PW SKILLS

Lesson Plan

DP-4

PW SKILLS

Today's checklist

- Roy and Coin Boxes
- matrix Chain Multiplication
- Mixtures
- Unique Paths II
- Unique binary Search Trees
- Deque
- Mancunian and K-Ordered LCS
- Best time to buy and sell stock IV
- Adjacent Bit counts
- Minimum number of Taps to open to water a Garden

Roy and Coin Boxes

Ques.
Roy has N coin boxes numbered from 1 to N.
Every day he selects two indices [L,R] and adds 1 coin to each coin box starting from L to R (both inclusive).
He does this for M number of days.
After M days, Roy has a query: How many coin boxes have at least X coins.
He has Q such queries.

Input:
First line contains N - number of coin boxes.
Second line contains M - number of days.
Each of the next M lines consists of two space separated integers L and R.
Followed by integer Q - number of queries.
Each of the next Q lines contain a single integer X.

Output:
For each query output the result in a new line.

Constraints:
1 ≤ N ≤ 1000000
1 ≤ M ≤ 1000000
1 ≤ L ≤ R ≤ N
1 ≤ Q ≤ 1000000
1 ≤ X ≤ N

Sample Input

7
4
1 3
2 5
1 2
5 6
4
1
7
4
2

Sample Output

6
0
0
4

Solution:

Observation 1: If we can keep track of how many days a box was between the starting box and the end box, then the total number of coins at the end is equal to this number. This is so because each day exactly one coin is added to a box which comes in between the starting and ending box.
Given this observation, now we have to come up with a way to count the total number of days a box was filled.

Observation 2: At a given day when Roy adds coins to boxes, he adds them to all the boxes within two points. We are given the information that for each day from which box adding coins to the boxes was started and till which box the coins were added. Using this, for a particular box, we can count on how many days we start at this box and for how many days we end at the box.
This can be done quite simply by keeping two arrays (start and end), each of size N. start counts for how many days we start adding coins from this box and end counts for how many days we stop adding coins at this box. For a day where we start from L to R, we increment start[L] by 1 and increment end[R] by 1. For a single day, this can be done in O(1) time.

Finding the total number of coins at the end for each box:
Consider box 1, if we know that this box was opening box for ks1 days, then the total number of coins in this box can only be ks1 (because each day a coin was added to this box).
Consider box 2, we know that box 1 was starting box for ks1 days, we would add a coin to box 2, for each day where we start from box 2 itself (say ks2) and for those days where we start from box 1 but don't stop at box 1. Those days will be exactly ks1 - ke1, where ke1 is the number of days box 1 was the final box of adding coins. Hence, the total number of coins in box 2 will be ks1 + ks2 - ke1 .
Now, for box 3, we can see that the total number of coins will be ks1 + ks2 + ks3 - ke1 - ke2. This follows for other boxes as well.

To compute the value for each of box, we can keep a running count of sum of all start values before that box minus the sum of all ending values before that box. Adding the starting value for that box itself to this sum gives the total number of coins in that box.
This solution will give us the total number of coins in a box in O(N) time.
This approach can also be implemented using single array like:

```
import java.util.Scanner;

public class PrefixSum {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        long n = scanner.nextLong();
        long[] cnt = new long[(int) (n + 1)];
        long[] arr = new long[(int) (n + 1)];
        long[] tree = new long[(int) (n + 1)];

        long q = scanner.nextLong();
        while (q-- > 0) {
            long a = scanner.nextLong();
            long b = scanner.nextLong();
            cnt[(int) a]++;
            cnt[(int) (b + 1)]--;
        }

        long currentSum = 0;
        for (long i = 1; i <= n; i++) {
            currentSum += cnt[(int) i];
            arr[(int) i] = currentSum;
            tree[(int) currentSum]++;
        }
    }
}
```

q = Number of Queries cnt = Merged array start and end currentsum = Sum upto that index tree[idx] = number of boxes with coins = idx

Observation 3: Note that the total number of coins in a box can never be more than M. We keep an array of size M and using the above computation of finding total number of coins in a box, we populate how many boxes there are with i coins at the end for each i.

Observation 4: If a box is counted when we were counting for boxes with X or more coins, it will also be counted when we would count boxes with X-i or more coins, for positive values of i.
This observation helps us computing how many boxes have atleast X coins. We already know the count of boxes with exactly i number of coins for each i. This small last step is left for the reader to solve.

```
   for(ll i=1;i<=1000000;i++)
{
    tree[i]+=tree[i-1];
}
Number of boxes with atleast x coins = n-tree[x-1]
```

Code:

```
import java.util.Scanner;

public class PrefixSum {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        long n = scanner.nextLong();
        long[] arr = new long[1000003];
        long[] tree = new long[1000001];
        long[] cnt = new long[(int) (n + 1)];

        for (int i = 0; i ⩽ n; i++)
            cnt[i] = 0;

        long q = scanner.nextLong();
        while (q-- > 0) {
            long a = scanner.nextLong();
            long b = scanner.nextLong();
            cnt[(int) a]++;
            cnt[(int) (b + 1)]--;
        }

        long currentSum = 0;
        for (long i = 1; i ⩽ n; i++) {
            currentSum += cnt[(int) i];
            arr[(int) i] = currentSum;
            tree[(int) currentSum]++;
        }

        for (long i = 1; i ⩽ 1000000; i++) {
            tree[(int) i] += tree[(int) (i - 1)];
        }

        long t = scanner.nextLong();
        while (t-- > 0) {
            long x = scanner.nextLong();
            if (x == 0)
                System.out.println(n);
            else
                System.out.println(n - tree[(int) (x - 1)]);
        }
    }
}
```

Matrix Chain Multiplication

Ques
Given a sequence of matrices, find the most efficient way to multiply these matrices together. The efficient way is the one that involves the least number of multiplications.
The dimensions of the matrices are given in an array arr[] of size N (such that N = number of matrices + 1) where the ith matrix has the dimensions (arr[i-1] x arr[i]).

Example 1:
Input: N = 5
arr = {40, 20, 30, 10, 30}
Output: 26000

Explanation: There are 4 matrices of dimension 40x20, 20x30, 30x10, 10x30. Say the matrices are named as A, B, C, D. Out of all possible combinations, the most efficient way is (A*(B*C))*D.
The number of operations are -
20*30*10 + 40*20*10 + 40*10*30 = 26000.

Example 2:
Input: N = 4
arr = {10, 30, 5, 60}
Output: 4500

Explanation: The matrices have dimensions 10*30, 30*5, 5*60. Say the matrices are A, B and C. Out of all possible combinations,the most efficient way is (A*B)*C. The number of multiplications are -
10*30*5 + 10*5*60 = 4500.

Your Task:
You do not need to take input or print anything. Your task is to complete the function matrixMultiplication() which takes the value N and the array arr[] as input parameters and returns the minimum number of multiplication operations needed to be performed.

Code:

```
public class Solution {
    // Function to find the minimum cost of matrix multiplication
    public int matrixMultiplication(int N, int[] arr) {
        // Creating a 2D array to store the result
        int[][] dp = new int[N + 1][N + 1];
        for (int i = 0; i < N; i++)
            dp[i][i] = 0;

        // Nested loops to calculate the minimum cost
        // of multiplying matrices
        for (int L = 2; L < N; L++) {
            for (int i = 1; i < N - L + 1; i++) {
                int j = i + L - 1;
                dp[i][j] = Integer.MAX_VALUE;
                for (int k = i; k < j; k++) {
                    dp[i][j] = Math.min(dp[i][j],
                            dp[i][k] + dp[k + 1][j] + arr[i - 1] *
arr[k] * arr[j]);
                }
            }
        }
        return dp[1][N - 1];
    }
}
```

Mixtures

Harry Potter has n mixtures in front of him, arranged in a row. Each mixture has one of 100 different colors (colors have numbers from 0 to 99).

He wants to mix all these mixtures together. At each step, he is going to take two mixtures that stand next to each other and mix them together, and put the resulting mixture in their place.

When mixing two mixtures of colors a and b, the resulting mixture will have the color (a+b) mod 100.

Also, there will be some smoke in the process. The amount of smoke generated when mixing two mixtures of colors a and b is a*b.

Find out what is the minimum amount of smoke that Harry can get when mixing all the mixtures together.

Problem Solution
The problem is very similar to matrix multiplication problem. In the marix multiplication problem, when we multiply two matrices of the order (a,b) and (b,c), we get a matrix of the order (a,c). Here, when we mix two adjacent colors a and b, we will get (a+b)%100. So, we will proceed in similar fashion using bottom up dynamic programming and store the result of sub-problems in a matrix.

Expected Input and Output
Case-1:

number of mixtures - 2
Initial color of mixtures - 18, 19
Minimum amount of smoke produced in mixing all mixtures together - 342

Code:

```
import java.util.Scanner;

public class MinSmoke {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int i, j, k;
        int x;
        int n;

        System.out.print("Enter the number of mixtures: ");
        n = scanner.nextInt();
        int[] a = new int[n];

        // 2D array to store the minimum smoke and the resulting
color
        int[][][] dp = new int[n + 1][n + 1][2];

        System.out.println("Enter the initial color of mixtures:");
        for (i = 1; i <= n; i++) {
            dp[i][i][1] = scanner.nextInt();
            dp[i][i][0] = 0;
        }

        for (int gap = 2; gap <= n; gap++) {
            for (i = 1; i <= (n - gap + 1); i++) {
                j = i + gap - 1;
                dp[i][j][0] = Integer.MAX_VALUE;

                for (k = i; k < j; k++) {
                    // try for every value of k
                    x = dp[i][k][0] + dp[k + 1][j][0] + dp[i][k][1]
* dp[k + 1][j][1];
                    if (x < dp[i][j][0]) {
                        dp[i][j][0] = x;
                        dp[i][j][1] = (dp[i][k][1] + dp[k + 1][j]
[1]) % 100;
                    }
                }
            }
        }

        System.out.println("The minimum amount of smoke that will
be produced by mixing all the mixtures together is:");
        System.out.println(dp[1][n][0]);
    }
}
```

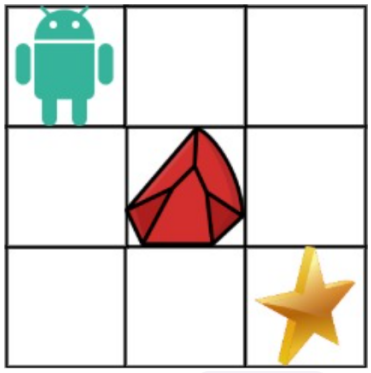
Paths II

You are given an m x n integer array grid. There is a robot initially located at the top-left corner (i.e., grid[0][0]). The robot tries to move to the bottom-right corner (i.e., grid[m - 1][n - 1]). The robot can only move either down or right at any point in time.

An obstacle and space are marked as 1 or 0 respectively in grid. A path that the robot takes cannot include any square that is an obstacle.

Return the number of possible unique paths that the robot can take to reach the bottom-right corner.
The testcases are generated so that the answer will be less than or equal to 2 * 109.

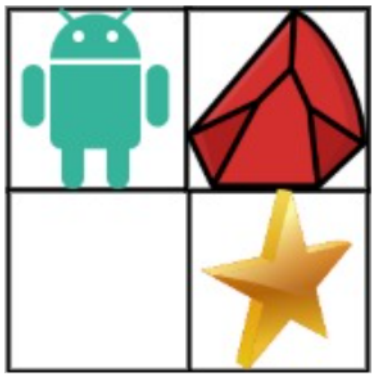
Example 1:



Input: obstacleGrid = [[0,0,0],[0,1,0],[0,0,0]]
Output: 2

Explanation: There is one obstacle in the middle of the 3x3 grid above.
There are two ways to reach the bottom-right corner:
1. Right -> Right -> Down -> Down
2. Down -> Down -> Right -> Right

Example 2:



Input: obstacleGrid = [[0,1],[0,0]]
Output: 1

Constraints:

- m == obstacleGrid.length
- n == obstacleGrid[i].length
- 1 <= m, n <= 100
- obstacleGrid[i][j] is 0 or 1.

Code:

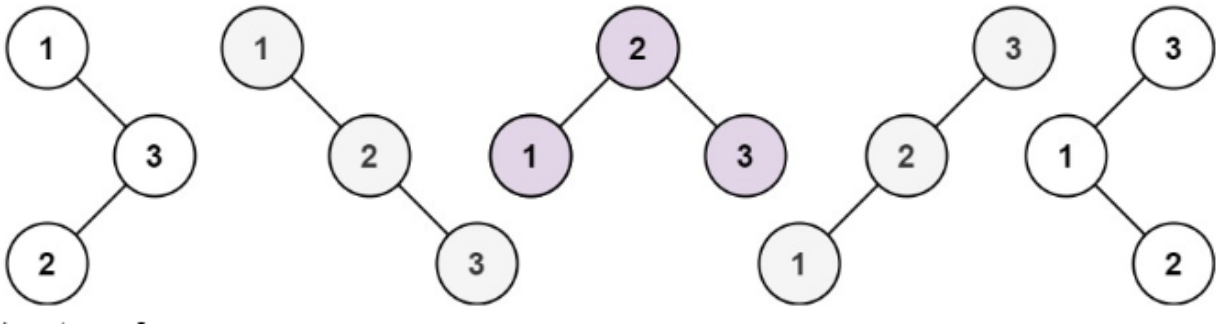
```
public int uniquePathsWithObstacles(int[][] obstacleGrid) {
 int m = obstacleGrid.length, n = obstacleGrid[0].length;
 int[][] path = new int[m][n];

 for (int i = 0; i < m; i++) {
  if (obstacleGrid[i][0] == 1)  {
   path[i][0] = 0;
   //on the first column, if there is an obstacle, the rest are
blocked.
   //no need to continue.
   break;
  } else
   path[i][0] = 1;
 }
 for (int j = 0; j < n; j++) {
  if (obstacleGrid[0][j] == 1)  {
   path[0][j] = 0;
   //First row, once obstacle found, the rest are blocked.
   break;
  } else
   path[0][j] = 1;
 }
 for (int i = 1; i < m; i++) {
  for (int j = 1; j < n; j++) {
   if (obstacleGrid[i][j] == 1)
    path[i][j] = 0;
   else
    path[i][j] = path[i-1][j] + path[i][j-1];
  }
 }
 return path[m-1][n-1];
}
```

Unique binary Search Trees

Given an integer n, return the number of structurally unique BST's (binary search trees) which has exactly n nodes of unique values from 1 to n.

Example 1:



Input: n = 3
Output: 5

Example 2:

Input: n = 1
Output: 1

Constraints:

- 1 <= n <= 19

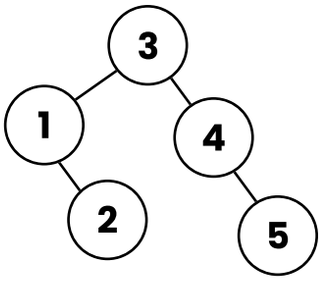
Intuition

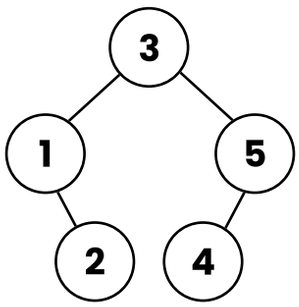
The Idea behind this problem :
We should iterate every possible node between 1 and n as a root node. Calculate the number of possible combinations for each root node by calculating the number of possible combinations of left subtree and right subtree.

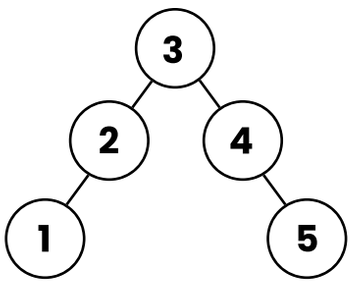
Approach

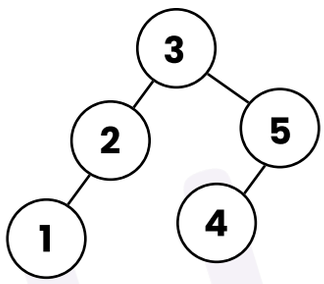
Let us suppose we take n=5
we should iterate every root from 1 to 5 as a root node.
If the root node is 3 we can have 1 and 2 in left subtree, 4 and 5 in right subtree.
The number of possible combinations of left subtree = 2
The number of possible combinations of right subtree = 2

1 2 3 4 5









The number of possible ways to build a binary tree with root node 3 = (The number of possible ways to build left binary subtree)*(The number of possible ways to build right binary subtree) = 4.
Therefore the total number of possible binary search trees with value of n = The sum of the total number of possible binary search trees with every node as root node.

Complexity

- Time complexity:
- O(n^2)
- Space complexity:
- O(n)

Code:

```
public class Solution {
    public int numTrees(int n) {
        int[] dp = new int[n + 1];
        dp[0] = 1;
        for (int i = 1; i ⩽ n; i++) {
            int sum = 0;
            for (int j = 0; j < i; j++) {
                sum += dp[j] * dp[i - j - 1];
            }
            dp[i] = sum;
        }
        return dp[n];
    }
}
```

Mancunian and K-Ordered LCS

Any programmer worth his salt would be familiar with the famous Longest Common Subsequence problem. Mancunian was asked to solve the same by an incompetent programmer. As expected, he solved it in a flash. To complicate matters, a twist was introduced in the problem.
In addition to the two sequences, an additional parameter k was introduced. A k-ordered LCS is defined to be the LCS of two sequences if you are allowed to change at most k elements in the first sequence to any value you wish to. Can you help Mancunian solve this version of the classical problem?

Input:
The first line contains three integers N, M and k, denoting the lengths of the first and second sequences and the value of the provided parameter respectively.
The second line contains N integers denoting the elements of the first sequence.
The third line contains M integers denoting the elements of the second sequence.

Output:
Print the answer in a new line.

Constraints:
1 <= N, M <= 2000
1 <= k <= 5
1 <= element in any sequence <= 109

Input
5 5 1
1 2 3 4 5
5 3 1 4 2

Output
3

As mentioned in the problem statement, given problem is quite similar to standard LCS problem. We can have following dp state
DP(n,m,k) => denotes LCS for first n number of first array, first m numbers of second array when we are allowed to change at max k numbers in first array.
Recursion look like this ->
DP(n,m,k) = max(DP(n-1, m,k), DP(n, m-1,k), DP(n-1, m-1,k -1)+1 if arr[n]!=arr[m])
DP(n,m,k) = max(DP(n-1, m,k), DP(n, m-1,k), DP(n-1, m-1,k)+1 if arr[n]==arr[m])

Code:

```
import java.util.Arrays;
import java.util.Scanner;

public class Main {
    static final int N = 2002;
    static int n, m, k;
    static int[] arr1 = new int[N];
    static int[] arr2 = new int[N];
    static int[][][] DP = new int[N][N][6];

    public static int dp(int n, int m, int k) {
        if (k < 0) return -10000000;
        if (n == 0 || m == 0) return 0;

        int ans = DP[n][m][k];
        if (ans != -1) return ans;

        ans = Math.max(dp(n - 1, m, k), dp(n, m - 1, k));
        if (arr1[n] == arr2[m]) ans = Math.max(ans, 1 + dp(n - 1, m
- 1, k));
        ans = Math.max(ans, 1 + dp(n - 1, m - 1, k - 1));

        return ans;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        n = scanner.nextInt();
        m = scanner.nextInt();
        k = scanner.nextInt();
        assert (n >= 1 && n <= 2000);
        assert (m >= 1 && m <= 2000);
        assert (k >= 1 && k <= 5);

        for (int i = 1; i <= n; i++) {
            arr1[i] = scanner.nextInt();
            assert (arr1[i] >= 1 && arr1[i] <= 1e9);
        }

        for (int i = 1; i <= m; i++) {
            arr2[i] = scanner.nextInt();
            assert (arr2[i] >= 1 && arr2[i] <= 1e9);
        }

        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                Arrays.fill(DP[i][j], -1);
            }
        }

        System.out.println(dp(n, m, k));
    }
}
```

Best time to buy and sell stock IV

You are given an integer array prices where prices[i] is the price of a given stock on the ith day, and an integer k.
Find the maximum profit you can achieve. You may complete at most k transactions: i.e. you may buy at most k times and sell at most k times.
Note: You may not engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again).

Example 1:

Input: k = 2, prices = [2,4,1]
Output: 2

Explanation: Buy on day 1 (price = 2) and sell on day 2 (price = 4), profit = 4-2 = 2.

Example 2:
Input: k = 2, prices = [3,2,6,5,0,3]
Output: 7

Explanation: Buy on day 2 (price = 2) and sell on day 3 (price = 6), profit = 6-2 = 4. Then buy on day 5 (price = 0) and sell on day 6 (price = 3), profit = 3-0 = 3.

Constraints:

- 1 <= k <= 100
- 1 <= prices.length <= 1000
- 0 <= prices[i] <= 1000

Code:

```
/**
 * dp[i, j] represents the max profit up until prices[j] using at
most i transactions.
 * dp[i, j] = max(dp[i, j-1], prices[j] - prices[jj] + dp[i-1, jj])
{ jj in range of [0, j-1] }
 *          = max(dp[i, j-1], prices[j] + max(dp[i-1, jj] -
prices[jj]))
 * dp[0, j] = 0; 0 transactions makes 0 profit
 * dp[i, 0] = 0; if there is only one price data point you can't
make any transaction.
 */

public int maxProfit(int k, int[] prices) {
 int n = prices.length;
 if (n ≤ 1)
  return 0;
 //if k ≥ n/2, then you can make maximum number of transactions.
 if (k ≥  n/2) {
  int maxPro = 0;
  for (int i = 1; i < n; i++) {
   if (prices[i] > prices[i-1])
    maxPro += prices[i] - prices[i-1];
  }
  return maxPro;
 }
    int[][] dp = new int[k+1][n];
    for (int i = 1; i ≤ k; i++) {
     int localMax = dp[i-1][0] - prices[0];
     for (int j = 1; j < n; j++) {
      dp[i][j] = Math.max(dp[i][j-1],  prices[j] + localMax);
      localMax = Math.max(localMax, dp[i-1][j] - prices[j]);
     }
    }
    return dp[k][n-1];
}
```

Adjacent Bit counts

Given an integer n, return an array ans of length n + 1 such that for each i (0 <= i <= n), ans[i] is the number of 1's in the binary representation of i.

Example 1:

Input: n = 2
Output: [0,1,1]

Explanation:
0 --> 0
1 --> 1
2 --> 10

Example 2:

Input: n = 5
Output: [0,1,1,2,1,2]

Explanation:
0 --> 0
1 --> 1
2 --> 10
3 --> 11
4 --> 100
5 --> 101

Constraints:
- 0 <= n <= 105

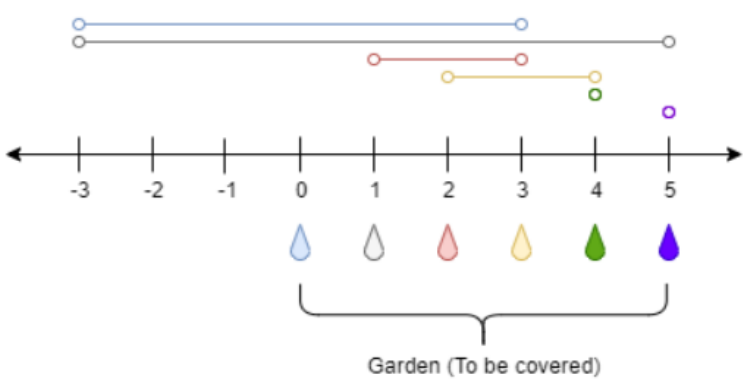
```
import java.util.Arrays;

public class Solution {
    public int[] countBits(int num) {
        int[] ret = new int[num + 1];
        Arrays.fill(ret, 0);
        for (int i = 1; i <= num; ++i)
            ret[i] = ret[i & (i - 1)] + 1;
        return ret;
    }
}
```

- Minimum number of Taps to open to water a Garden

There is a one-dimensional garden on the x-axis. The garden starts at the point 0 and ends at the point n. (i.e., the length of the garden is n).
There are n + 1 taps located at points [0, 1, ..., n] in the garden.
Given an integer n and an integer array ranges of length n + 1 where ranges[i] (0-indexed) means the i-th tap can water the area [i - ranges[i], i + ranges[i]] if it was open.
Return the minimum number of taps that should be open to water the whole garden, If the garden cannot be watered return -1.

Example 1:



Input: n = 5, ranges = [3,4,1,1,0,0]
Output: 1

Explanation: The tap at point 0 can cover the interval [-3,3]
The tap at point 1 can cover the interval [-3,5]
The tap at point 2 can cover the interval [1,3]
The tap at point 3 can cover the interval [2,4]
The tap at point 4 can cover the interval [4,4]
The tap at point 5 can cover the interval [5,5]
Opening Only the second tap will water the whole garden [0,5]

Example 2:

Input: n = 3, ranges = [0,0,0,0]
Output: -1

Explanation: Even if you activate all the four taps you cannot water the whole garden.

Constraints:

- 1 <= n <= 104
- ranges.length == n + 1
- 0 <= ranges[i] <= 100

Approach

- Create an array maxReach to store the farthest endpoints that can be reached from each starting point.
- Iterate through the ranges array:
- Calculate starting and ending points for each tap's reach.
- Update maxReach at the starting point with the corresponding ending point.
- Initialize variables for tap count (tap), current endpoint (currEnd), and next endpoint (nextEnd).
- Loop through indices 0 to n:
- If i is beyond nextEnd, return -1 as a gap cannot be covered.
- If i is beyond currEnd, increment tap and update currEnd to nextEnd.
- Update nextEnd with the maximum of its current value and maxReach[i].
- Return the value of tap as the minimum taps needed to cover the garden.

Complexity

- Time complexity: O(n)
- Space complexity: O(n)

Code

```
import java.util.Arrays;

public class Solution {
    public int minTaps(int n, int[] ranges) {
        int[] maxReach = new int[n + 1];

        for (int i = 0; i < ranges.length; ++i) {
            int s = Math.max(0, i - ranges[i]);
            int e = i + ranges[i];
            maxReach[s] = Math.max(maxReach[s], e);
        }

        int tap = 0;
        int currEnd = 0;
        int nextEnd = 0;

        for (int i = 0; i <= n; ++i) {
            if (i > nextEnd) {
                return -1;
            }
            if (i > currEnd) {
                tap++;
                currEnd = nextEnd;
            }
            nextEnd = Math.max(nextEnd, maxReach[i]);
        }

        return tap;
    }
}
```

PW SKILLS

THANK YOU !