DYNAMIC PROGRAMMING PROBLEMS

ANANDIKA M (23PD03)

FARHANA S (23PD10)

NANDHANA S (23PD22)

Question 1.(a) :

Problem Statement

There are N stones, numbered $1, 2, \ldots, N$. For each i ($1 \leq i \leq N$), the height of Stone i is h_i.

There is a frog who is initially on Stone 1. He will repeat the following action some number of times to reach Stone N:

- If the frog is currently on Stone i, jump to Stone $i+1$ or Stone $i+2$. Here, a cost of $|h_i - h_j|$ is incurred, where j is the stone to land on.

Find the minimum possible total cost incurred before the frog reaches Stone N.

Constraints

- All values in input are integers.
- $2 \leq N \leq 10^5$
- $1 \leq h_i \leq 10^4$

Understanding Question 1 (a)

We have N stones, each with a given height h[i].

The frog starts at **stone 1** and must reach **stone N**.

The **cost** of a jump from stone i to stone j is:

abs(h[i]−h[j])

At each step, the frog can jump:

- From stone i → i+1, cost : abs(h[i]-h[i+1])
- From stone i → i+2, cost : abs(h[i]-h[i+2])

Goal: **Find the minimum possible total cost** for the frog to reach stone N.

WHY IS THIS A DYNAMIC PROGRAMMING PROBLEM ?

At each stone, the frog has **two choices**: jump to the next stone or skip one stone.

To decide the minimum cost at stone i, we need the **minimum cost of reaching previous stones**.

Overlapping of subproblems are evident here -> Dynamic programming is used

Example explained

N = 4 and h = [10, 30, 40, 20]

Reaching stone 1 :

dp[1] = 0

Reaching Stone 2 :

Stone 1 → Stone 2

dp[2] = dp[1] + |30 - 10| =0 + 20 = 20

Reaching stone 3

Option 1: Stone 2 → Stone 3

dp[2] + |40 - 30| = 20 + 10 = 30

Option 2: Stone 1 → Stone 3

dp[1] + |40 - 10| = 0 + 30 = 30

dp[3] = min(30, 30) = 30

Reaching stone 4 :

Option 1: Stone 3 → Stone 4

dp[3] + |20 - 40| = 30 + 20 = 50

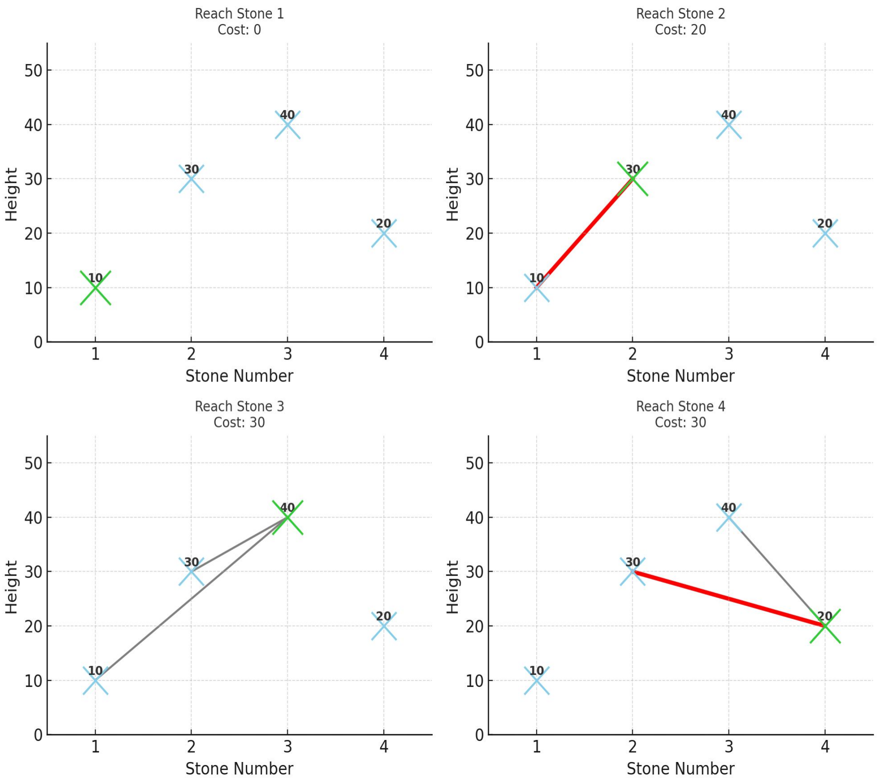
Option 2: Stone 2 → Stone 4

dp[2] + |20 - 30| = 20 + 10 = 30

dp[4] = min(50, 30) = 30

Minimum cost to reach Stone 4 = 30 and the best path was: **Stone 1** → **Stone 2** → **Stone 4**.

Representation



Algorithm

```
def min_frog_jump_cost(h):
    N = len(h)
    Initialise dp[N]={0}
    dp[0] = 0
    dp[1] = abs(h[1] - h[0])

    for i in range(2, N):
        jump1 = dp[i-1] + abs(h[i] - h[i-1])
        jump2 = dp[i-2] + abs(h[i] - h[i-2])
        dp[i] = min(jump1, jump2)

    return dp[N-1]
```

Time Complexity = O(N)
Explanation : For every i, we only look at two previous states. Each step is O(1) work. We loop from stone 2 up to stone N.
That’s (N-1) iterations ≈ O(N).

Question 1.(b) :

Problem Statement

There are N stones, numbered $1, 2, \ldots, N$. For each i ($1 \leq i \leq N$), the height of Stone i is h_i.

There is a frog who is initially on Stone 1. He will repeat the following action some number of times to reach Stone N:

- If the frog is currently on Stone i, jump to one of the following: Stone $i + 1, i + 2, \ldots, i + K$. Here, a cost of $|h_i - h_j|$ is incurred, where j is the stone to land on.

Find the minimum possible total cost incurred before the frog reaches Stone N.

Constraints

- All values in input are integers.
- $2 \leq N \leq 10^5$
- $1 \leq K \leq 100$
- $1 \leq h_i \leq 10^4$

Understanding Question 1 (b)

We have N stones, each with a given height h[i…N]. The frog starts at **stone 1** and must reach **stone N**.

The **cost** of a jump from stone i to stone j is :   abs(h[i]−h[j])

At each step, the frog can jump:

- From stone i → i+k   ->   cost : abs(h[i]-h[i+k])
- From stone i → i+(k+1) ->   cost : abs(h[i]-h[i+(k+1)]

Goal: **Find the minimum possible total cost** for the frog to reach stone N.

GENERAL :

$$dp[i] = \min \begin{cases} dp[i-k] + c_{i-k,\,i} & \text{iff } dp[i-k] \neq -1 \\ dp[i-(k+1)] + c_{i-(k+1),\,i} & \text{iff } dp[i-(k+1)] \neq -1 \end{cases}$$

Example explained

N = 6 and h = [30, 10, 60, 10, 60, 50] with k = 2

Base : $dp[0] = 0$

Reaching stone 1 : (1 < k)

$dp[1] = -1$

Reaching Stone 2 : Stone 0 → Stone 2

$$dp[2] = \min \left\{ \begin{array}{l} dp[0] + |h[2] - h[0]| = 0 + |60 - 30| = 30 \\ (\text{dp}[-1] \text{ invalid for 3-step}) \end{array} \right\} = 30$$

Reaching stone 3

Option 1: Stone 1 → Stone 3 → invalid (dp[1] = -1)

Option 2: Stone 0 → Stone 3 (k + 1 steps),

$$dp[3] = \min \left\{ \begin{array}{l} dp[1] + |h[3] - h[1]| = -1 \text{ (invalid)} \\ dp[0] + |h[3] - h[0]| = 0 + |10 - 30| = 20 \end{array} \right\} = 20$$

Reaching stone 4 :

Option 1: Stone 2 → Stone 4

Option 2: Stone 1 → Stone 4 (Not possible)

$$dp[4] = \min \left\{ \begin{array}{l} dp[2] + |h[4] - h[2]| = 30 + |60 - 60| = 30 \\ dp[1] + |h[4] - h[1]| = -1 \text{ (invalid)} \end{array} \right\} = 30$$

Reaching Stone 5 :

Option 1: Stone 3 → Stone 5

Option 2: Stone 2 → Stone 5

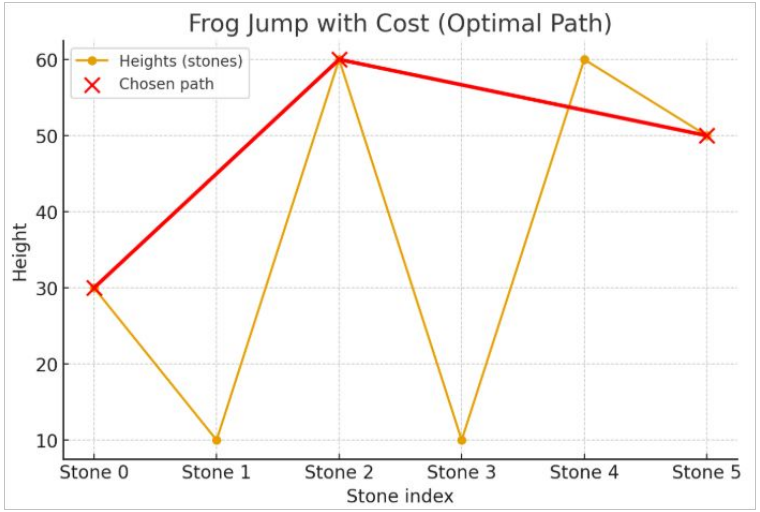
$$dp[5] = \min \left\{ \begin{array}{l} dp[3] + |h[5] - h[3]| = 20 + |50 - 10| = 60 \\ dp[2] + |h[5] - h[2]| = 30 + |50 - 60| = 40 \end{array} \right\} = 40$$

$$dp = [0, -1, 30, 20, 30, 40]$$

$$\text{Minimum cost to reach Stone 5 (last stone)} = 40$$

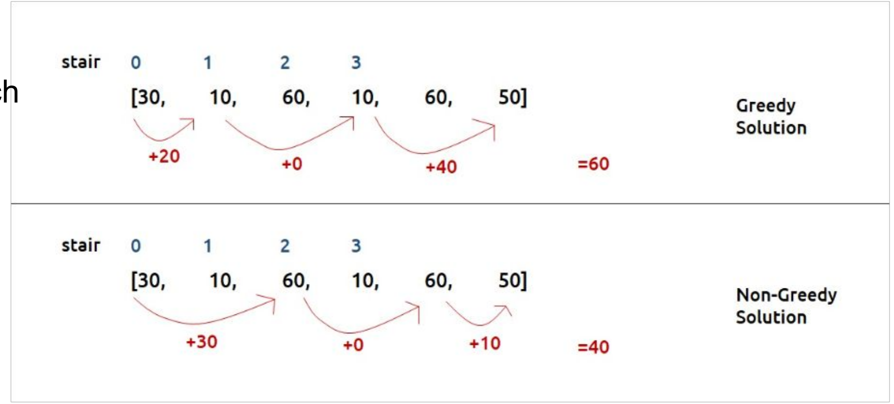
Algorithm

```
def min_frog_jump_cost(heights, K):
    N = len(heights)
    dp = [float('inf')] * N
    dp[0] = 0   # base case: starting stone

    for i in range(1, N):
        for step in range(1, K+1):
            if i - step >= 0:
                dp[i] = min(dp[i], dp[i - step] + abs(heights[i] - heights[i - step]))

    return dp[N-1]
```



Time Complexity = O(N)

Explanation : For every i, we only look at two previous states. Each step is O(1) work. We loop from stone k up to stone N. That's (N-k) iterations ≈ O(N).



Note :
For larger K, we get optimum value. However the exact K value can be found using naive approach $O(n^2)$

Contractor problem :

Question :

You are a freelance contractor, and your available jobs change weekly over `n` weeks. The jobs are split into two groups:

- `ls` (low-stress jobs): These jobs require no special preparation. The revenue for a low-stress job in week `i` is given by `l[i]`.
- `hs` (high-stress jobs): If you choose a job from `hs` in a given week, you must prepare the week beforehand by taking no job at all. The revenue for a high-stress job in week `i` is given by `h[i]`.

Eg : `n = 5` weeks

- `l = [30, 5, 20, 25, 500]` (low-stress)
- `h = [0, 50, 70, 100, 110]` (high-stress)

```
function max_income(l, h, n):
    memo = empty dictionary  // key: (i, prev_taken), value: max revenue from week i to end
    function solve(i, prev_taken):
        if i >= n:
            return 0
        if (i, prev_taken) in memo:
            return memo[(i, prev_taken)]
        best = solve(i+1, False)  // Option 1: Rest this week
        best = max(best, l[i] + solve(i+1, True))  // Option 2: Take low-stress job
        if not prev_taken:
            best = max(best, h[i] + solve(i+1, True))  // Option 3: Take high-stress job
        memo[(i, prev_taken)] = best
        return best
    return solve(0, False)  // Start at week 0, previous week assumed rest
```

```
solve(0, False)
├─ Rest → solve(1, False)
│   ├─ Rest → solve(2, False)
│   │   ├─ Rest → solve(3, False)
│   │   │   ├─ Rest → solve(4, False)
│   │   │   │   ├─ Rest → solve(5, False) = 0
│   │   │   │   ├─ Low (500 + solve(5, True)=500)
│   │   │   │   └─ High (110 + solve(5, True)=110)
│   │   │   │      → best = 500
│   │   │   ├─ Low (25 + solve(4, True))
│   │   │   │   ├─ Rest → solve(5, False)=0
│   │   │   │   └─ Low (500+solve(5,True)=500) → total=525
│   │   │   │      → best=525
│   │   │   └─ High (100 + solve(4, True))
│   │   │       ├─ Rest → solve(5,False)=0
│   │   │       └─ Low (500+solve(5,True)=500) → total=600
```

```
│    │    │        → best=600
│    │    │        → solve(3,False)=max(500,525,600)=600
│    │    ├─ Low (20 + solve(3, True))
│    │    │    ├─ Rest → solve(4, False)=500
│    │    │    ├─ Low (25+solve(4,True)=25+500=525)
│    │    │    └─ (no High because prevTaken=True)
│    │    │   → best=525 → total=20+525=545
│    │    └─ High (70 + solve(3, True))
│    │        ├─ Rest → solve(4, False)=500
│    │        ├─ Low (25+solve(4,True)=525)
│    │       → best=525 → total=70+525=595
│    │       → solve(2,False)=max(600,545,595)=600
│    ├─ Low (5 + solve(2, True))
│    │    ├─ Rest → solve(3, False)=600
│    │    ├─ Low (20+solve(3,True)=20+525=545)
│    │   → best=600 → total=605
```

```
└─ High (50 + solve(2, True))
│        ├─ Rest → solve(3, False)=600
│        ├─ Low (20+solve(3,True)=545)
│        → best=600 → total=650
│        → solve(1,False)=max(600,605,650)=650
├─ Low (30 + solve(1, True))
│    ├─ Rest → solve(2, False)=600
│    ├─ Low (5+solve(2,True)=5+545=550)
│    └─ High not allowed
│    → best=600 → total=30+600=630
└─ High (0 + solve(1, True))   # h[0]=0
     ├─ Rest → solve(2, False)=600
     ├─ Low (5+solve(2,True)=550)
    → best=600 → total=600
```

FINAL ANSWER = max(650, 630, 600) = 650

Time Complexity :

O(n) where n is the no of weeks

Space Complexity :

O(n) for the memoization dictionary (stores 2n states).

O(n) for the recursion call stack (depth of recursion is n).

So, overall space complexity is O(n).