IIT, BOMBAY
ALGONINJAS CONTEST 2.
TOPICS: RECURSION & DP

Question 1:
[9411]
Dynamic Programming
https://www.naukri.com/code360/problems/min-jumps_985273?interviewProblemRedirection=true&practice_topic%5B%5D=Dynamic%20Programming&difficulty%5B%5D=Easy

Question 2:
[23948]
Recursion : (Medium Level)
https://www.naukri.com/code360/problems/roll-number_7396629

Question 3:
[23527]
Dynamic Programming :
https://www.naukri.com/code360/problems/randomly-sorted_6868423

Question 4:
[10433]
Dynamic Programming
https://www.naukri.com/code360/problems/ninja-jasoos_1215014

Question 5:
[23783]
Dynamic Programming : (Hard Level)
https://www.naukri.com/code360/problems/oggy-and-cockkroaches_7100348

Question 6:
[24743]

Recursion: (Medium Level)
https://www.naukri.com/code360/problems/nearby-squares_7641754

Solutions for Contest 2
Q1. Minimum Jumps
C++

```
/*
    Time Complexity: O(3^(M*N))
    Space Complexity: O(N*M)

    Where N is the number of rows and M is the number of columns in the
    array.
*/

// Checking if the cell is valid or not.
bool isValidCell(int x, int y, int n, int m) {
    return (x < n && y < m);
}

int minCost(vector < vector < int >> & arr, int x, int y, int n, int m) {
    // Base case.
    if (x == n - 1 && y == m - 1){
       return 0;
    }

    // Finding the cost to go right from current cell.
    int rightCost = INT_MAX;

    if (isValidCell(x, y + 1, n, m)){
        rightCost = minCost(arr, x, y + 1, n, m) + abs(arr[x][y] - arr[x][y + 1]);
    }

    // Finding the cost to go down from current cell.
    int downCost = INT_MAX;
    if (isValidCell(x + 1, y, n, m)){
        downCost = minCost(arr, x + 1, y, n, m) + abs(arr[x][y] - arr[x + 1][y]);
    }

    // Finding the cost to go diagonally from current cell.
    int diagCost = INT_MAX;
    if (isValidCell(x + 1, y + 1, n, m)){
        diagCost = minCost(arr, x + 1, y + 1, n, m) + abs(arr[x][y] - arr[x + 1][y + 1]);
    }

    // Return minimum jumps
    return min({ downCost, rightCost, diagCost });
```

```
}

int findMinCost(vector < vector < int >> arr, int n, int m) {
    return minCost(arr, 0, 0, n, m);
}
```

Solution Using DP :

```
/*
    Time Complexity: O(N*M)
    Space Complexity: O(N*M)

    Where N is the number of rows and M is the number of columns in the array.
*/

int findMinCost(vector < vector < int >> arr, int n, int m) {

    vector < vector < int >> dp(n, vector < int > (m, INT_MAX - 1));
    dp[0][0] = 0;

    // Preprocessing dp array for first row and first column.
    for (int i = 1; i < m; i++) {
        dp[0][i] = dp[0][i - 1] + abs(arr[0][i] - arr[0][i - 1]);
    }

    for (int i = 1; i < n; i++) {
        dp[i][0] = dp[i - 1][0] + abs(arr[i][0] - arr[i - 1][0]);
    }

    // Finding the minimum cost of visiting each of adjacent arr.
    for (int i = 1; i < n; i++) {
        for (int j = 1; j < m; j++) {

            int leftCost = dp[i][j - 1] + abs(arr[i][j] - arr[i][j - 1]);

            int topCost = dp[i - 1][j] + abs(arr[i][j] - arr[i - 1][j]);
            int diagCost = dp[i - 1][j - 1] + abs(arr[i][j] - arr[i - 1][j - 1]);

            dp[i][j] = min({ leftCost, topCost, diagCost });
        }
    }

    // The answer will be stored at the last index.
```

```
    return dp[n - 1][m - 1];
}
```

Python

```python
'''
    Time Complexity: O(3^(N*M))
    Space Complexity: O(N*M)

    Where N is the number of rows and M is the number of columns in the array.
'''

def isValidCell(x, y, n, m):
    return x < n and y < m

def minCost(arr, x, y, n, m):
    if x == n - 1 and y == m - 1:
        return 0

    rightCost = 5 * (10**9)
    downCost = 5 * (10**9)
    diagCost = 5 * (10**9)

    if isValidCell(x, y + 1, n, m):
        rightCost = minCost(arr, x, y + 1, n, m) + abs(arr[x][y] - arr[x][y + 1])

    if isValidCell(x + 1, y, n, m):
        downCost = minCost(arr, x + 1, y, n, m) + abs(arr[x][y] - arr[x + 1][y])

    if isValidCell(x + 1, y + 1, n, m):
        diagCost = minCost(arr, x + 1, y + 1, n, m) + abs(arr[x][y] - arr[x + 1][y + 1])

    return min(rightCost, diagCost, downCost)


def findMinCost(arr, n, m):
    return minCost(arr, 0, 0, n, m)
```

Solution using DP :

```python
'''
    Time Complexity: O(N*M)
    Space Complexity: O(N*M)

    Where N is the number of rows and M is the number of columns in the array.
'''
```

```python
def findMinCost(arr, n, m):

    dp = [[5*(10**9) for j in range(m)] for i in range(n)]

    dp[0][0] = 0

    for i in range(1, m):
        dp[0][i] = dp[0][i-1] + abs(arr[0][i] - arr[0][i - 1])

    for i in range(1, n):
        dp[i][0] = dp[i - 1][0] + abs(arr[i][0] - arr[i - 1][0])

    for i in range(1, n):
        for j in range(1, m):
            leftCost = dp[i][j - 1] + \
                abs(arr[i][j] - arr[i][j - 1])
            topCost = dp[i - 1][j] + abs(arr[i][j] - arr[i - 1][j])
            diagCost = dp[i - 1][j - 1] + \
                abs(arr[i][j] - arr[i - 1][j - 1])
            dp[i][j] = min({leftCost, topCost, diagCost})

    return dp[n - 1][m - 1]
```

Java

```java
/*
    Time Complexity: O(3^(M*N))
    Space Complexity: O(N*M)

    Where N is the number of rows and M is the number of columns in the array.
*/

public class Solution {
  // Checking if the cell is valid or not.
  public static boolean isValidCell(int x, int y, int n, int m) {
    return (x < n && y < m);
  }

  public static int minCost(int[][] arr, int x, int y, int n, int m) {
    // Base case.
    if (x == n - 1 && y == m - 1){
      return 0;
    }
```

```
    // Finding the cost to go right from current cell.
    int rightCost = Integer.MAX_VALUE;

    if (isValidCell(x, y + 1, n, m)){
      rightCost = minCost(arr, x, y + 1, n, m) + Math.abs(arr[x][y] - arr[x][y + 1]);
    }

    // Finding the cost to go down from current cell.
    int downCost = Integer.MAX_VALUE;

    if (isValidCell(x + 1, y, n, m)){
      downCost = minCost(arr, x + 1, y, n, m) + Math.abs(arr[x][y] - arr[x + 1][y]);
    }

    // Finding the cost to go diagonally from current cell.
    int diagCost = Integer.MAX_VALUE;

    if (isValidCell(x + 1, y + 1, n, m)){
      diagCost = minCost(arr, x + 1, y + 1, n, m) + Math.abs(arr[x][y] - arr[x + 1][y + 1]);
    }

    // return minimum jumps
    return Math.min(Math.min(downCost, rightCost), diagCost);
  }

  public static int findMinCost(int[][] arr, int n, int m) {
    return minCost(arr, 0, 0, n, m);
  }


}
```

Solution using DP:

```
/*
    Time Complexity: O(N*M)
    Space Complexity: O(N*M)

    Where N is the number of rows and M is the number of columns in the array.
*/

public class Solution {

  public static int findMinCost(int[][] arr, int n, int m) {
```

```
        int dp[][] = new int[n][m];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                dp[i][j] = Integer.MAX_VALUE;
            }
        }

        dp[0][0] = 0;

        // Preprocessing dp array for first row and first column.
        for (int i = 1; i < m; i++) {
            dp[0][i] = dp[0][i - 1] + Math.abs(arr[0][i] - arr[0][i - 1]);
        }

        for (int i = 1; i < n; i++) {
            dp[i][0] = dp[i - 1][0] + Math.abs(arr[i][0] - arr[i - 1][0]);
        }

        // Finding the minimum cost of visiting each of adjacent arr.
        for (int i = 1; i < n; i++) {
            for (int j = 1; j < m; j++) {

                int leftCost =
                    dp[i][j - 1] + Math.abs(arr[i][j] - arr[i][j - 1]);

                int topCost =
                    dp[i - 1][j] + Math.abs(arr[i][j] - arr[i - 1][j]);

                int diagCost = dp[i - 1][j - 1] +
                    Math.abs(arr[i][j] - arr[i - 1][j - 1]);

                dp[i][j] = Math.min(Math.min(leftCost, topCost), diagCost);
            }
        }
        // The answer will be stored at the last index.
        return dp[n - 1][m - 1];
    }
}
```

Q2. Roll Numbers

C++

```
/*
    Time Complexity: O(2^K)
    Space Complexity: O(2^K)

    where 'K' is the number of operations.
*/

/*
    'recur(id, val)' where the 'id' is the current operation number and 'val'
    is the current value of 'x', it maintains unique values of 'x' in the unordered_set.
*/
void recur(int id, long long val, int &k, unordered_set<long long> &us) {

    // Check if the there have been 'k' operations.
    if (id == k) {

        // If so return.
        return;
    }

    // Initialize variables to represent the two possible operations.
    long long opt1 = (val + 1) / 2;
    long long opt2 = (val - 1) * 2;

    // Insert the values obtained by both operations in the unordered_set.
    us.insert(opt1);
    us.insert(opt2);

    // Recurse to the next operation with both the possible values of 'x'.
    recur(id + 1, opt1, k, us);
    recur(id + 1, opt2, k, us);
}

int rollNumbers(int x, int k) {

    // Initialize unordered_set 'us' to store the unique values of 'x'.
    unordered_set<long long> us;

    // Insert the value 'x' into the unordered_set 'us'.
    us.insert(x);

    // Start the recursive function.
    recur(0, x, k, us);
```

```
    // Return the length of 'us'.
    return us.size();
}
```

Java

```java
/*
    Time Complexity: O(2^K)
    Space Complexity: O(2^K)

    where 'K' is the number of operations.
*/

import java.util.HashMap;
import java.util.HashSet;

public class Solution {

    /*
        'recur(id, val)' where the 'id' is the current operation number and 'val'
        is the current value of 'x', it maintains unique values of 'x' in the unordered_set.
    */
    static void recur(int id, long val, int k, HashSet<Long> us) {

        // Check if the there have been 'k' operations.
        if (id == k) {

            // If so return.
            return;
        }

        // Initialize variables to represent the two possible operations.
        long opt1 = (val + 1) / 2;
        long opt2 = (val - 1) * 2;

        // Insert the values obtained by both operations in the unordered_set.
        us.add(opt1);
        us.add(opt2);

        // Recurse to the next operation with both the possible values of 'x'.
        recur(id + 1, opt1, k, us);
        recur(id + 1, opt2, k, us);
    }
```

```
    static int rollNumbers(int x, int k) {
        // Initialize unordered_set 'us' to store the unique values of 'x'.
        HashSet<Long> us = new HashSet<>();

        // Insert the value 'x' into the unordered_set 'us'.
        us.add((long) x);

        // Start the recursive function.
        recur(0, x, k, us);

        // Return the length of 'us'.
        return us.size();
    }
}
```

Python

```
"""
    Time Complexity: O(2^K)
    Space Complexity: O(2^K)

    where 'K' is the number of operations.
"""


# Define 'recur(moveId, val, k, us)' where the 'moveId' is the current operation number and 'val'
is the current value of 'x', 'us' maintains unique values of 'x' in the unordered_set, 'k' is given in
problem.
def recur(moveId, val, k, us):
    # Check if the there have been 'k' operations.
    if moveId == k:
        return

    # Initialize variables to represent the two possible operations.
    opt1 = (val+1)//2

    # Handling case when 'val' is negative, because in python '(-5/2) = -3' but we want -2 as
desired value.
    if val < 0:
        # Treating 'val' as positive integer and then multiply the final answer with -1.
        opt1 = -1*((-1*(val+1))//2)
    opt2 = (val-1)*2
```

```
    # Insert the values obtained by both operations in the unordered_set.
    us.add(opt1)
    us.add(opt2)

    # Recurse to the next operation with both the possible values of 'x'.
    recur(moveId+1, opt1, k, us)
    recur(moveId+1, opt2, k, us)

def rollNumbers(x: int, k: int) -> int:
    # Initialize unordered_set 'us' to store the unique values of 'x'.
    us = set()

    # Insert the value 'x' into the unordered_set 'us'.
    us.add(x)

    # Start the recursive function.
    recur(0, x, k, us)

    # Return the length of 'us'.
    return len(us)
```

Q3. Randomly Sorted

C++

```
/*
    Time Complexity: O(N * M^2)
    Space Complexity: O(N * M)

    where 'N' is the length of array 'A' and 'M' is the maximum value allowed in 'A'.
*/

/* Helper function to compute the Multiplicative Inverse
of all natural numbers from 1 to 'N' */
int modInverse(long long n, long long mod) {
    vector<long long> multiplicativeInverse(n + 1);
    multiplicativeInverse[0] = multiplicativeInverse[1] = 1;

    for (int i = 2; i <= n; i++) {
        multiplicativeInverse[i] = multiplicativeInverse[mod % i] * (mod - mod / i) % mod;
    }

    return multiplicativeInverse[n];
}
```

```
int randomlySorted(int n, int m) {

    // Initialize helper variables 'mod' and 'invM'.
    int mod = 1e9 + 7, invM = modInverse(m, mod);

    // Initialize a 2d array 'dp'.
    long long dp[n][m];
    memset(dp, 0, sizeof(dp));

    // Set the base case.
    for (int j = 0; j < m; j++) {
        dp[0][j] = invM;
    }

    // Iterate over all 'i' from 1 to 'N - 1'.
    for (int i = 1; i < n; i++) {

        // Iterate over all possible values of 'A[i]'.
        for (int j = 0; j < m; j++) {

            // Iterate over all possible values of 'A[i - 1]' that keep the array sorted.
            for (int k = 0; k <= j; k++) {

                // Transition from 'dp[i - 1][k]' to 'dp[i][j]'.
                dp[i][j] = (dp[i][j] + dp[i - 1][k] * invM) % mod;
            }
        }
    }

    // Initialize a variable 'ans' to store the final answer.
    int ans = 0;

    for (int j = 0; j < m; j++) {

        // Add the probability of ('A' being sorted taking 'A[n - 1] = j') to 'ans'.
        ans = (ans + dp[n - 1][j]) % mod;
    }

    // Return the final probability.
    return ans;
}
```

Solution Using DP :

```
/*
    Time Complexity: O(N * M)
    Space Complexity: O(N * M)

    where 'N' is the length of array 'A' and 'M' is the maximum value allowed in 'A'.
*/

// Helper function to compute the Multiplicative Inverse.
int modInverse(long long n, long long mod) {
    vector<long long> multiplicativeInverse(n + 1);
    multiplicativeInverse[0] = multiplicativeInverse[1] = 1;

    for (int i = 2; i <= n; i++) {
        multiplicativeInverse[i] = multiplicativeInverse[mod % i] * (mod - mod / i) % mod;
    }

    return multiplicativeInverse[n];
}

int randomlySorted(int n, int m) {

    // Initialize helper variables 'mod' and 'invM'.
    int mod = 1e9 + 7, invM = modInverse(m, mod);

    // Initialize two 2d arrays 'dp' and 'pref'.
    long long dp[n][m], pref[n][m];

    // Set the base case.
    dp[0][0] = invM;
    pref[0][0] = dp[0][0];
    for (int j = 1; j < m; j++) {
        dp[0][j] = invM;
        pref[0][j] = (pref[0][j - 1] + dp[0][j]) % mod;
    }

    // Iterate over all 'i' from 1 to 'N - 1'.
    for (int i = 1; i < n; i++) {

        // Transition from 'i - 1' to 'i' and maintain prefix sum of 'dp[i]' in 'pref[i]'.

        dp[i][0] = pref[i - 1][0] * invM % mod;
        pref[i][0] = dp[i][0];

        for (int j = 1; j < m; j++) {
```

```
            dp[i][j] = pref[i - 1][j] * invM % mod;
            pref[i][j] = (pref[i][j - 1] + dp[i][j]) % mod;
        }
    }

    // Return the final probability.
    return pref[n - 1][m - 1];
}
```

Python

```
"""
    Time Complexity: O(N * M)
    Space Complexity: O(N * M)

    where 'N' is the length of array 'A' and 'M' is the maximum value allowed in 'A'.
"""

from typing import *

# Helper function to compute the Multiplicative Inverse.
def modInverse(n, mod):
    multiplicativeInverse = [0 for _ in range(n + 1)]
    multiplicativeInverse[0] = multiplicativeInverse[1] = 1

    for i in range(2, n + 1):
        multiplicativeInverse[i] = (multiplicativeInverse[mod % i] * (mod - mod // i)) % mod

    return int(multiplicativeInverse[n])

def randomly_sorted(n: int, m: int) -> int:

    # Initialize helper variables 'mod' and 'invM'.
    mod = int(1e9) + 7
    invM = modInverse(m, mod)

    # Initialize a 2d array 'dp'.
    dp = [[0 for _ in range(m)] for _ in range(n)]

    # Set the base case.
    for j in range(0, m):
        dp[0][j] = invM
```

```
    # Iterate over all 'i' from 1 to 'N - 1'.
    for i in range(1, n):

        # Iterate over all possible values of 'A[i]'.
        for j in range(0, m):

            # Iterate over all possible values of 'A[i - 1]' that keep the array sorted.
            k = 0
            while k <= j:
                # Transition from 'dp[i - 1][k]' to 'dp[i][j]'.
                dp[i][j] = (dp[i][j] + dp[i - 1][k] * invM) % mod
                k += 1

    # Initialize a variable 'ans' to store the final answer.
    ans = 0

    for j in range(0, m):
        # Add the probability of ('A' being sorted taking 'A[n - 1] = j') to 'ans'.
        ans = (ans + dp[n - 1][j]) % mod

    # Return the final probability.
    return ans
```

Solutions using DP :

```
"""
    Time Complexity: O(N * M)
    Space Complexity: O(N * M)

    where 'N' is the length of array 'A' and 'M' is the maximum value allowed in 'A'.
"""

from typing import *

# Helper function to compute the Multiplicative Inverse.
def modInverse(n, mod):
    multiplicativeInverse = [0 for _ in range(n + 1)]
    multiplicativeInverse[0] = multiplicativeInverse[1] = 1

    for i in range(2, n + 1):
        multiplicativeInverse[i] = (multiplicativeInverse[mod % i] * (mod - mod // i)) % mod

    return int(multiplicativeInverse[n])

def randomly_sorted(n: int, m: int) -> int:
```

```
    # Initialize helper variables 'mod' and 'invM'.
    mod = int(1e9) + 7
    invM = modInverse(m, mod)

    # Initialize two 2d arrays 'dp' and 'pref'.
    dp = [[0 for _ in range(m)] for _ in range(n)]
    pref = [[0 for _ in range(m)] for _ in range(n)]

    # Set the base case.
    dp[0][0] = invM
    pref[0][0] = dp[0][0]
    for j in range(1, m):
        dp[0][j] = invM
        pref[0][j] = (pref[0][j - 1] + dp[0][j]) % mod

    # Iterate over all 'i' from 1 to 'N - 1'.
    for i in range(1, n):

        # Transition from 'i - 1' to 'i' and maintain prefix sum of 'dp[i]' in 'pref[i]'.

        dp[i][0] = (pref[i - 1][0] * invM) % mod
        pref[i][0] = dp[i][0]

        for j in range(1, m):
            dp[i][j] = (pref[i - 1][j] * invM) % mod
            pref[i][j] = (pref[i][j - 1] + dp[i][j]) % mod

    # Return the final probability.
    return int(pref[n - 1][m - 1])
```

Java

```
/*
    Time Complexity: O(N * M^2)
    Space Complexity: O(N * M)

    where 'N' is the length of array 'A' and 'M' is the maximum value allowed in 'A'.
*/

import java.util.*;
public class Solution {
```

```
/* Helper function to compute the Multiplicative Inverse
of all natural numbers from 1 to 'N' */
static int modInverse(int n, int mod) {
    long[] multiplicativeInverse = new long[n + 1];
    multiplicativeInverse[0] = 1;
    multiplicativeInverse[1] = 1;

    for (int i = 2; i <= n; i++) {
        multiplicativeInverse[i] = multiplicativeInverse[mod % i] * (mod - mod / i) % mod;
    }

    return (int)multiplicativeInverse[n];
}

static int randomlySorted(int n, int m) {

    // Initialize helper variables 'mod' and 'invM'.
    int mod = 1000000007;
    int invM = modInverse(m, (int)mod);

    long[][] dp = new long[n][m];

    // Set the base case.
    for (int j = 0; j < m; j++) {
        dp[0][j] = invM;
    }

    // Iterate over all 'i' from 1 to 'N - 1'.
    for (int i = 1; i < n; i++) {

        // Iterate over all possible values of 'A[i]'.
        for (int j = 0; j < m; j++) {

            // Iterate over all possible values of 'A[i - 1]' that keep the array sorted.
            for (int k = 0; k <= j; k++) {

                // Transition from 'dp[i - 1][k]' to 'dp[i][j]'.
                dp[i][j] = (dp[i][j] + dp[i - 1][k] * invM) % mod;
            }
        }
    }

    // Initialize a variable 'ans' to store the final answer.
```

```
        long ans = 0;

        for (int j = 0; j < m; j++) {

            // Add the probability of ('A' being sorted taking 'A[n - 1] = j') to 'ans'.
            ans = (ans + dp[n - 1][j]) % mod;
        }

        // Return the final probability.
        return (int)ans;
    }
}
```

Solutions Using DP :

```
/*
    Time Complexity: O(N * M)
    Space Complexity: O(N * M)

    where 'N' is the length of array 'A' and 'M' is the maximum value allowed in 'A'.
*/

import java.util.*;
public class Solution {

    /* Helper function to compute the Multiplicative Inverse
    of all natural numbers from 1 to 'N' */
    static int modInverse(int n, int mod) {
        long[] multiplicativeInverse = new long[n + 1];
        multiplicativeInverse[0] = 1;
        multiplicativeInverse[1] = 1;

        for (int i = 2; i <= n; i++) {
            multiplicativeInverse[i] = multiplicativeInverse[mod % i] * (mod - mod / i) % mod;
        }

        return (int)multiplicativeInverse[n];
    }

    static int randomlySorted(int n, int m) {

        // Initialize helper variables 'mod' and 'invM'.
        int mod = 1000000007, invM = modInverse(m, mod);

        // Initialize two 2d arrays 'dp' and 'pref'.
        long[][] dp = new long[n][m];
```

```
        long[][] pref = new long[n][m];

        // Set the base case.
        dp[0][0] = invM;
        pref[0][0] = dp[0][0];
        for (int j = 1; j < m; j++) {
            dp[0][j] = invM;
            pref[0][j] = (pref[0][j - 1] + dp[0][j]) % mod;
        }

        // Iterate over all 'i' from 1 to 'N - 1'.
        for (int i = 1; i < n; i++) {

            // Transition from 'i - 1' to 'i' and maintain prefix sum of 'dp[i]' in 'pref[i]'.
            dp[i][0] = pref[i - 1][0] * invM % mod;
            pref[i][0] = dp[i][0];

            for (int j = 1; j < m; j++) {

                dp[i][j] = pref[i - 1][j] * invM % mod;
                pref[i][j] = (pref[i][j - 1] + dp[i][j]) % mod;
            }
        }

        // Return the final probability.
        return (int)pref[n - 1][m - 1];
    }
}
```

Q4. NINJA JASOOS (definitely a freebie question, that’s why I mentioned to go through all the questions)

C++

```
/*
    Time complexity: O(N)
    Space complexity: O(1)

    Where 'N' reprents the "Nth" number .
*/

int ninjaJasoos(int n) {
    int a = 0, b = 1, c;
    // Iterating upto the end.
```

```
    for (int i = 2; i <= n; i++) {
        c = a + b;
        // Updating values.
        a = b;
        b = c;
    }
    return b;
}
```

Java

```
/*
    Time complexity: O(N)
    Space complexity: O(1)

    Where 'N' reprents the "Nth" number .
*/

public class Solution {
    public static int ninjaJasoos(int n) {
        int a = 0, b = 1, c;
        // Iterating upto the end.
        for (int i = 2; i <= n; i++) {
            c = a + b;
            // Updating values.
            a = b;
            b = c;
        }
        return b;
    }
}
```

Python

```
'''
        Time complexity: O(N)
    Space complexity: O(1)

    Where 'N' reprents the "Nth" number .
'''

def ninjaJasoos(n):
    a = 0
    b = 1

    # Iterating upto the end.
```

```
    for i in range(2, n+1):
        c = a + b

        # Updating values.
        a = b
        b = c

    return b
```

Q5. Oggy and Cockroaches
C++

```
/*
    Time complexity : O(n ^ 2).
    Space complexity : O(n).
    where 'n' is the the number of cockroaches.

*/
#include <iostream>
#include <vector>
using namespace std;
typedef long long ll;

int maxCoin(int n, vector<int> &x, vector<int> &a) {


    // Initialise two vectors 'curr' and 'prev' to store the number of coins for all positions at current
time 't' and time 't-1' respectively.
    vector<int>curr(n + 2, 0), prev(n + 2, 0);


    // Run a for loop for time 't'= 1 to 't'='n'.
    for (int t = 1; t <= n; t++) {

        // Make vector 'prev' equal to 'curr'.
        prev = curr;

        // Initialise 'curr' of size 'n+2' with 0.
        curr = vector<int>(n + 2, 0);

        // Run a for loop for coordinate 'i'= 1 to 'i'<='t'.
        for (int i = 1; i <= t; i++) {
```

```
            // If 'x[t-1]' is equal to 'i'.
            if (x[t - 1] == i) {

                // Add 'a[t-1]' in 'curr[i]'.
                curr[i] += a[t - 1];
            }

            // Add 'max(prev[i], prev[i-1], prev[i+1])' in 'curr[i]'.
            curr[i] += max(prev[i],  max(prev[i - 1], prev[i + 1]));

        }
    }

    // Initialise a variable 'ans' to store the maximum number of coins Oggy collect.
    int ans = 0;

    // Run a for loop for all coordinate from 0 to 'n'.
    for (int i : curr) {

        // Make 'ans' = 'max(ans, i)'.
        ans = max(ans, i);
    }

    // Return ans.
    return ans;

}
```

Python

```
"""
    Time complexity : O(n ^ 2).
    Space complexity : O(n).
    where 'n' is the the number of cockroaches.
"""



def maxCoin(n: int, x: list, a: list) -> int:
    # Initialise two vectors 'curr' and 'prev' to store the number of coins for all positions at current time 't' and time 't-1' respectively.
```

```
    curr = [0]*(n+2)
    prev = [0]*(n+2)

    # Run a for loop for time 't'= 1 to 't'='n'.
    for t in range(1,n+1):
        # Make vector 'prev' equal to 'curr'.
        prev = curr
        # Initialise 'curr' of size 'n+2' with 0.
        curr = [0]*(n+2)
        # Run a for loop for coordinate 'i'= 1 to 'i'<='t'.
        for i in range(1,t+1):
            # If 'x[t-1]' is equal to 'i'.
            if x[t-1] == i:
                # Add 'a[t-1]' in 'curr[i]'.
                curr[i] += a[t-1]
            # Add 'max(prev[i], prev[i-1], prev[i+1])' in 'curr[i]'.
            curr[i] += max(prev[i], max(prev[i-1], prev[i+1]))

    # Initialise a variable 'ans' to store the maximum number of coins Oggy collect.
    ans = 0
    # Run a for loop for all coordinate from 0 to 'n'.
    for i in curr:
        # Make 'ans' = 'max(ans, i)'.
        ans = max(ans,i)
    # Return ans.
    return ans
```

Java

```
/*
    Time complexity : O(n ^ 2).
    Space complexity : O(n).
    where 'n' is the the number of cockroaches.

*/

public class Solution {
    static int maxCoin(int n, int []x, int []a) {

        // Initialise two vectors 'curr' and 'prev' to store the number of coins for all positions at
current time 't' and time 't-1' respectively.
        int []curr = new int[n + 2];
        int []prev = new int[n + 2];
        for (int i = 0; i <= n + 1; ++i) {
```

```
        curr[i] = prev[i] = 0;
    }


    // Run a for loop for time 't'= 1 to 't'='n'.
    for (int t = 1; t <= n; t++) {

        // Make vector 'prev' equal to 'curr'.
        prev = curr;

        // Initialise 'curr' of size 'n+2' with 0.
        curr = new int[n + 2];
        for (int j = 0; j <= n + 1; ++j) {
            curr[j] = 0;
        }

        // Run a for loop for coordinate 'i'= 1 to 'i'<='t'.
        for (int i = 1; i <= t; i++) {

            // If 'x[t-1]' is equal to 'i'.
            if (x[t - 1] == i) {

                // Add 'a[t-1]' in 'curr[i]'.
                curr[i] += a[t - 1];
            }

            // Add 'max(prev[i], prev[i-1], prev[i+1])' in 'curr[i]'.
            curr[i] += Math.max(prev[i],  Math.max(prev[i - 1], prev[i + 1]));

        }
    }

    // Initialise a variable 'ans' to store the maximum number of coins Oggy collect.
    int ans = 0;

    // Run a for loop for all coordinate from 0 to 'n'.
    for (int i : curr) {

        // Make 'ans' = 'max(ans, i)'.
        ans = Math.max(ans, i);
    }

    // Return ans.
    return ans;
```

```
    }
}
```

Q6. Nearby Squares
C++

```
/*
    Time Complexity: O(2^N)
    Space Complexity: O(N)

    where 'N' is the length of the array 'A'.
*/

long long recur(int i, int sumB, int sumC, int n, vector<int> &a) {

    // Check if all 'N' elements have been distributed.
    if (i == n) {

        // Return the absolute difference between the scores of 'B' and 'C'.
        return abs(1LL * sumB * sumB - 1LL * sumC * sumC);
    }

    // Initialize a variable 'res' to store the current answer.
    long long res = INT64_MAX;

    // Put the 'i-th' element in 'B'.
    res = min(res, recur(i + 1, sumB + a[i], sumC, n, a));

    // Put the 'i-th' element in 'C'.
    res = min(res, recur(i + 1, sumB, sumC + a[i], n, a));

    // Return the current answer.
    return res;
}

long long nearbySquares(int n, vector<int> &a) {

    // Call 'recur(0, 0, 0)' and return the result as the final answer.
    return recur(0, 0, 0, n, a);
}
```

Python

```
"""
    Time Complexity: O(2^N)
    Space Complexity: O(N)

    where 'N' is the length of the array 'A'.
"""

from typing import *

def recur(i, sumB, sumC, n, a):
    # Check if all 'N' elements have been distributed.
    if i == n:
        # Return the absolute difference between the scores of 'B' and 'C'.
        return abs(sumB * sumB - sumC * sumC)

    # Initialize a variable 'res' to store the current answer.
    res = float('inf')

    # Put the 'i-th' element in 'B'.
    res = min(res, recur(i + 1, sumB + a[i], sumC, n, a))

    # Put the 'i-th' element in 'C'.
    res = min(res, recur(i + 1, sumB, sumC + a[i], n, a))

    # Return the current answer.
    return res

def nearbySquares(n: int, a: List[int]) -> int:
    # Call 'recur(0, 0, 0)' and return the result as the final answer.
    return recur(0, 0, 0, n, a)
```

Java

```
/*
    Time Complexity : O(2^N)
    Space Complexity : O(1)

    Where 'N' is the length of the array 'A'.
*/

public class Solution {
```

```
    static long recur(int i, int sumB, int sumC, int n, int[] a) {

        // Check if all 'N' elements have been distributed.
        if (i == n) {

            // Return the absolute difference between the scores of 'B' and 'C'.
            return Math.abs((long)1 * (long)sumB * (long)sumB - (long)1 * (long)sumC *
(long)sumC);
        }

        // Initialize a variable 'res' to store the current answer.
        long res = Long.MAX_VALUE;

        // Put the 'i-th' element in 'B'.
        res = Math.min(res, recur(i + 1, sumB + a[i], sumC, n, a));

        // Put the 'i-th' element in 'C'.
        res = Math.min(res, recur(i + 1, sumB, sumC + a[i], n, a));

        // Return the current answer.
        return res;
    }

    static long nearbySquares(int n, int []a) {

        // Call 'recur(0, 0, 0)' and return the result as the final answer.
        return recur(0, 0, 0, n, a);
    }
}
```