Dynamic Programming - 1

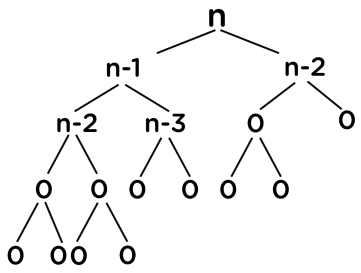
Introduction

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:

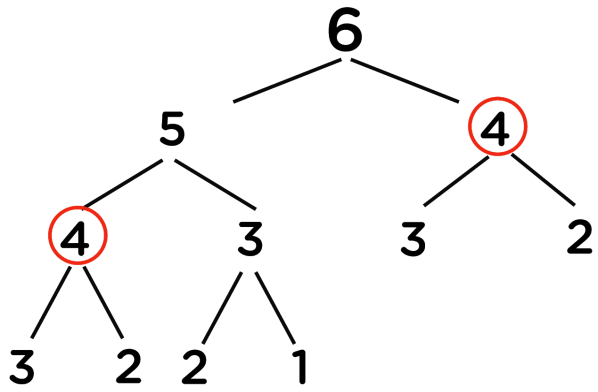


At every recursive call, we are doing constant work(k)(addition of previous outputs to obtain the current one). At every level, we are doing $2^n K$ work (where n = 0, 1, 2, ...). Since reaching 1 from n will take n calls, therefore, at the last level, we are doing $2^{n-1}k$ work. Total work can be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

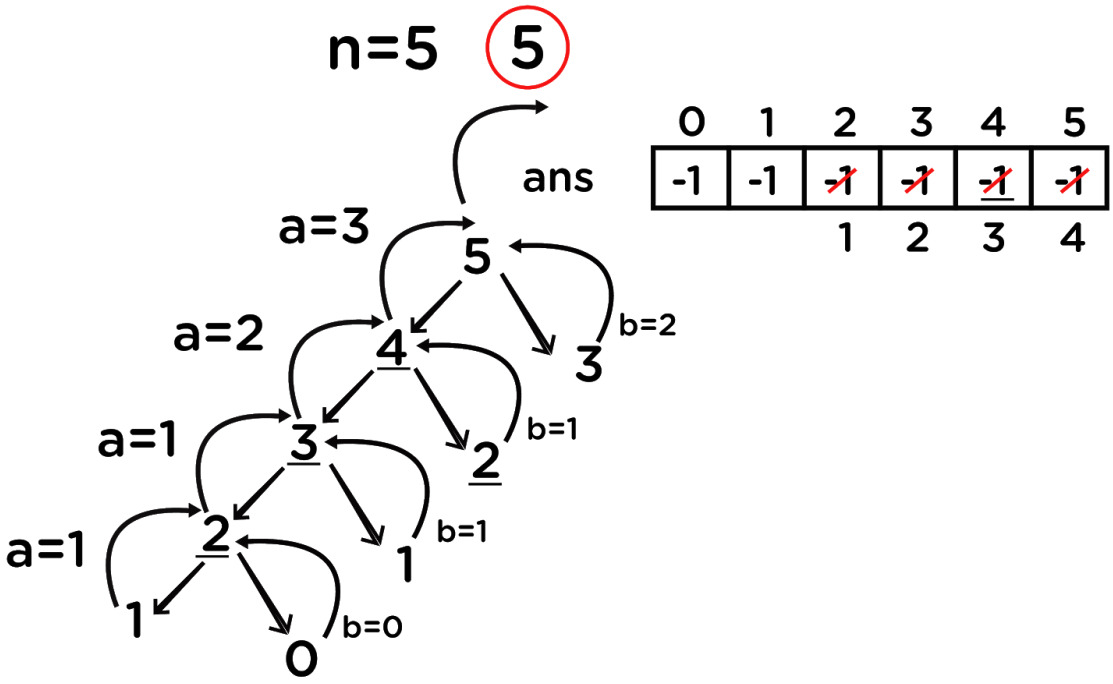
Now in this process of memoization, considering the above Fibonacci numbers example, it can be observed that the total number of unique calls will be at most (n+1) only.

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 calculating 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];
        }

        return ans[n];              // 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)

Min steps to 1

Let's now solve another problem named **min steps to 1**.

Problem statement: Given a positive integer n, find the minimum number of steps s, that takes n to 1. You can perform any one of the following three steps:

1. Subtract 1 from it. (n = n-1)
2. If its divisible by 2, divide by 2. (if n%2 == 0, then n= n/2) ,
3. If its divisible by 3, divide by 3. (if n%3 == 0, then n = n / 3).

Example 1: For n = 4:

STEP-1: n = 4/2 = 2

STEP-2: n = 2/2 = 1

Hence, the answer is 2.

Example 2: For n = 7:

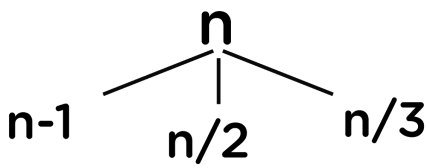
STEP-1: n = 7 - 1 = 6

STEP-2: n = 6/3 = 2

STEP-3: n = 2/2 = 1

Hence, the answer is 3.

Approach: We are only allowed to perform the above three mentioned ways to reduce any number to 1.



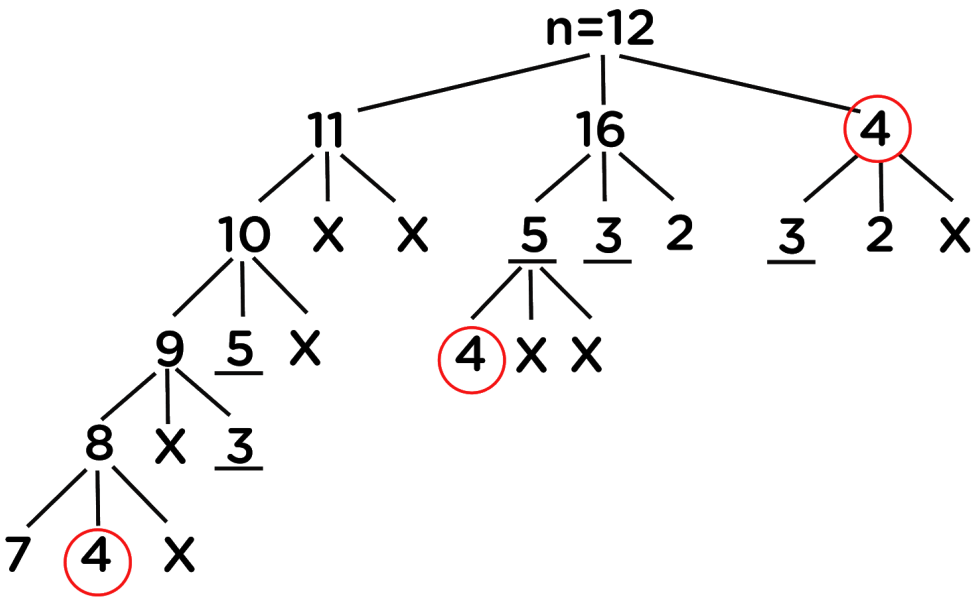
Let's start thinking about the brute-force approach first, i.e., recursion.

We will make a recursive call to each of the three steps keeping in mind that for dividing by 2, the number should be divisible by 2 and similarly for 3 as given in the question statement. After that take minimum value out of the three obtained and simply add 1 to the answer for the current step itself. Thinking about the base case, we can see that that on reaching 1, simply we have to return 0 as it is our destination value. Let's now look at the code:

```
int minSteps(int n) {
        if(n <= 1) {                                    // Base case
                return 0;
        }

        int x = minSteps(n - 1);                        // Recursive call - 1

        int y = INT_MAX, z = INT_MAX;                   // These values are intialized to +infinity
                                                        // so as to check if n is not divisible by 2 or 3
        if(n % 2 == 0) {
                y = minSteps(n/2);                      // Recursive call - 2
        }

        if(n % 3 == 0) {
                z = minSteps(n/3);                      // Recursive call - 3
        }

        // Calculate final output
        int ans = min(x, min(y, z)) + 1;
        return ans;
}
```

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. To check this let's dry run the problem for n = 12:

(Here X represents that the calls are not feasible as the number is not divisible by either of 2 or 3)



Here, if we blindly make three recursive calls at each step, then the time complexity will approximately be $O(3^n)$.

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

Now, we need to figure out the number of unique calls, i.e., how many answers we are required to save. It is clear that we need at most n+1 responses to be saved, starting from n = 0, and the final answer will be present at index n.

The code will be nearly the same as the recursive approach; just we will not be making recursive calls for already stored outputs. Follow the code and comments below:

```
int minStepsHelper(int n, int *ans) {
        if(n <= 1) {                                    // Base case
                return 0;
        }

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

        int x = minStepsHelper(n - 1, ans);   // Calculate output - 1

        int y = INT_MAX, z = INT_MAX;
        if(n % 2 == 0) {
                y = minStepsHelper(n/2, ans);       // Calculate output - 2
        }

        if(n % 3 == 0) {
                z = minStepsHelper(n/3, ans);     // Calculate output - 3
        }

        int output = min(x, min(y, z)) + 1;
        ans[n] = output;                                // Save output for future use

        return output;

}

int misSteps_2(int n) {
        int *ans = new int[n+1];

        for(int i = 0; i <= n; i++) {   // initialized to -1 denoting answer is
                ans[i] = -1;            // unknown at current stage for the particular index
        }

        return minStepsHelper(n, ans);
}
```

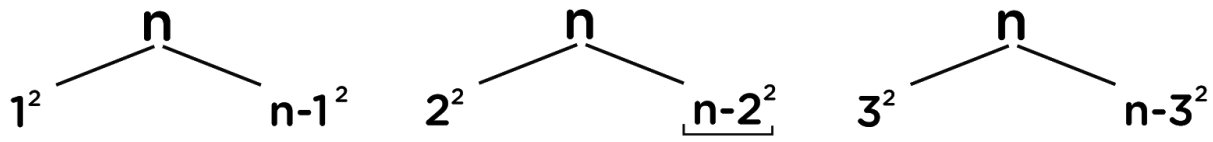
Time complexity has been reduced significantly to O(n) as there are only (n+1) unique iterations. Now, try to code the DP approach by yourself, and for the code, refer to the solution tab of the same.

Minimum Count

Problem statement: Given an integer N, find and return the count of minimum numbers, the sum of whose squares is equal to N.
That is, if N is 4, then we can represent it as : {1^2 + 1^2 + 1^2 + 1^2} and {2^2}. The output will be 1, as 1 is the minimum count of numbers required. (x^y represents x raised to the power y.)
Example: For n = 12, we have the following ways:

- 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1
- 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 2^2
- 1^1 + 1^1 + 1^1 + 1^1 + 2^2 + 2^2
- 2^2 + 2^2 + 2^2

Hence, the minimum count is obtained from the 4-th option. Therefore, the answer is equal to 3.
Approach: First-of-all, we need to think about breaking the problems into two parts, one of which will be handled by recursion and the other one will be handled by us(smaller sub-problem). We can break the problem as follows:



And so on...

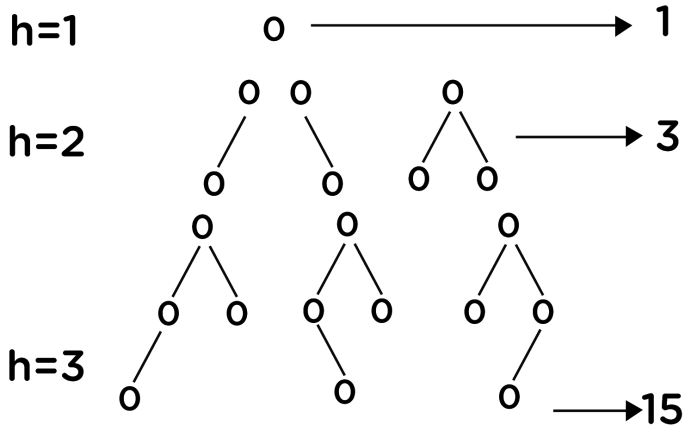
- In the above figure, it is clear that in the left subtree we are making ourselves try over a variety of values that can be included as a part of our solution.
- This left subtree's calculation will be done by us and the right subtree will be done by recursion.

- Hence, we will just handle the i^2 part and ($n-i^2$) will be handled by recursion.
- As by now, we have got ourselves an idea of solving this problem, the only thinking left is the loop's range on which we will be iterating, i.e., the values of **i** for which we will be deciding to consider while solving or not.
- As, the maximum value upto which i can be pushed to in order to reach n is √n as (√n*√n = n). Hence, we will be iterating over the range (1 to √n) and do consider each possible way by sending ($n-i^2$) over the recursion.
- This way we will get different subsequences and as per the question, we will simply return the minimum out of it.

This problem is left for you to try out using all the three approaches and for code, refer to the solution tab of the same.

No. of balanced BTs

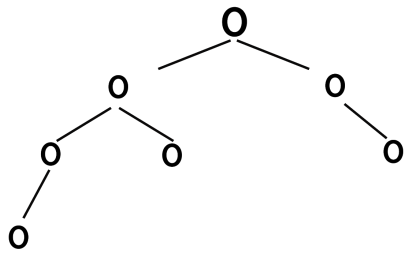
Problem Statement: Given an integer h, find the possible number of balanced binary trees of height h. You just need to return the count of possible binary trees which are balanced. This number can be huge, so return output modulo $10^9 + 7$.
For Example: In the figure below, the left side represents the height h, and the right side represents the possible binary trees along with the count.



Here for h = 1, the answer is 1. For h = 2, the answer is 3. For h = 3, the answer is 15.

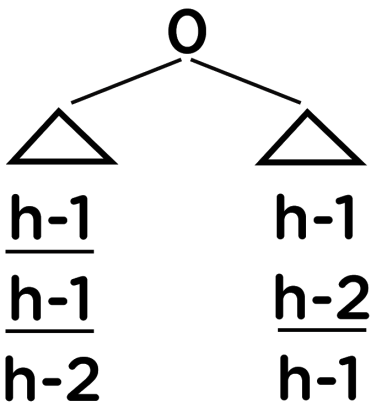
Approach: Suppose we have h = 3, so at level 3 there are four nodes and each node has two options, either it can be included or excluded from the binary tree, hence in total, we have 2^4 = 16 possibilities. Here, we need to exclude the possibility of the case when none out of 4 is present. Hence, the remaining options are 16-1 = 15. We can think that this approach could be efficient as we just need to know the number of nodes at the last level, and then we can simply apply the above formula.

Now, consider for h = 4, where the last level has got eight nodes, so according to the above approach, the answer could be 2^8 - 1 = 255 ways, but the solution for h = 4 is 315. Let's look at the cases which we missed. One of the examples could be:



Till now, we were only working over the last level, but in the above example, if we remove the nodes from upper levels, then also the binary tree could remain balanced.

Let's now think about implementing it using recursion on trees. If the height of the complete binary tree is h, that means the height of its left and right subtrees individually is at most h-1. So if the height of the left subtree is h-1, then the height of the right subtree could be any among {h-1, h-2} and vise versa.



Initially, we were given the problem of finding the output for height h. Now we have reduced the same to tell the output of height h-1 and h-2. Finally, we just need to figure out these counts, add them, and return.

Lets now look at the code below:

```
int balancedBTs(int h) {
        if(h <= 1) {                                    // Base case
                return 1;
        }

        int mod = (int) (pow(10, 9)) + 7;
        int x = balancedBTs(h - 1);                     // Answer for h-1
        int y = balancedBTs(h - 2);                     // Answer for h-2

/* Since, we need to find the total number of combinations, so will multiply the left
height's output and the right height's output as they are independent of each other
(Using law of multiplication in combinations)

    Possible Cases:
    • Both h-1          =       x*x
    • h-1 and h-2       =       x*y
    • h-2 and h-1       =       y*x

    Now, we will add all these together.
        */
```

```
        int temp1 = (int)(((long)(x)*x) % mod);
        int temp2 = (int)((2* (long)(x) * y) % mod);
        int ans = (temp1 + temp2) % mod;

        return ans;
}
```

Time Complexity: If we observe this function, then we can find it very similar to the pattern of the Fibonacci number. Hence, the time complexity is $O(2^h)$.

Now, try to reduce the time complexity of the code using memoization and DP by yourselves and for solution refer to the solution tab of the problem.