Dynamic Programming - 1

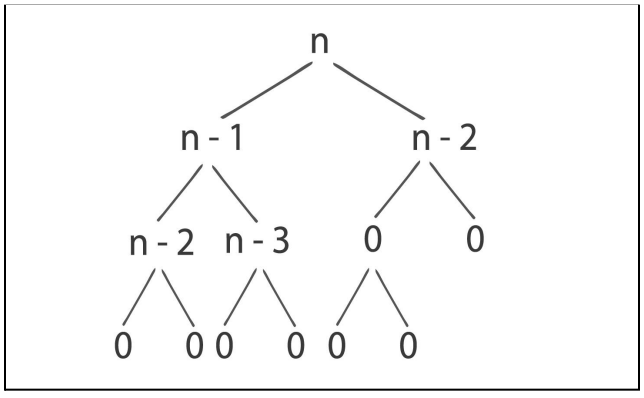
Introduction

Suppose we need to find the n^{th} Fibonacci number using recursion that we have already found out in our previous sections. Let's directly look at its code:

```
int fibo(int n) {
    if(n <= 1) {
        return n;
    }
    int a = fibo(n - 1);
    int b = fibo(n - 2);
    return a + b;
}
```

Here, for every n, we need to make a recursive call to f(n-1) and f(n-2).

For f(n-1), we will again make the recursive call to f(n-2) and f(n-3). Similarly, for f(n-2), recursive calls are made on f(n-3) and f(n-4) until we reach the base case. The recursive call diagram will look something like shown below:

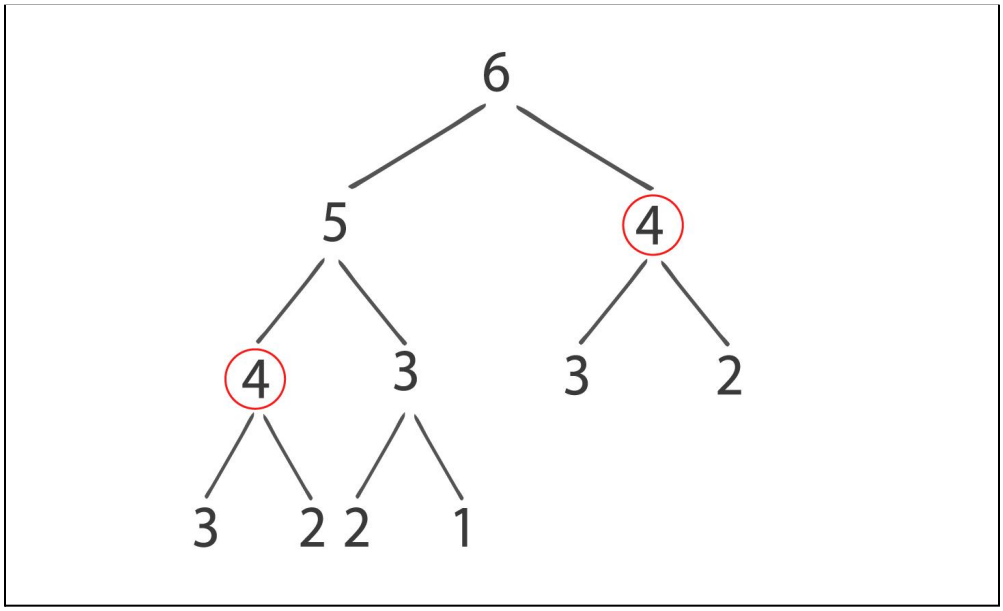


At every recursive call, we are doing constant work(k)(addition of previous outputs to obtain the current one). At every level, we are doing $2^n K$ work (where n = 0, 1, 2, ...). Since reaching 1 from n will take n calls, therefore, at the last level, we are doing $2^{n-1}k$ work. Total work can be calculated as:

$$(2^0 + 2^1 + 2^2 + ... + 2^{n-1}) * k \simeq 2^n k$$

Hence, it means time complexity will be $O(2^n)$.

We need to improve this complexity as it is terrible. Let's look at the example below for finding the 6^{th} Fibonacci number.



IMPORTANT OBSERVATION: We can observe that there are repeating recursive calls made over the entire program. As in the above figure, for calculating f(5), we need the value of f(4) (first recursive call over f(4)), and for calculating f(6), we again need the value of f(4)(second similar recursive call over f(4)). Both of these recursive calls are shown above in the outlining circle. Similarly, there are many others for

which we are repeating the recursive calls. Generally, in recursion, there are repeated recursive calls, which increases the time complexity of the program.

To overcome this problem, we will store the output of previously encountered values(preferably in arrays as these are most efficient to traverse and extract data). Next time whenever we will be making the recursive calls over these values, we will directly consider their already stored outputs and then use these in our calculations instead of calculating them over again. This way, we can improve the running time of our code. This process of storing each recursive call's output and then using them for further calculations preventing the code from calculating these again is called **memoization**.

To achieve this in our example we will simply take an answer array initialized to -1. Now while making a recursive call, we will first check if the value stored in this answer array corresponding to that position is -1 or not. If it is -1, it means we haven't calculated the value yet and need to proceed further by making recursive calls for the respective value. After obtaining the output, we need to store this in the answer array so that next time, if the same value is encountered, it can be directly used from this answer array.

Now in this process of memoization, considering the above Fibonacci numbers example, it can be observed that the total number of unique calls will be at most (n+1) only.

Let's look at the memoization code for Fibonacci numbers below:

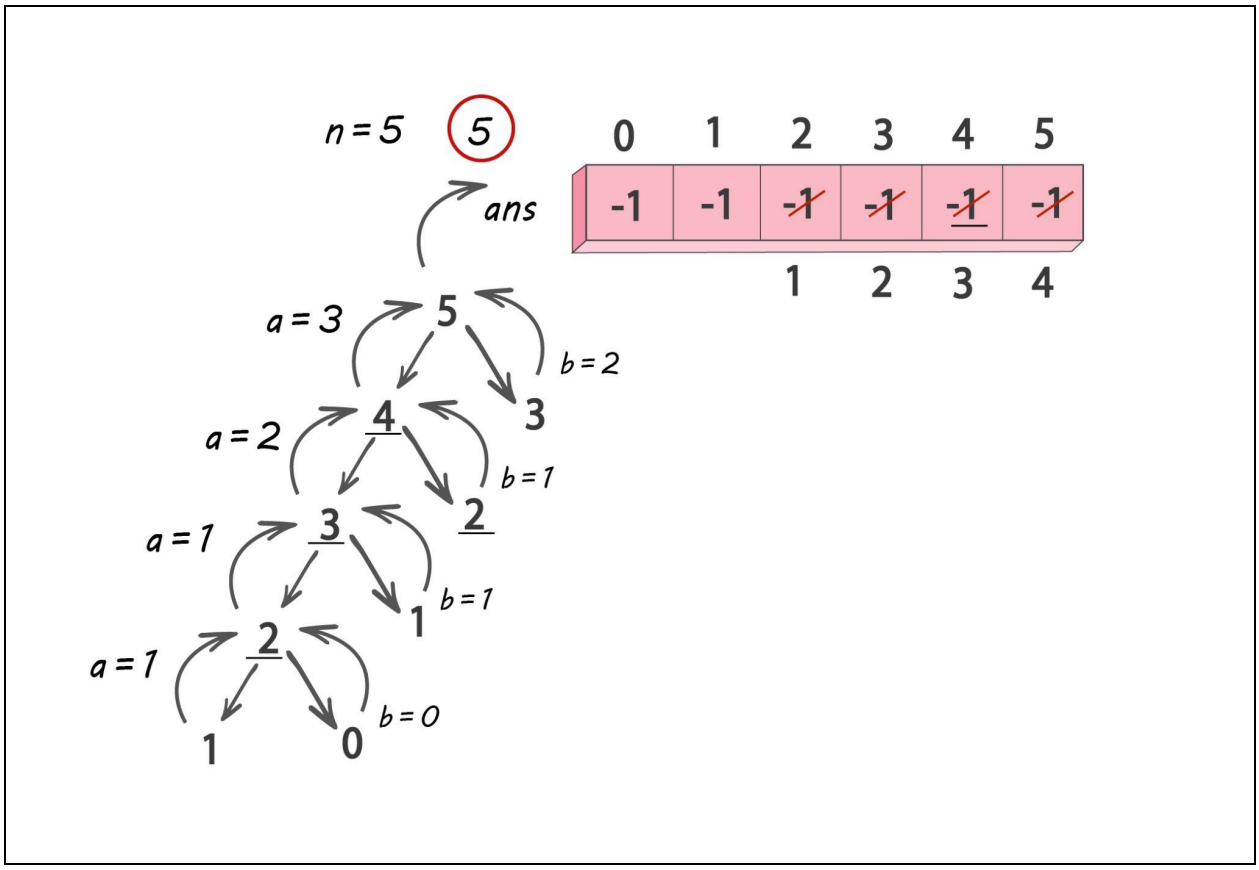
```
int fibo_helper(int n, int *ans) {
    if(n <= 1) {            // Base case
        return n;
    }

    // Check if output already exists
    if(ans[n] != -1) {
```

```
        return ans[n];
    }

    // Calculate output
    int a = fibo_helper(n-1, ans);
    int b = fibo_helper(n-2, ans);

    // Save the output for future use
    ans[n] = a + b;

    // Return the final output
    return ans[n];
}


int fibo_2(int n) {
    int *ans = new int[n+1];

    for(int i = 0; i <= n; i++) {       // -1 marks that answer for
corresponding
        ans[i] = -1;                    // number does not exist
    }
    return fibo_helper(n, ans);         // Now simply following the
memoization
}
```

Let's dry run for n = 5, to get a better understanding:

Again, if we observe, we can see that for any number we are not able to make a recursive call on the right side of it which means that we can make at most 5+1 = 6 (n+1) unique recursive calls which reduce the time complexity to O(n) which is highly optimized as compared to simple recursion.

Summary: Memoization is a **top-down approach** where we



save the previous answers so that they can be used to calculate the future answers and improve the time complexity to a greater extent.

Finally, what we are doing is making a recursive call to every index of the answer array and calculating the value for it using previous outputs stored. Recursive calls terminate over the base case, which means we are already aware of the answers that should be stored in the base case's indexes. In cases of Fibonacci numbers, these indexes are 0 and 1 as f(0) = 0 and f(1) = 1. So we can directly allot these two values to our answer array and then use these to calculate f(2), which is f(1) + f(0), and so on for every other index. This can be simply done iteratively by running a loop from i = (2 to n). Finally, we will get our answer at the 5^{th} index of the answer array as we already know that the i-th index contains the answer to the i-th value.

Simply, we are first trying to figure out the dependency of the current value on the previous values and then using them to calculate our new value. Now, we are looking for those values which do not depend on other values, which means they are independent (the base case's values as these are the smallest problems about which we are already aware of). Finally, following a **bottom-up approach** to reach the desired index. This approach of converting recursion into iteration is known as **Dynamic programming(DP).**

Let us now look at the code for calculating the n^{th} Fibonacci number:

```
int fibo_3(int n) {
    int *ans = new int[n+1];

    ans[0] = 0;    // Storing the independent values in the solution array.
    ans[1] = 1;

    for(int i = 2; i <= n; i++) { //Following bottom-up approach to reach n
        ans[i] = ans[i-1] + ans[i-2];
    }
    int output = arr[n];
    delete [] arr;
    return output;                  // final answer
}
```

Note: Generally, memoization is a recursive approach, and DP is an iterative approach.

For all further problems, we will do the following:

1. Figure out the most straightforward approach for solving a problem using recursion.
2. Now, try to optimize the recursive approach by storing the previous outputs using memoization.

3. Finally, replace recursion by iteration using dynamic programming. (The best possible solution that seems to appear for every problem when in case of recursion the space complexity is more in comparison to iteration)

Alphacode

Problem Statement: Alice and Bob need to send secret messages to each other and are discussing ways to encode their messages:

Alice: *"Let's just use a very simple code: We'll assign 'A' the code word 1, 'B' will be 2, and so on down to 'Z' being assigned 26."*

Bob: *"That's a stupid code, Alice. Suppose I send you the word 'BEAN' encoded as 25114. You could decode that in many different ways!"*

Alice: *"Sure you could, but what words would you get? Other than 'BEAN', you'd get 'BEAAD', 'YAAD', 'YAN', 'YKD' and 'BEKD'. I think you would be able to figure out the correct decoding. And why would you send me the word 'BEAN' anyway?"*

Bob: *"OK, maybe that's a bad example, but I bet you that if you got a string of length 5000 there would be tons of different decodings and with that many you would find at least two different ones that would make sense."*

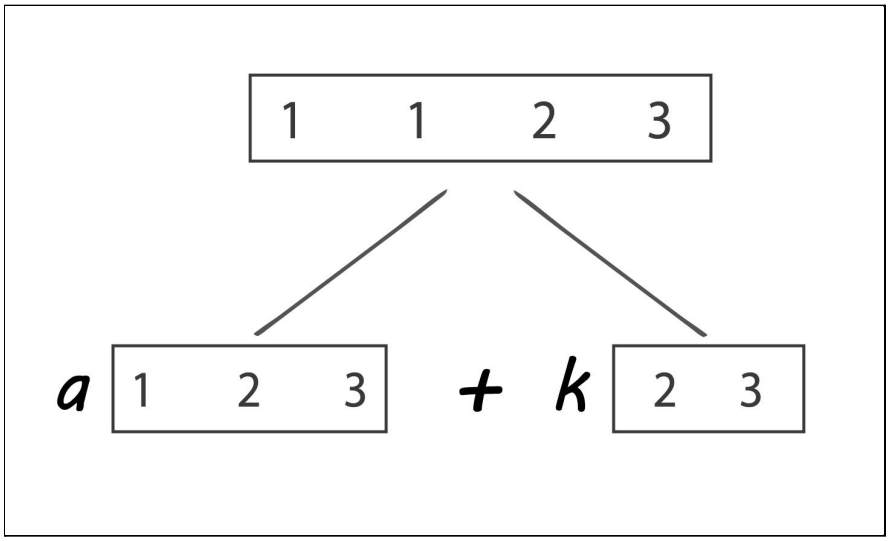
Alice: *"How many different decodings?"*

Bob: *"Jillions!"*

For some reason, Alice is still unconvinced by Bob's argument, so she requires a program that will determine how many decodings there can be for a given string using her code.

Explanation:

Example: Let's take a message **1 1 2 3.** Now this message can be reduced or decoded in two different ways by recursion:



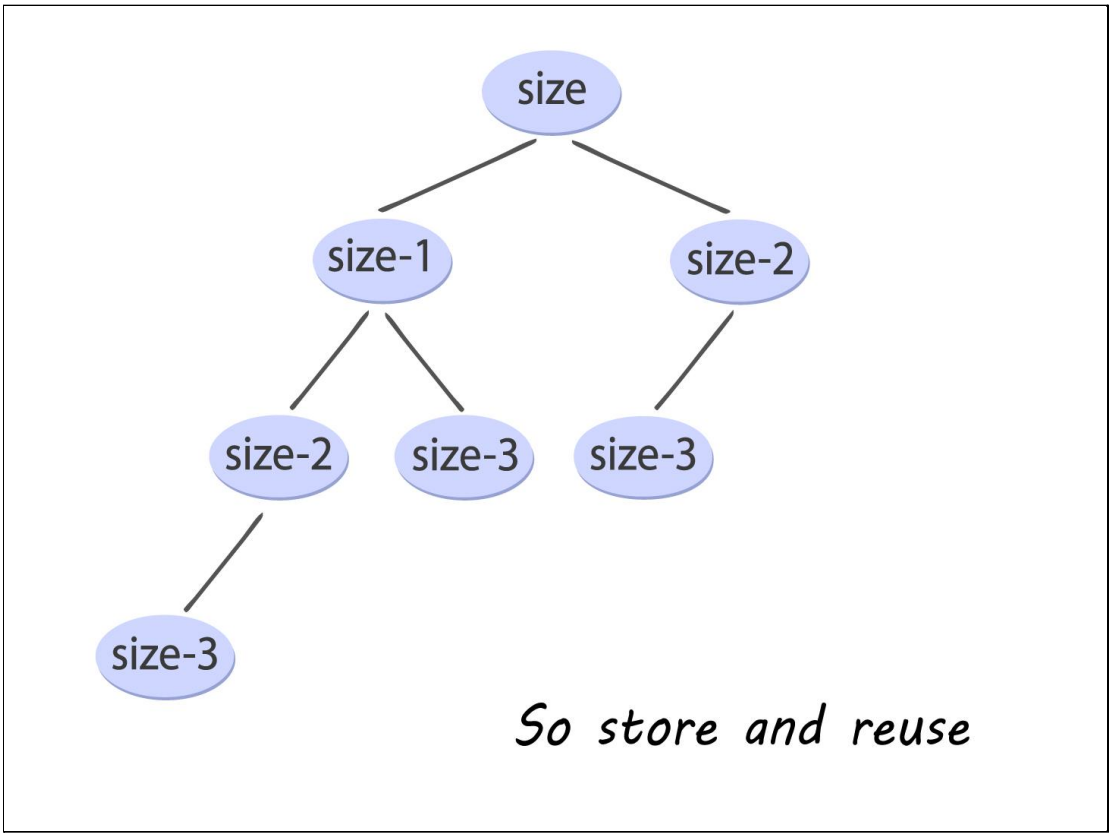
We will always make the first recursive call , but for the second one we check whether the two consecutive digits form a number that is less than or equal to 26.

Now, let's take a look at the code for better understanding:

```
int num_codes(int* n, int size) {
     if (size == 1) {
         return 1;
     }
     if (size == 0) {
         return 1;
     }
     int output = num_codes(n, size - 1);
     if (output[size - 2] * 10 + output[size - 1] <= 26) {
         output += num_codes(n, size -2);
     }
     return output;
}
```

Note: In the above code, we have called recursion from the end rather than start.

Now, we have to check if we can optimize the code that we have written. It can be done using memoization. But, for memoization to apply, we need to check if there are any overlapping sub-problems so that we can store the previous values to obtain the new ones.



Here, if we blindly make two recursive calls at each step, then the time complexity will approximately be $O(2^n)$.

From the above, it is clearly visible that there are repeating sub-problems. Hence, this problem can be optimized using memoization.

```
int num_codes2(int *n, int size, int *arr){

    if(size == 1){
        return 1;
    }
    if(size == 0){
        return 1;
    }
    if(arr[size] > 0){
        return arr[size];
    }
    int output = num_codes(n, size - 1, arr);
    if (output[size - 2] * 10 + output[size - 1] <= 26) {
        output += num_codes(n, size -2, arr);
    }
    arr[size] = output;
    return output;
}
```

Iterative Solution:

We make an array and the i^{th} index of that array will store the number of possible codes if we just consider first i digits.

Code:

```
int num_codes_i(int * input, int size) {
    int* output = new int[size + 1];
    output[0] = 1;
    output[1] = 1;

    for (int i = 2; i <= size; i++) {
        output[i] = output[i - 1];
        if (input[i-2] *10 + input[i - 1] <= 26) {
            output[i] += output[i - 2];
        }
    }
    int ans = output[size];
```

```
    delete [] output;
    return ans;
}
```

Longest Increasing Subsequence

Problem Statement: You are given an array of positive integers as input. Write a code to return the length of the largest such subsequence in which the values are arranged first in strictly ascending order and then in strictly descending order.

Such a subsequence is known as bitonic subsequence. A purely increasing or purely decreasing subsequence will also be considered as a bitonic sequence with the other part empty.

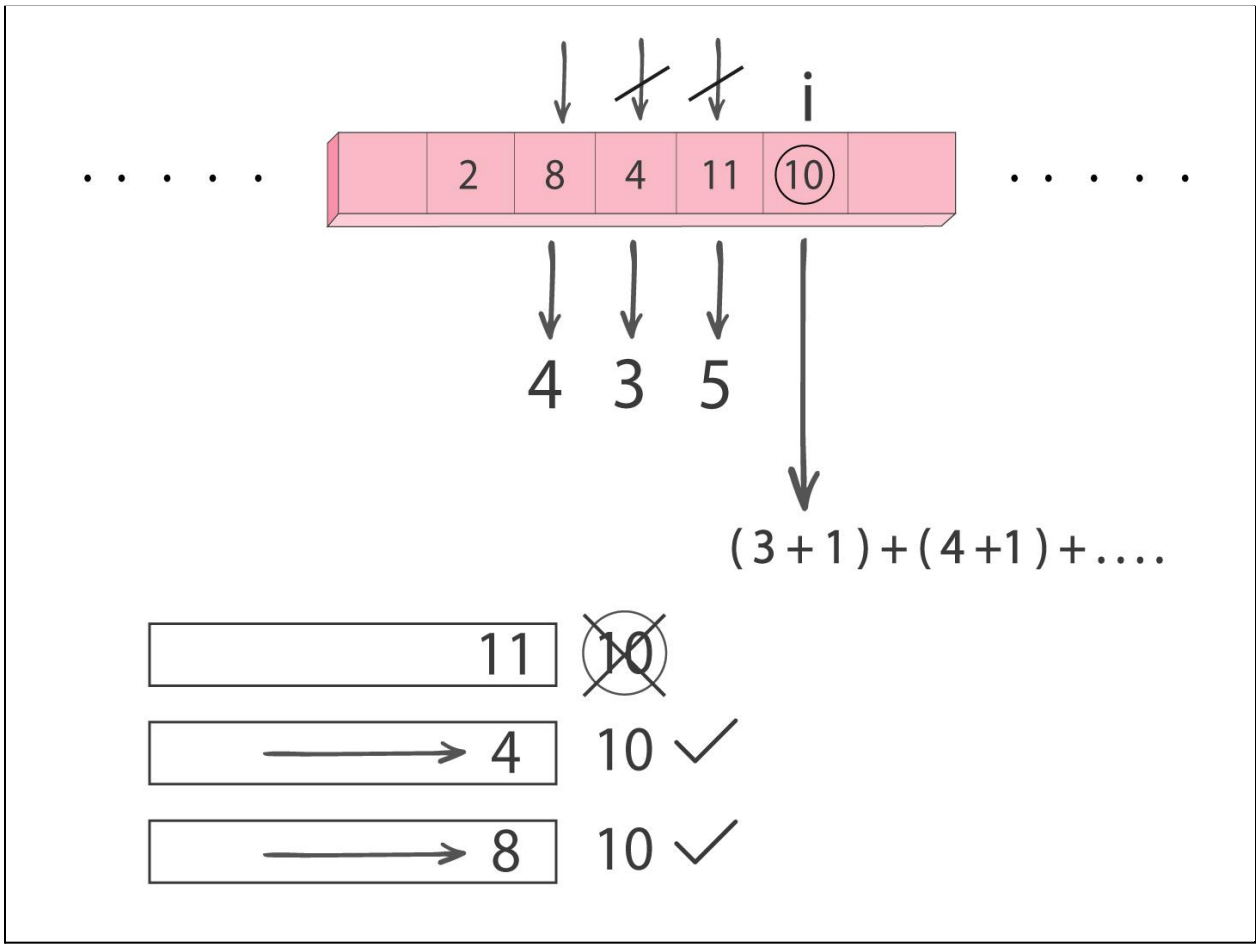
Note that the elements in the bitonic subsequence need not be consecutive in the given array but the order should remain the same.

Explanation:

The brute force approach to this problem is to find all the subsequences and then give the maximum length. The time complexity of this will be exponential as there can be 2^n subsequences.

Dynamic Programming Approach is to divide this problem into a smaller problem by finding the longest subsequence finishing at each index.

Let's try to understand this by an example:



Time Complexity of this solution is **O(n^2)**.

Code:

```
#include <iostream>
using namespace std;

int lis(int* input, int n) {
    int* output = new int[n];
    output[0] = 1;
    for (int i = 1; i < n; i++) {
        output[i] = 1;
        for (int j = i - 1; j >= 0; j--) {
            if (input[j] > input[i]) {
                continue;
```

```
            }
            int possibleAns = output[j] + 1;
            if (possibleAns > output[i]) {
                output[i] = possibleAns;
            }
        }
    }
    int best = 0;
    for (int i = 0; i < n; i++) {
        if (best < output[i]) {
            best = output[i];
        }
    }
    delete [] output;
    return best;
}

int main() {
    int n;
    cin >> n;
    int * input = new int[n];
    for (int i = 0; i < n; i++) {
        cin >> input[i];
    }
    int ans = lis(input, n);
    cout << ans << endl;
    delete [] input;
}
```

Staircase Problem

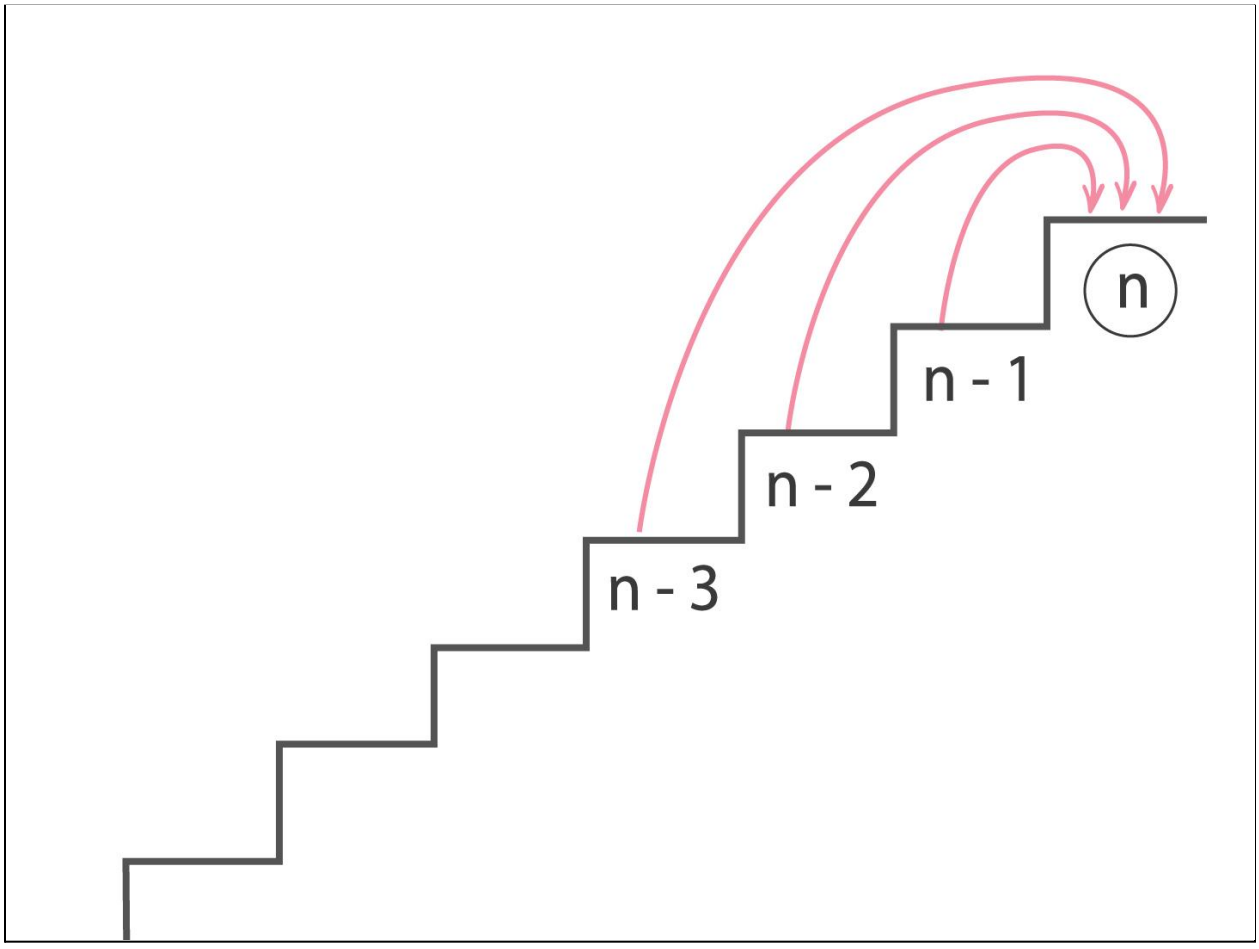
Problem Statement: A child is running up a staircase with n steps and can hop either 1 step, 2 steps or 3 steps at a time. Implement a method to count how many possible ways the child can run up to the stairs. You need to return the number of possible ways.

Time complexity of your code should be O(n).

Since the answer can be pretty large print the answer % mod(10^9 +7)

Explanation:

There are **n** steps and reaching the last step could have been achieved from **(n-1)th**, **(n-2)th** or **(n-3)th** step because the child can only take 3 or less steps at a time.



So, basically it is like a fibonacci series as **S(n) = S(n-1) + S(n-2) + S(n-3)**, and implementing the solution using recursion will not be an ideal approach. So we will use the Dynamic Programming approach just like we did in the Fibonacci series.

Base Cases:

1. There is only one way to take no step.
2. There is only one way to take one step.
3. There are two ways to take 2 steps, i.e., {1, 1} and {2}.

Coin Change Problem

Problem Statement: You are given an infinite supply of coins of each denomination D = {D0, D1, D2, D3, ... Dn-1}. You need to figure out the total number of ways W, in which you can make a change for Value V using coins of denominations D.

Note : Return 0, if change isn't possible.

W can be pretty large so output the answer % mod(10^9 + 7)

Explanation:

This seems to be a very simple problem in which, like the previous one, **F(n) = F(n-D_0) + F(n-D1) + F(n-D2) and so on..** But there may be similar cases in this problem, for example, if we take **n = 4**, and we have **1, 2, 5 Rs coins** then **{1, 1, 2} and {1, 2, 1}** are exactly the same because in coins the order doesn't matter.

n, {D_0, D_1, D_2, D_3, ... D_4}

The problem can be reduced in two ways:

1. D_0 is included: **n-D_0, {D_0, D_1, D_2, D_3, ... D_4}**
2. D_0 is not included: **n, {D_1, D_2, D_3, ... D_4}**

Base Cases:

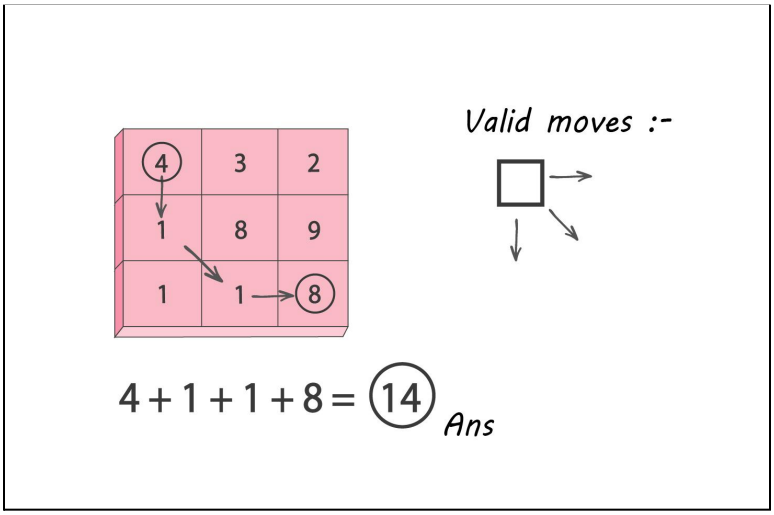
1. If n is equal to 0, return 1
2. If length of array is equal to 0, return 0

Minimum Cost

Problem Statement: We are given a 2D array with [i][j]th cell holding the cost to travel through that cell. Our goal is to find the minimum cost to reach the end point from the start.

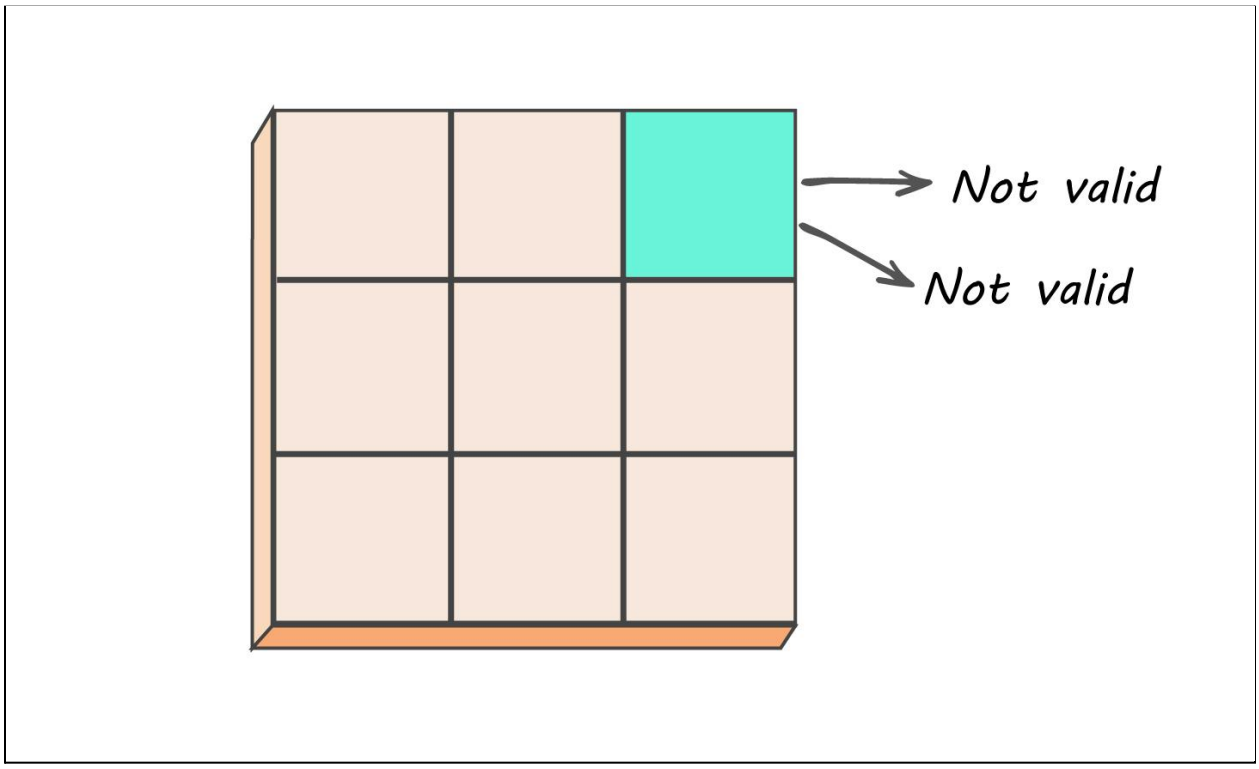
Explanation:

Naive Approach: The valid directions in which we can move are right, down and down-right.



Now, when we are at the edge of the matrix, one or two of our paths will also become invalid. So to choose, which path to take at that time, we have two options:

1. Either to check every time whether we are at an edge.
2. For an invalid move at the edge, we take the cost to travel to be infinite, so it automatically doesn't get chosen.



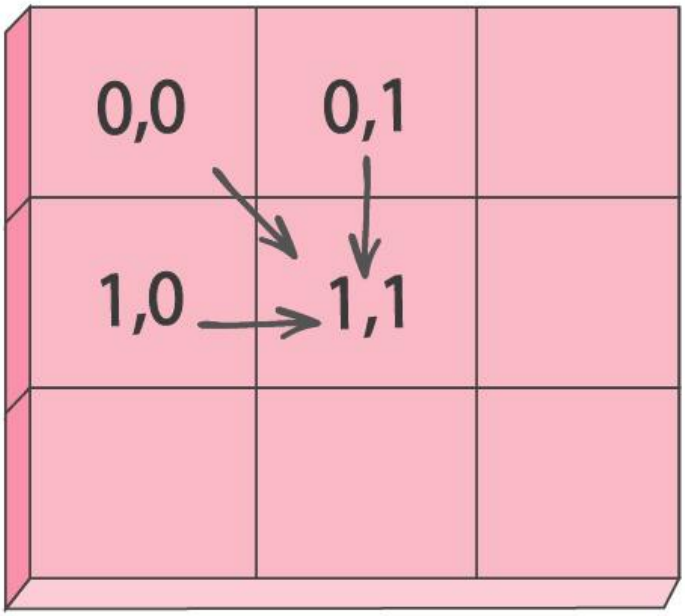
Code:

```
int min_cost(int** input, int si, int sj, int ei, int ej) {
    if (si == ei && sj == ej) {
        return input[ei][ej];
    }
    if (si > ei || sj > ej) {
        return INT_MAX;
    }
```

```
    int option1 = min_cost(input, si + 1, sj, ei, ej);
    int option2 = min_cost(input, si + 1, sj + 1, ei, ej);
    int option3 = min_cost(input, si, sj + 1, ei, ej);
    return input[si][sj] + min(option1, min(option2, option3));
}

int main() {
    int ** input = new int*[3];
    input[0] = new int[3];
    input[1] = new int[3];
    input[2] = new int[3];
    input[0][0] = 4;
    input[0][1] = 3;
    input[0][2] = 2;
    input[1][0] = 1;
    input[1][1] = 8;
    input[1][2] = 3;
    input[2][0] = 1;
    input[2][1] = 1;
    input[2][2] = 8;

    cout << min_cost(input, 0,0,2,2) << endl;
    cout << min_cost2(input,3,3) << endl;
    delete [] input[0];
    delete [] input[1];
    delete [] input[2];
    delete [] input;
}
```

Time Complexity: From every cell we are making three recursive calls, so if the height of the tree is m, then this solution has time complexity **$O(3^m)$** which is exponential.

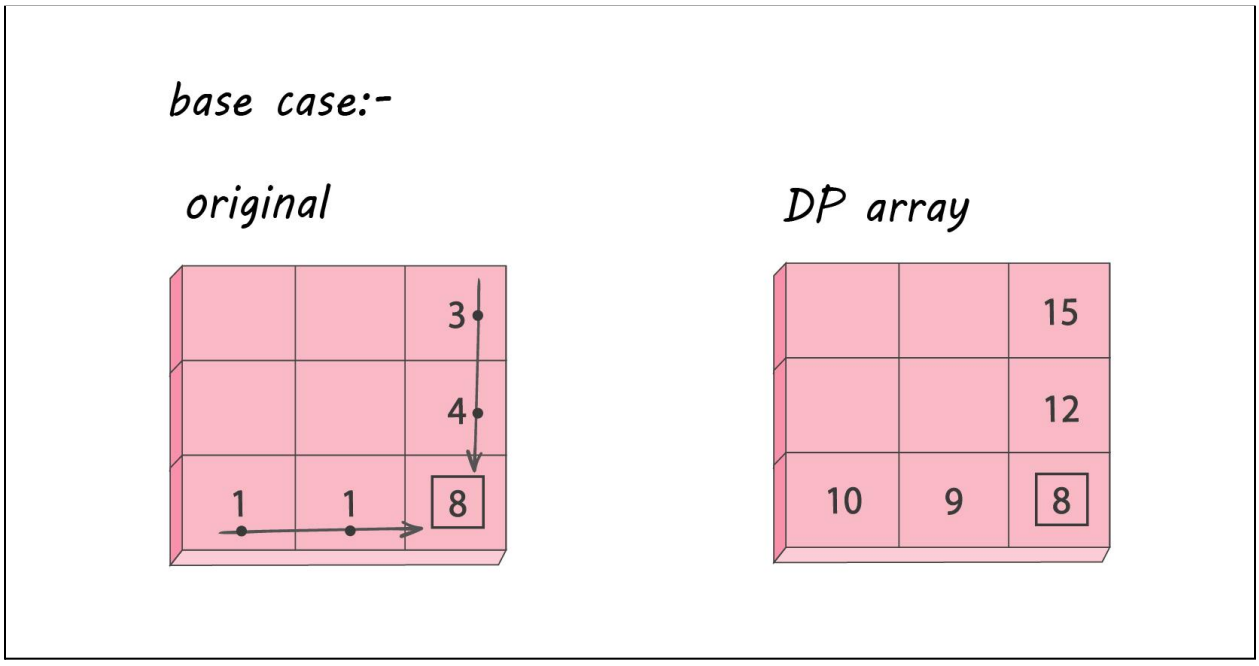
But this can be reduced using memoization, since lots of repetitions are happening in this solution.



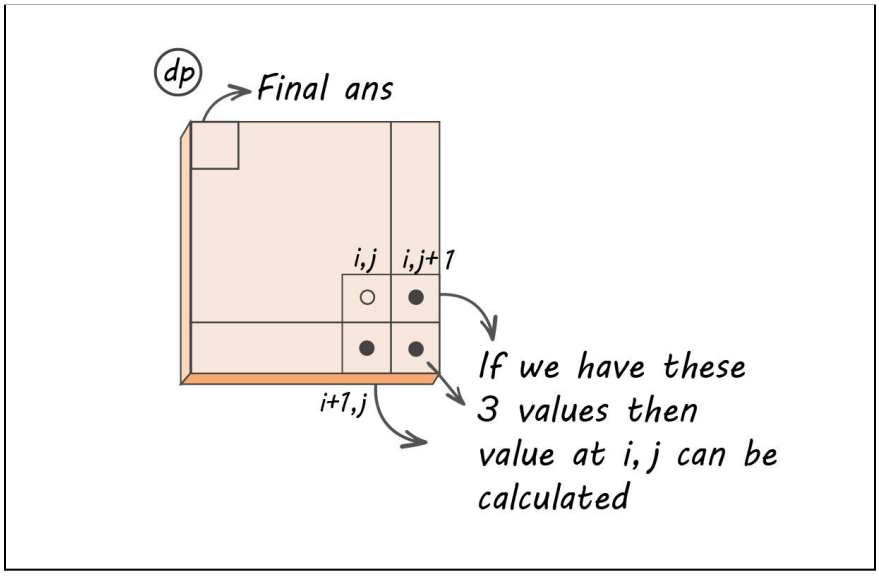
To implement this, we simply pass another 2D array to check whether some [i][j] position has already been checked or not. Try to implement this yourself.

Iterative Solution:

We make a dp array that at each index, stores the cost to go to the last index from the current index.



Now, to fill the value at [i][j], we need three values at **[i][j+1], [i+1][j], [i+1][j+1].**



Code:

```
int min_cost2(int** input, int m, int n) {
    int ** dp = new int*[m];
    for (int i = 0; i < m; i++) {
        dp[i] = new int[n];
    }
    dp[m - 1][n-1] = input[m-1][n-1];
    for (int i = m - 2; i >= 0; i--) {
        dp[i][n -1] = dp[i + 1][n-1] + input[i][n-1];
    }

    for (int j = n - 2; j >=0; j--) {
        dp[m -1][j] = dp[m - 1][j + 1] + input[m-1][j];
    }

    for (int i = m - 2; i >=0; i--) {
        for (int j = n - 2; j >=0 ; j--) {
            dp[i][j] = input[i][j] + min(dp[i+1][j], min(dp[i+1][j+1],
dp[i][j + 1]));
        }
    }
    return dp[0][0];
}
```