largest square submatrix of all 1s that ends at cell (i, j)`**.

This square must include cell `(i, j)` as the **bottom-right corner**.

Example:

Let's say:

```
dp[2][2] = 3
```

This means: there is a **3×3 square** of all 1s that ends at `matrix[2][2]`.

How do we build the DP table?

We use the rule:

```
if matrix[i][j] == 1:
    dp[i][j] = 1 + min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1])
else:
    dp[i][j] = 0
```

Why these three neighbors?

To form a square ending at `(i, j)`, you need:

- The cell **above**: `dp[i-1][j]` (same column, one row up)
- The cell **to the left**: `dp[i][j-1]` (same row, one column left)
- The **top-left diagonal** cell: `dp[i-1][j-1]`

These three cells determine the **maximum possible square size** you can extend to `(i, j)`.

How to get the total number of squares?

Once we've built the DP table:

- **Each `dp[i][j]` tells you how many squares end at that position**.
    - A `dp[i][j] = 3` means:
        - One 3×3 square ends here,
        - One 2×2 square (inside the 3×3) also ends here,
        - One 1×1 square also ends here.

So we add all `dp[i][j]` values across the matrix to get the **total number of square submatrices with all 1s**.

Summary of the Common DP Approach

- Build a DP table where `dp[i][j] = size of largest square ending at (i, j)`
- Use the recurrence:
```
dp[i][j] = 1 + min(top, left, top-left)  if matrix[i][j] == 1
dp[i][j] = 0                             if matrix[i][j] == 0
```
- Sum all `dp[i][j]` values to get the final answer

Step-by-Step Walkthrough

Input matrix (garden):

```
[
  [1, 0, 1],
  [1, 1, 0],
  [1, 1, 0]
]
```

We will initialize a DP table of the same size:

```
dp = [
  [0, 0, 0],
  [0, 0, 0],
  [0, 0, 0]
]
```

We also initialize `total = 0` to store the number of squares.

Fill the DP table

Row 0:

- **(0,0)**: garden[0][0] = 1 → first row, so `dp[0][0] = 1` → total = 1
- **(0,1)**: garden[0][1] = 0 → `dp[0][1] = 0` → total = 1
- **(0,2)**: garden[0][2] = 1 → first row, so `dp[0][2] = 1` → total = 2

dp after row 0:

```
[
  [1, 0, 1],
  [0, 0, 0],
  [0, 0, 0]
]
```

Row 1:

- **(1,0)**: garden[1][0] = 1 → first column, so `dp[1][0] = 1` → total = 3
- **(1,1)**: garden[1][1] = 1
  → check min of top (dp[0][1]=0), left (dp[1][0]=1), and top-left (dp[0][0]=1)
  → `dp[1][1] = 1 + min(0,1,1) = 1` → total = 4
- **(1,2)**: garden[1][2] = 0 → `dp[1][2] = 0` → total = 4

dp after row 1:

```
[
  [1, 0, 1],
  [1, 1, 0],
  [0, 0, 0]
]
```

Row 2:

- **(2,0)**: garden[2][0] = 1 → first column, so `dp[2][0] = 1` → total = 5

- **(2,1)**: garden[2][1] = 1
  → check min of top (dp[1][1]=1), left (dp[2][0]=1), top-left (dp[1][0]=1)
  → `dp[2][1] = 1 + min(1,1,1) = 2` → total = 7

- **(2,2)**: garden[2][2] = 0 → `dp[2][2] = 0` → total = 7

Final dp table:

```
[
  [1, 0, 1],
  [1, 1, 0],
  [1, 2, 0]
]
```

Final Calculation

To get the **total number of square submatrices with all 1s**, sum all values in the `dp` table:

```
1 + 0 + 1 + 1 + 1 + 0 + 1 + 2 + 0 = 7
```

Problem 5

You are working on a video compression algorithm. A video is broken into `n` frames, and each frame has a **complexity score** represented by an integer in an array `frames`. To compress the video, you can **group contiguous frames** into segments, with each segment being **at most `k` frames long**.

For every segment you create, the compression algorithm simplifies all frames in that segment by assigning them the **maximum complexity** within that segment (since the hardest frame to compress defines the difficulty of the segment). The **total cost** of compression is the **sum of all new frame values after partitioning**.

Your task is to find the **maximum total cost** you can get by optimally partitioning the video into segments of length at most `k`.

Example

Input:

```
frames = [1, 15, 7, 9, 2, 5, 10]
k = 3
```

Output: `84`

Explanation:

- One optimal way to partition is: `[15,15,15] [9] [10,10,10]`
- Sum = 15+15+15 + 9 + 10+10+10 = **84**

Task:

1. Explain how you would solve this problem using dynamic programming in O(nk) time
2. Use the input `frames = [1, 4, 1, 5, 7, 3, 6, 1, 9, 9, 3], k = 4` to walk through how your approach calculates the result step by step.

Answer

To solve this problem efficiently, we use **dynamic programming**. But before jumping into code or formulas, let's understand how to think about the problem and how this approach naturally fits.

How to think about the problem

We are given an array of frame complexities, and we can group **contiguous frames** into segments of at most `k` frames. Each frame in a segment is replaced by the **maximum value** of that segment, and our goal is to **maximize the total sum** of the resulting array.

When you're at a certain index `i` in the array (say frame `i`), you don't know yet how long your current segment should be — it could be:

- just `frames[i]`
- or `[frames[i], frames[i+1]]`
- or up to `k` frames (as long as it doesn't go past the end).

For each of these choices, you:

1. **Calculate the group score**: take the max value in the group × the number of frames in it.
2. **Add the best total score** from whatever comes **after** this group — that's what we've already stored in `dp[i + length]`.

The best choice at `i` is the one that gives you the highest **combined score**: current group score **plus** best future score.

This is why we use **dynamic programming**:

- We store answers for future positions (like `dp[i+1]`, `dp[i+2]`...) so we don't have to recalculate them.
- We build the best answer for each index by **reusing** the best answers we've already computed for later positions.

Defining the DP Approach

- Let `dp[i]` represent the **maximum total compression cost** starting from index `i`.
- We build this solution **from the end to the beginning** of the array.

At each position `i`:

- Try all possible group lengths from `1` to `k`.
- For each possible group:
  - Keep track of the **maximum frame value** within the group.
  - Compute the group cost = `max_value × group_size`.
  - Combine it with the **previously computed** `dp[i + group_size]`.
  - Take the **maximum** total score among all options.

Why solve from the end?

To compute `dp[i]`, we need `dp[i+1]`, `dp[i+2]`, ..., depending on the group size.
That means we must solve the **later parts of the array first**, then work our way backward.

Step by step with input `frames = [1, 4, 1, 5, 7, 3, 6, 1, 9, 9, 3], k = 4`

Let's trace the dynamic programming computation step by step.

We'll go **from the end to the beginning**, updating `dp[start]` at each step. Here's a high-level view of what happens:

- At each `start` index, we try all segment lengths `1 to k` (up to `k = 4`), calculate the best segment ending at that point, and store the best result.

Example highlights:

- **start = 10** (value = 3):
  Only one possible segment `[3]`, cost = 1 × 3 = 3 → `dp[10] = 3`
- **start = 9** (value = 9):
  Try:
  - `[9]` → cost = 9 + dp[10] = 9 + 3 = 12
  - `[9, 3]` → max = 9, len = 2 → 18
    → `dp[9] = 18`
- **start = 8** (value = 9):
  Try:
  - `[9]` → 9 + dp[9] = 9 + 18 = 27
  - `[9, 9]` → 9×2 + dp[10] = 18 + 3 = 21
  - `[9, 9, 3]` → 9×3 + dp[11] = 27 + 0 = 27
    → `dp[8] = 27`
- **start = 7** (value = 1):
  Try:
  - `[1]` → 1 + dp[8] = 1 + 27 = 28
  - `[1, 9]` → max = 9, len = 2 → 18 + dp[9] = 18 + 18 = 36
  - `[1, 9, 9]` → 9×3 + dp[10] = 27 + 3 = 30
  - `[1, 9, 9, 3]` → 9×4 = 36 + dp[11] = 36
    → `dp[7] = 36`

- And so on, all the way to `start = 0`.

Finally, `dp[0] = 83`, which is the **maximum total compression score** achievable.