Climbing Stairs

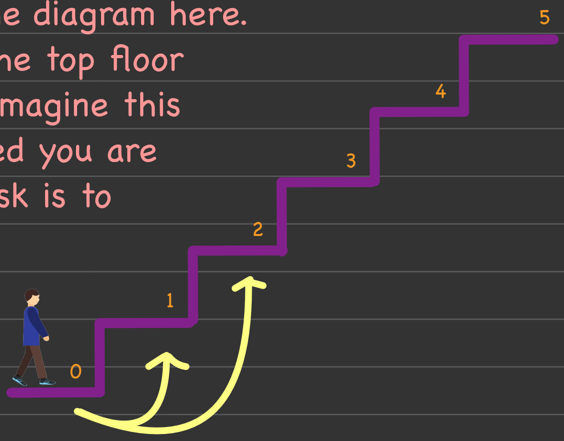
What exactly is the problem statement for climbing stairs?

You are climbing a staircase with 0 to 'n' floors. 'n' is the top floor and the value of 'n' is specified by the user. Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Understanding the problem visually

Imagine you have the following stairs case, where n represents the number of floors and n = 5

Visually we can imagine our problem as shown in the diagram here. We have a total of n floors, in this case n = 5, so the top floor is 5 and we are starting from floor 0, we can also imagine this as ground floor! As the problem statement mentioned you are allowed to climb 1 or 2 steps at every floor. Our task is to find total number of paths from 0th floor to nth floor.



For example the following are some valid paths from oth to nth floor:

1, 1, 1, 1, 1
1, 1, 1, 2
1, 1, 2, 1
1, 2, 1, 1

and so on...

Here as you can see I have given only 4 different paths to reach 5th floor from 0th floor. Our task is to find all the possible ways to reach nth floor and return that value.

NOTE: Whether you climb from 0th to nth floor or nth to 0th floor the number of ways will be same! In other words whether I start from bottom and climb up or start from top and climb down to reach 0th floor the total number of paths will be same. So lets assume for now that I am on the top floor climbing down to 0th floor and I want to find all the possible ways from nth floor to 0th floor.

Now that we have understood the problem statement, lets try to relate this to a real life scenario so that we can see how to go about solving this question.

Let's assume that we are trying to fly from Bangalore to London. Here our source is Bangalore and destination is London.

Lets say to go from source to destination these are the following ways

Way 1: Bangalore to London via Dubai with British Airways (BA)

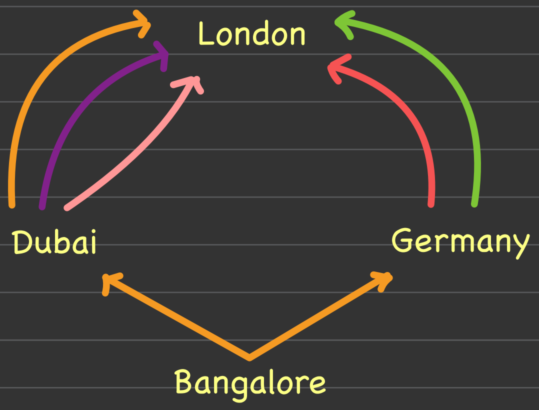
Way 2: Bangalore to London via Dubai with Lufthansa (L)

Way 3: Bangalore to London via Dubai with Singapore Airways (SA)

Way 4: Bangalore to London via Germany with Delta Air (DA)

Way 5: Bangalore to London via Germany with Turkish Air (TA)

Now that we know all the ways from Bangalore to to London lets try to draw a simple diagram for this.



So just by looking at this diagram we can see that the total number of ways from Bangalore to London is all the ways via Dubai + all the ways via Germany.

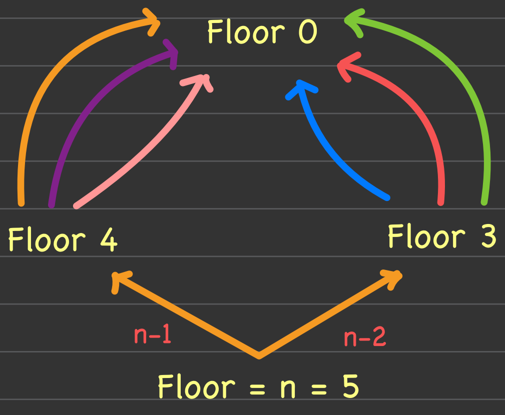
Let all the ways via Dubai be ‘x’ and similarly let all the ways via Germany be ‘y’

Total no of ways = x + y

NOTE: even if the number of ways via Dubai or via Germany changed the, the total would still be x + y right? Correct!

Similarly lets try to make the same diagram for our climbing the staircase problem!

- [ ] Source = nth floor (from our previous example n=5)
- [ ] Destination = 0th floor
- [ ] We can either climb 1 or 2 steps so (n-1) or (n-2) Since we are going down!



Now from our diagram to get the total number of paths from nth floor to 0th floor we will simply have to find all the paths from (n-1) to 0th floor and all the paths from (n-2) to 0th floor and sum their results. This will give the total number of paths.

That would simply be (n-1) + (n-2)

NOTE: In our diagram we have 6 paths, this is just for demonstration purposes so that we understand how to implements our recursive solution. The total number of paths when n=5 is actually 8!

Now that we have a pretty clear idea of how to approach this problem recursively, lets write the implementation for it.

Implementation - Recursive approach

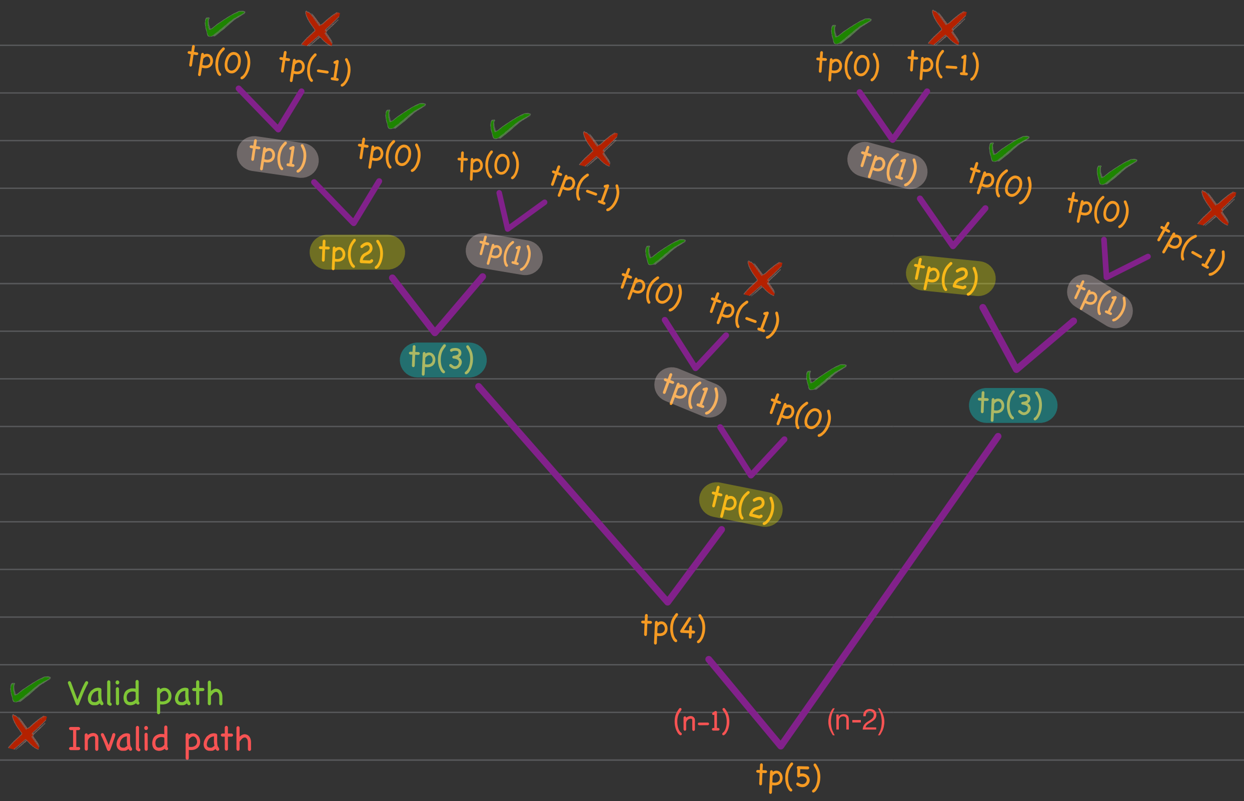
```
public static int totalPaths(int n) {
    if (n == 0)
        return 1;
    else if (n < 0)
        return 0;

    int nm1 = totalPaths(n-1);
    int nm2 = totalPaths(n-2);
    int cp = nm1 + nm2;

    return cp;
}
```

Now, lets trace out our code step-by-step and draw out our tree diagram for all the recursive calls to find totalPaths(5)

<u>Tree Diagram for totalPaths(5)</u>



From the tree diagram we can make out how many total paths there are. When we go up the tree eventually we will either hit 0th floor or below 0th floor. Obviously if we below 0th floor those are invalid paths, and the ones which reach 0th floor they are valid paths. If we count paths leading to 0th floor we will see we get 8 paths, where (n-1) gives 5 paths and (n-2) gives 3 paths, total = 8, and that is the expected answer!

But here you can see,
We are calculating tp(1) 5 times even though we know that tp(2) will yield in same result all 5 times.

We are calculating tp(2) 3 times even though we know that tp(2) will yield in same result all 3 times.

We are calculating tp(3) 2 times even though we know that tp(3) will yield in same result all 2 times.

There is so much of recalculations happening here which makes this approach such a bad one.

Whenever we encounter such repetition in our tree diagram we should make us of dynamic programming. Wouldn't it be much easier to just reuse the result of the already calculated floors. Why go through recalculating it again and again just for the sake of it. In Dynamic programming we have something called memoization. The idea behind memoization is to store the results and re-use the stored results whenever needed again instead of recalculating them again!

Solution using dynamic programming - memoization

WHAT is our task?
Our task is to find the nth fibonacci number, where n is given by the user.

HOW can we achieve our task?

- The method signature will also take in an array of ints called dp to store the number of paths for each floor. Since we want to store the results of each floor, our dp array see should be floor+1 or n+1, where n represent floors.
- We check if our dp[n] != 0
  -> If yields to true that means result is already stored for tp(n), so we can just return the result by saying return dp[n]
  -> If it yields to false that means the result is not stored for tp(n) and we need to calculate it
- Find tp(n-1) for climbing 1 step down
- Find tp(n-2) for climbing 2 steps down
- Store the sum of tp(n-1) and tp(n-2) int dp[n]
- Return the sum of tp(n-1) and tp(n-2)

Note: Since we know that a vlid path is when n reaches 0 then we simply return 1 meaning we have found a valid path, or else if n goes below 0 that means we have gone down to basement level which is not a valid path so we return 0, as we do not want to count this path.

Implementation - Memoization approach

```
public static int totalPaths(int n, int[] dp) {
    if (n == 0)
        return 1;     // found a valid path so we count it
    else if (n < 0)
        return 0;     // did not find a valid path so we do not count it

    // checking if result of tp(n) is already stored in dp array, if so return result
    if (dp[n] != 0)
        return dp[n];

    // If result of tp(n) is not stored in dp array then calculate it
    int nm1 = totalPaths(n-1, dp);
    int nm2 = totalPaths(n-2, dp);
    int cp = nm1 + nm2;

    dp[n] = cp;          // Store the result of tp(n) into dp array, for later re-use
    return cp;           // return cp
}
```

Now, lets trace out our code step-by-step and draw out our tree diagram along the way for all the recursive calls to find total no. of paths if n=5

In this implementation the method takes 2 parameters:

- The nth floor for which we have to find all possible paths
- An int array of size n+1 where initially all values would be 0

So if we wanted to find the total number of valid paths from nth to 0th floor, we can safely assume the following:

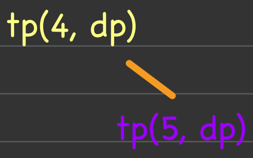
n = 5

dp →

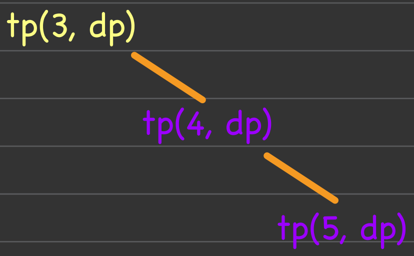
0	0	0	0	0	0
0	1	2	3	4	5

This means the following call was made totalPaths(5, new int[n+1]), where n=5 obviously. That would give us the above n value and dp array. Our tree diagram at this point would just contain the root tp(5, dp)

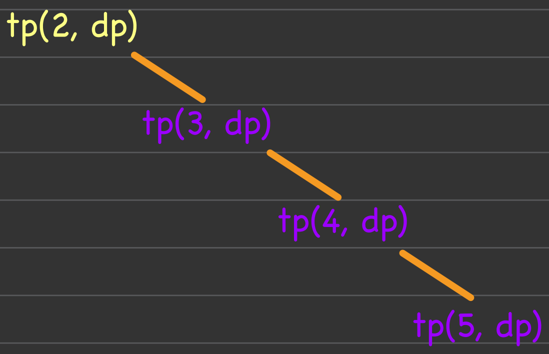
Now, we will check if (n == 0), this is false as 5 is NOT equal to 0.
Then we are checking if (n < 0), this is false as 5 is NOT less than 0.
Next we will check if (dp[n] != 0), our n = 5 so we are checking if dp[5] != 0, this condition is false because the values stored in at index 5 of dp array is indeed 0. This means we have not previously calculated and stored the result for totalPaths(5). Due to this we need to go ahead and calculate it. A call will be made to totalPaths(4, dp), so now out tree diagram looks like this:



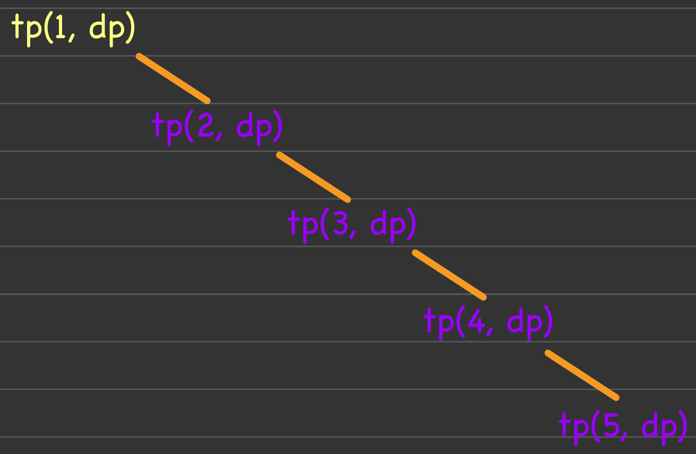
Now, we will check if (n == 0), this is false as 4 is NOT equal to 0.
Then we are checking if (n < 0), this is false as 4 is NOT less than 0.
Next we will check if (dp[n] != 0), our n = 4 so we are checking if dp[4] != 0, this condition is false because the values stored in at index 4 of dp array is indeed 0. This means we have not previously calculated and stored the result for totalPaths(4). Due to this we need to go ahead and calculate it. A call will be made to totalPaths(3, dp), so now out tree diagram looks like this:



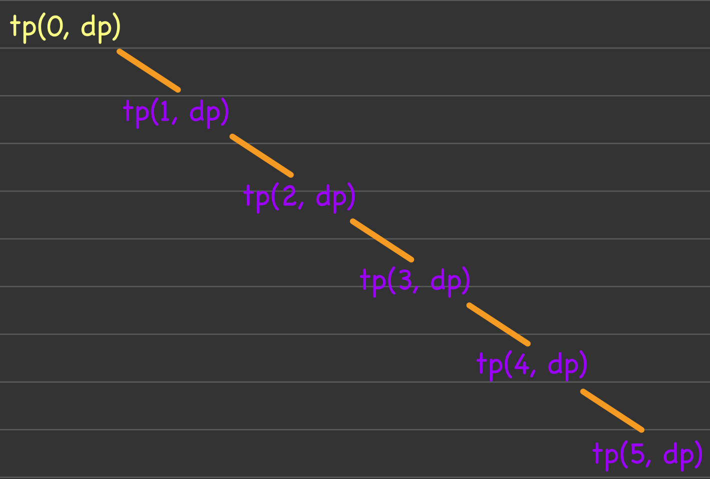
Now, we will check if (n == 0), this is false as 3 is NOT equal to 0.
Then we are checking if (n < 0), this is false as 3 is NOT less than 0.
Next we will check if (dp[n] != 0), our n = 3 so we are checking if dp[3] != 0, this condition is false because the values stored in at index 3 of dp array is indeed 0. This means we have not previously calculated and stored the result for totalPaths(3). Due to this we need to go ahead and calculate it. A call will be made to totalPaths(2, dp), so now out tree diagram looks like this:



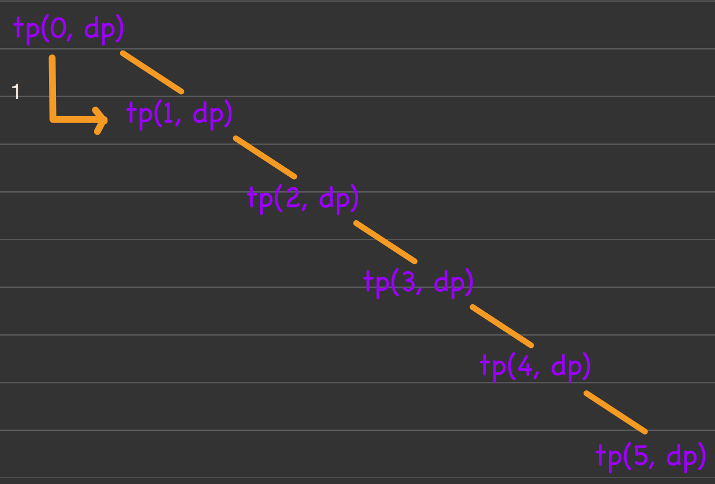
Now, we will check if (n == 0), this is false as 2 is NOT equal to 0.
Then we are checking if (n < 0), this is false as 2 is NOT less than 0.
Next we will check if (dp[n] != 0), our n = 2 so we are checking if dp[2] != 0, this condition is false because the values stored in at index 2 of dp array is indeed 0. This means we have not previously calculated and stored the result for totalPaths(2). Due to this we need to go ahead and calculate it. A call will be made to totalPaths(1, dp), so now out tree diagram looks like this:



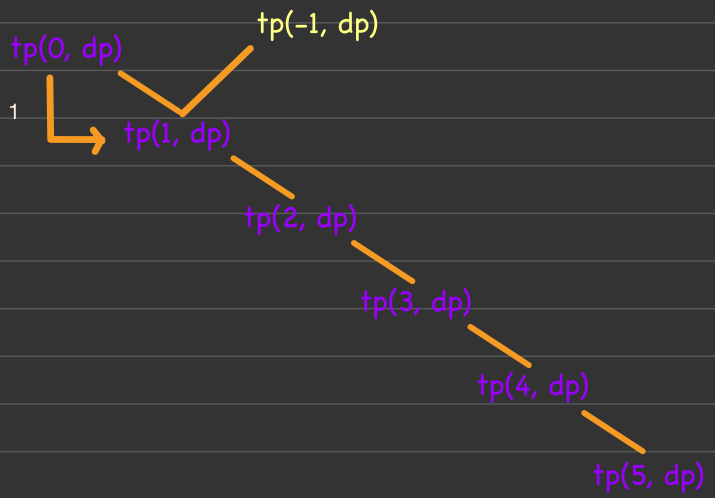
Now, we will check if (n == 0), this is false as 1 is NOT equal to 0.
Then we are checking if (n < 0), this is false as 1 is NOT less than 0.
Next we will check if (dp[n] != 0), our n = 1 so we are checking if dp[1] != 0, this condition is false because the values stored in at index 1 of dp array is indeed 0. This means we have not previously calculated and stored the result for totalPaths(1). Due to this we need to go ahead and calculate it. A call will be made to totalPaths(0, dp), so now out tree diagram looks like this:



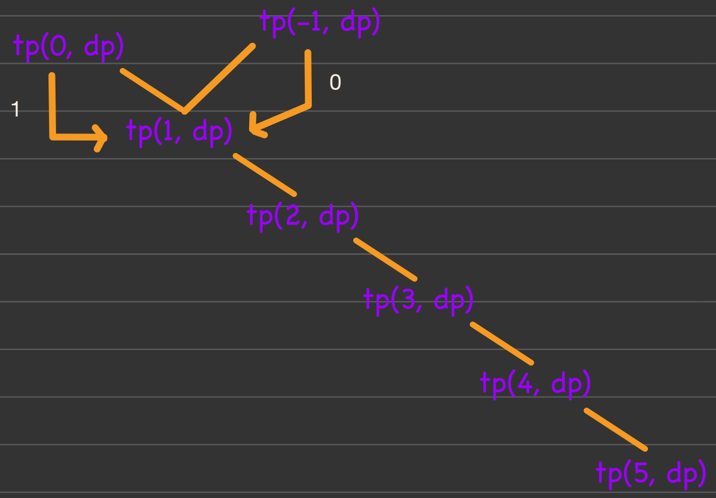
Now, we will check if (n == 0), this is true as 0 is equal to 0. This means we have found a path from nth floor to 0th floor and we can clearly see that path in our tree diagram also. So we will return 1. And now our tree diagram would look like this:



Now, the control goes back to totalPaths(1, dp) and will continue from where we left off. totalPaths(1, dp) will make its second recursive call to find all the paths from (n-2)th to 0th floor by calling totalPaths(-1, dp), so now our tree diagram look like this:



Now, we will check if (n == 0), this is false as -1 is NOT equal to 0.
Then we are checking if (n < 0), this is true as -1 is less than 0.
This means we have found a path that goes below floor 0 which in our case is invalid and hence we return 0 as we do not want to count it, so now out tree diagram looks like this:

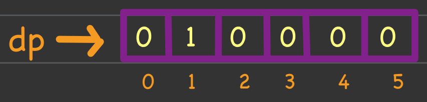


Now, the control goes back to totalPaths(1, dp)

Now we will add up the returned values from totalPaths(0, dp) and totalPaths(-1, dp)

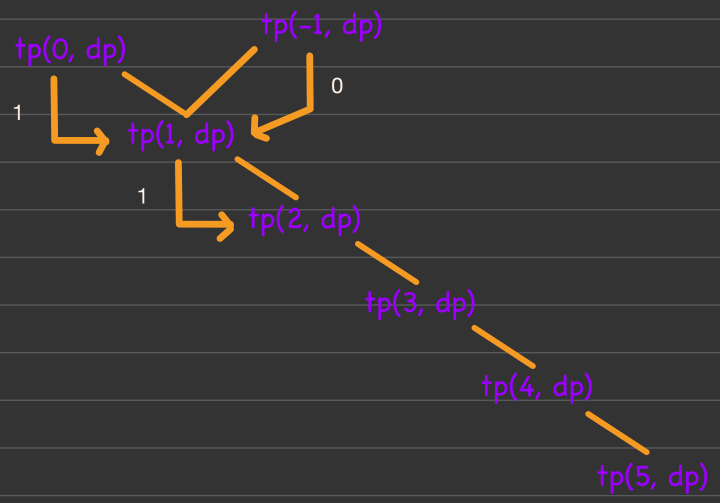
```
    totalPaths(1) = totalPaths(0, dp) + totalPaths(-1, dp)
    totalPaths(1) = 1 + 0
    totalPaths(1) = 1
```

After calculating the result of totalPaths(1, dp) we are now storing this result into dp[n] where n = 1, so our array would now look like this:

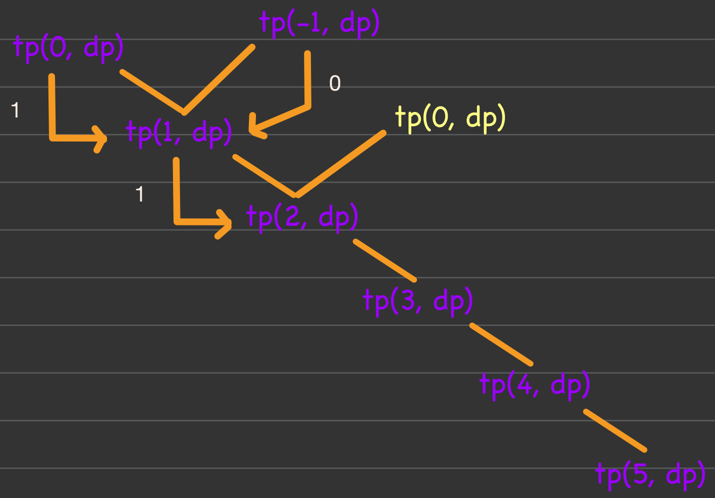


0	1	0	0	0	0
0	1	2	3	4	5

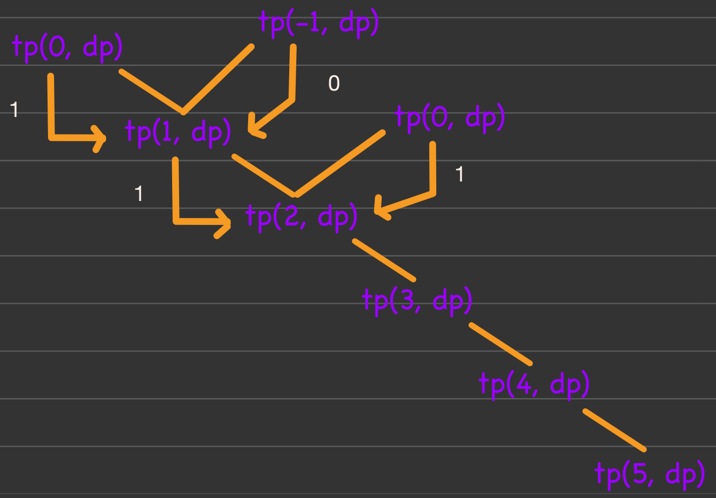
One we have finished storing the result we are simply returning the result of totalPaths(1, dp) and with that totalPaths(1, dp) will be completely done, so our tree diagram would end up looking like this:



Now the control will go back to where we left off in totalPaths(2, dp). the next call for totalPaths(2, dp) would be totalPaths(0, dp), so now out tree diagram looks like this:



Now, we will check if (n == 0), this is true as 0 is equal to 0. This means we have found a path from nth floor to 0th floor and we can clearly see that path in our tree diagram also. So we will return 1. And now our tree diagram would look like this:

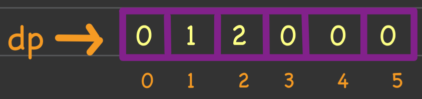


Now, the control goes back to totalPaths(2, dp)

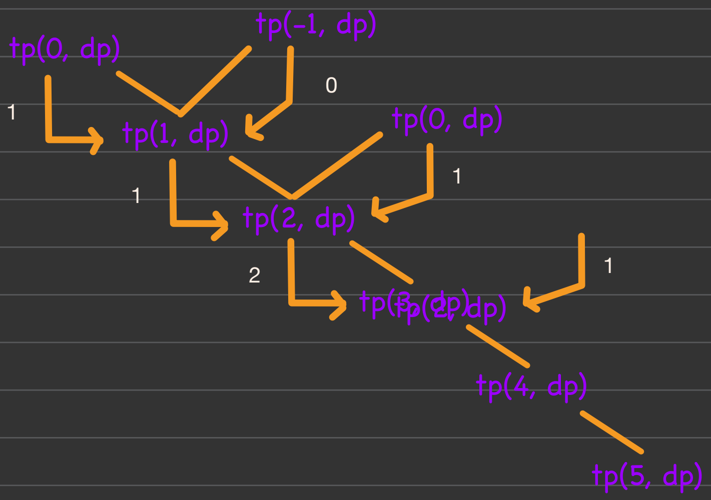
Now we will add up the returned values from totalPaths(1, dp) and totalPaths(0, dp)

totalPaths(2) = totalPaths(1, dp) + totalPaths(0, dp)
totalPaths(2) = 1 + 1
totalPaths(2) = 2

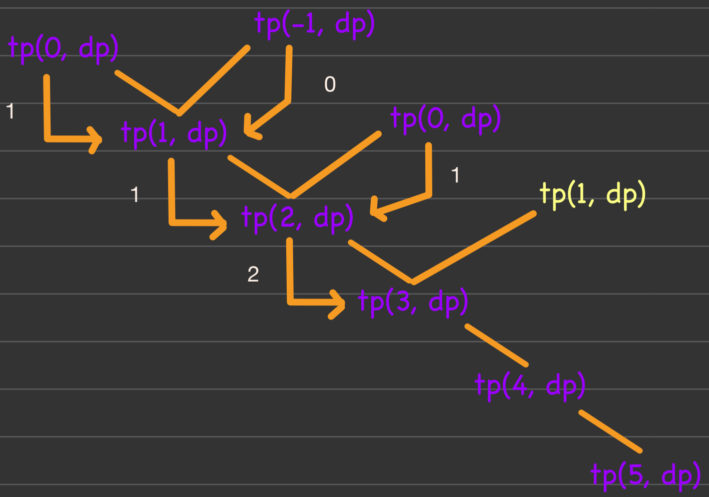
After calculating the result of totalPaths(2, dp) we are now storing this result into dp[n] where n = 2, so our array would now look like this:



Once we have finished storing the result we are simply returning the result of totalPaths(2, dp) and with that totalPaths(2, dp) will be completely done, so our tree diagram would end up looking like this:



Now the control will go back to where we left off in totalPaths(3, dp). the next call for totalPaths(3, dp) would be totalPaths(1, dp), so now out tree diagram looks like this:



NOTE: tp(1, dp) is called but actually we have already calculated tp(1, dp) before! Lets see how the use of **memoization** prevents recalculating tp(1, dp)

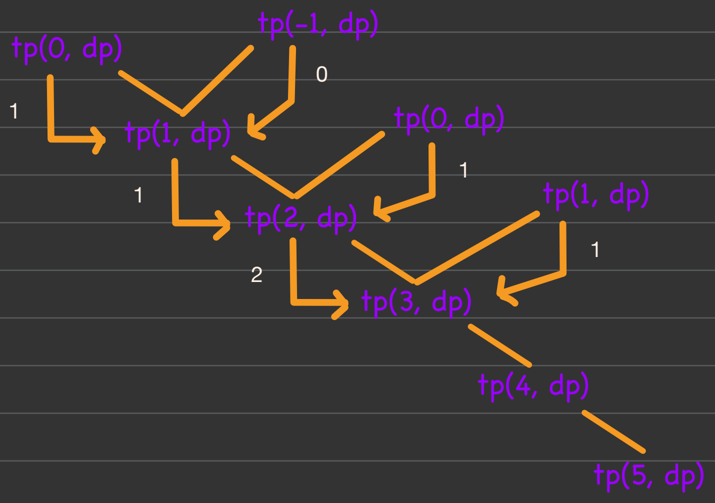
Quick reminder that our dp array currently looks like this:

dp →

0	1	2	0	0	0
0	1	2	3	4	5

Now, we will check if (n == 0), this is false as 1 is NOT equal to 0.
Then we are checking if (n < 0), this is false as 1 is NOT less than 0.
Next we will check if (dp[n] != 0), our n = 1 so we are checking if dp[1] != 0, this condition is true because the values stored in at index 1 of dp array is not 0. This means we have previously calculated and stored the result for totalPaths(1). Due to this we DO NOT need to re-calculate it. We simply return the stored values and to access and return the value we simply say return dp[n], where n=1 so we are essentially saying return the value at dp[1], which will return 1.

Now our tree diagram would look like this:



Now, the control goes back to totalPaths(3, dp)

Now we will add up the returned values from totalPaths(2, dp) and totalPaths(1, dp)

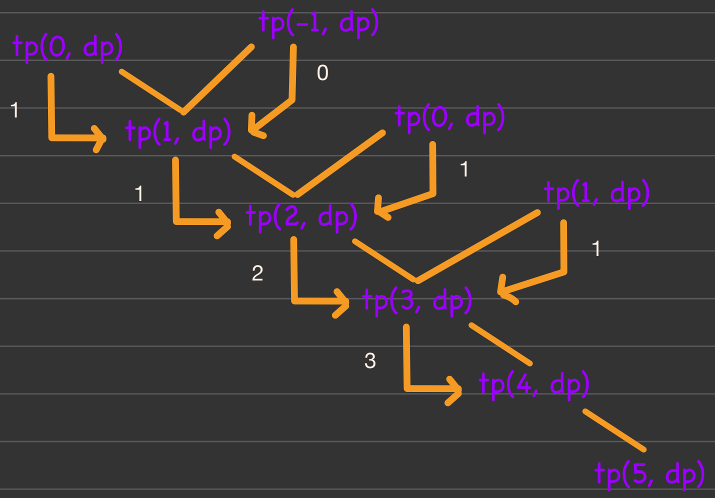
totalPaths(3) = totalPaths(2, dp) + totalPaths(1, dp)
totalPaths(3) = 2 + 1
totalPaths(3) = 3

After calculating the result of totalPaths(3, dp) we are now storing this result into dp[n] where n = 3, so our array would now look like this:

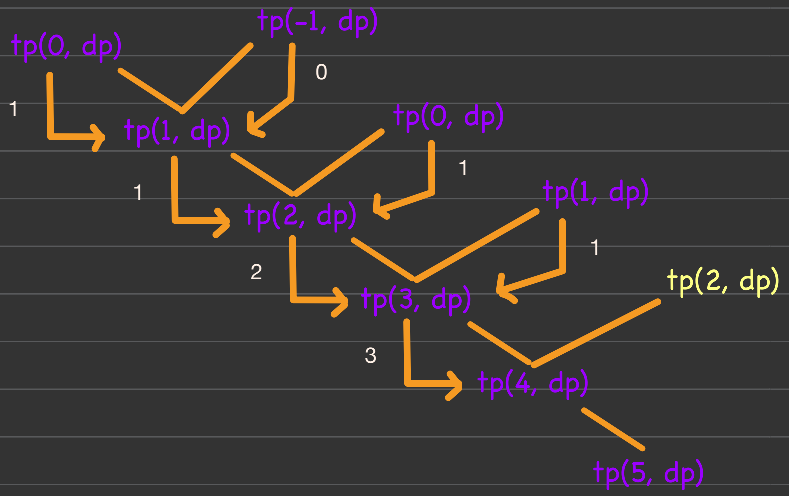
dp →

0	1	2	3	0	0
0	1	2	3	4	5

Once we have finished storing the result we are simply returning the result of totalPaths(3, dp) and with that totalPaths(3, dp) will be completely done, so our tree diagram would end up looking like this:



Now the control will go back to where we left off in totalPaths(4, dp). the next call for totalPaths(4, dp) would be totalPaths(2, dp), so now out tree diagram looks like this:



NOTE: tp(2, dp) is called but actually we have already calculated tp(2, dp) before! Lets see how the use of **memoization** prevents recalculating tp(2, dp)

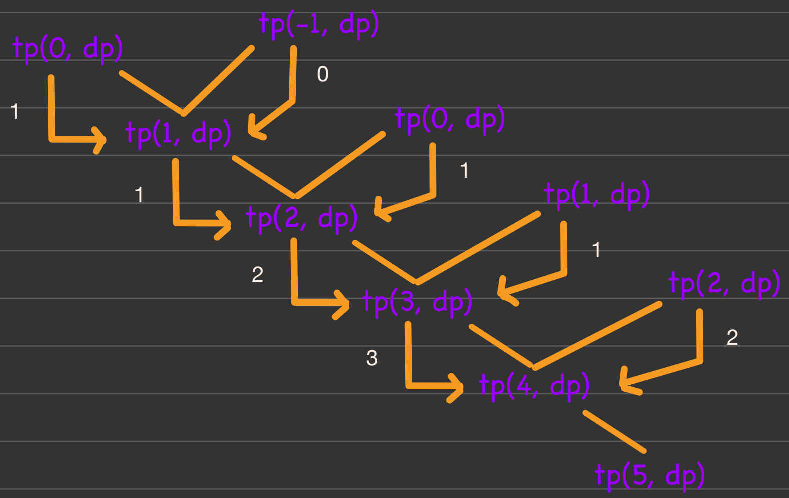
Quick reminder that our dp array currently looks like this:

dp →

0	1	2	3	0	0
0	1	2	3	4	5

Now, we will check if (n == 0), this is false as 2 is NOT equal to 0.
Then we are checking if (n < 0), this is false as 2 is NOT less than 0.
Next we will check if (dp[n] != 0), our n = 2 so we are checking if dp[2] != 0, this condition is true because the values stored in at index 2 of dp array is not 0. This means we have previously calculated and stored the result for totalPaths(2, dp). Due to this we DO NOT need to re-calculate it. We simply return the stored values and to access and return the value we simply say return dp[n], where n=2 so we are essentially saying return the value at dp[2], which will return 2.

Now our tree diagram would look like this:



Now, the control goes back to totalPaths(4, dp)

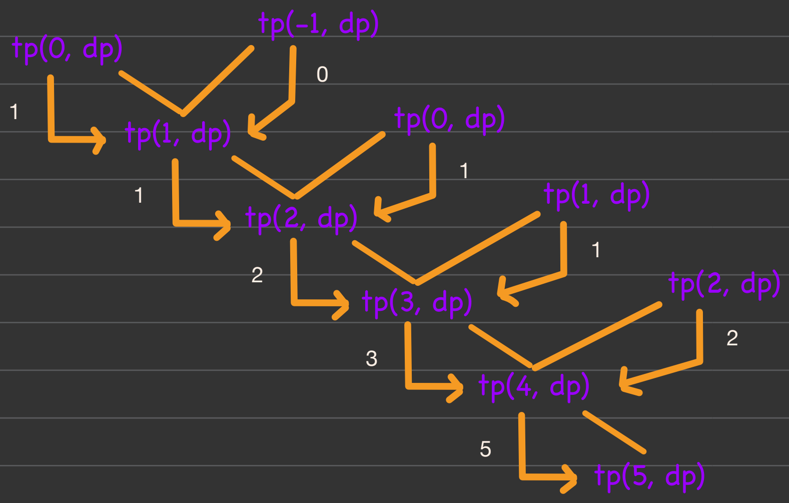
Now we will add up the returned values from totalPaths(3, dp) and totalPaths(2, dp)

totalPaths(4) = totalPaths(3, dp) + totalPaths(2, dp)
totalPaths(4) = 3 + 2
totalPaths(4) = 5

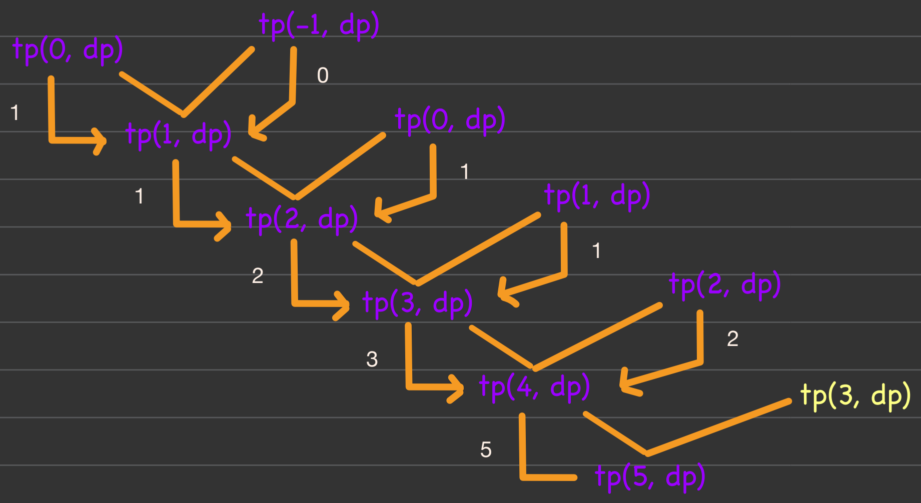
After calculating the result of totalPaths(4, dp) we are now storing this result into dp[n] where n = 4, so our array would now look like this:

dp →	0	1	2	3	5	0
	0	1	2	3	4	5

Once we have finished storing the result we are simply returning the result of totalPaths(4, dp) and with that totalPaths(4, dp) will be completely done, so our tree diagram would end up looking like this:

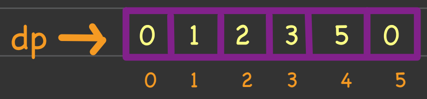


Now the control will go back to where we left off in totalPaths(5, dp). the next call for totalPaths(5, dp) would be totalPaths(3, dp), so now out tree diagram looks like this:



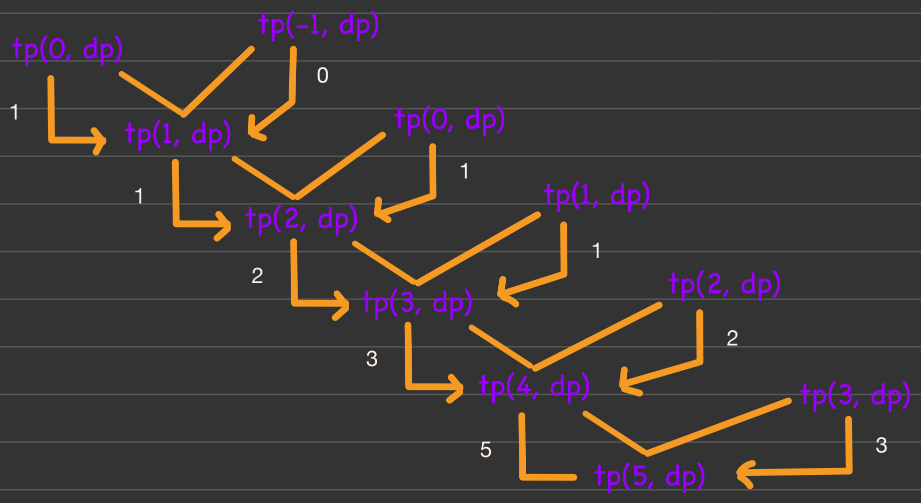
NOTE: tp(3, dp) is called but actually we have already calculated tp(3, dp) before! Lets see how the use of **memoization** prevents recalculating tp(3, dp)

Quick reminder that our dp array currently looks like this:



Now, we will check if (n == 0), this is false as 3 is NOT equal to 0.
Then we are checking if (n < 0), this is false as 3 is NOT less than 0.
Next we will check if (dp[n] != 0), our n = 3 so we are checking if dp[3] != 0, this condition is true because the values stored in at index 3 of dp array is not 0. This means we have previously calculated and stored the result for totalPaths(3, dp). Due to this we DO NOT need to re-calculate it. We simply return the stored values and to access and return the value we simply say return dp[n], where n=3 so we are essentially saying return the value at dp[3], which will return 3.

Now our tree diagram would look like this:



Now, the control goes back to totalPaths(5, dp)

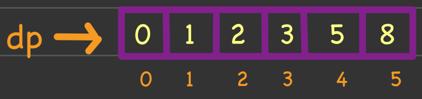
Now we will add up the returned values from totalPaths(4, dp) and totalPaths(3, dp)

totalPaths(5) = totalPaths(4, dp) + totalPaths(3, dp)

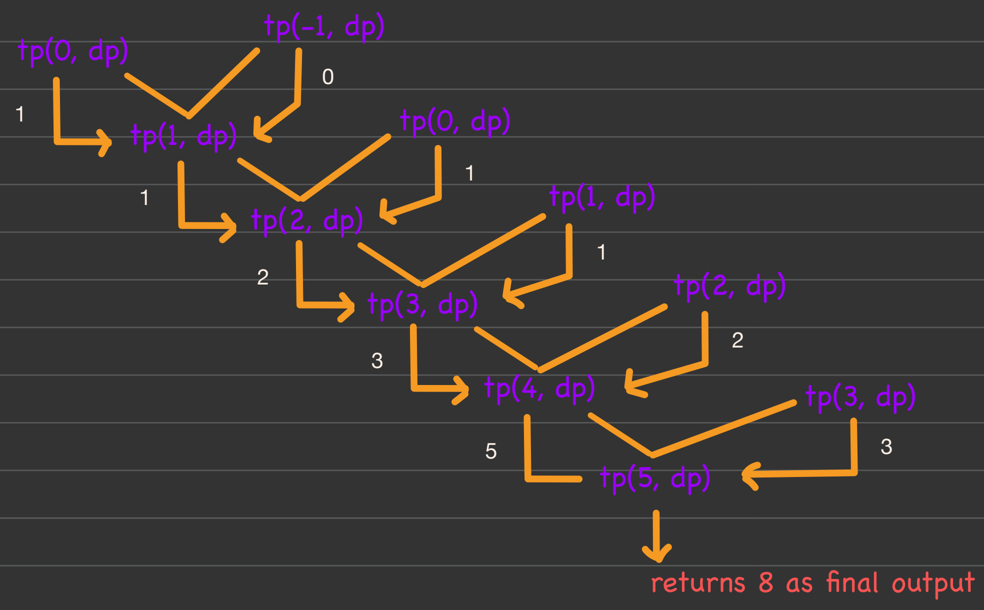
totalPaths(5) = 5 + 3

totalPaths(5) = 8

After calculating the result of totalPaths(5, dp) we are now storing this result into dp[n] where n = 5, so our array would now look like this:

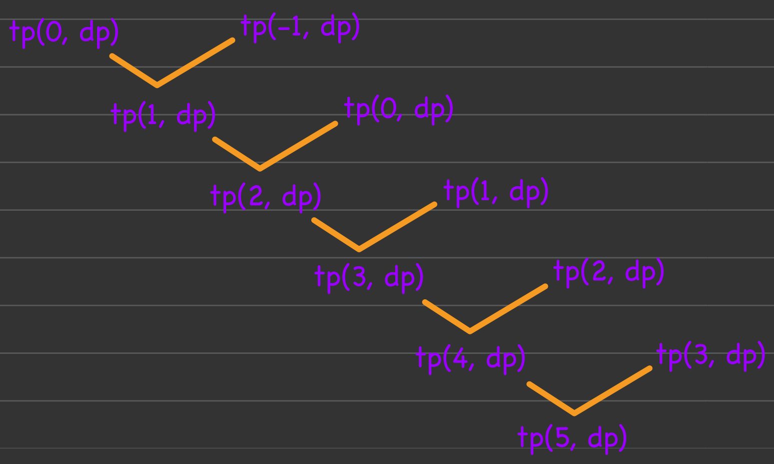


Once we have finished storing the result we are simply returning the result of totalPaths(5, dp) and with that totalPaths(5, dp) will be completely done, so our tree diagram would end up looking like this:



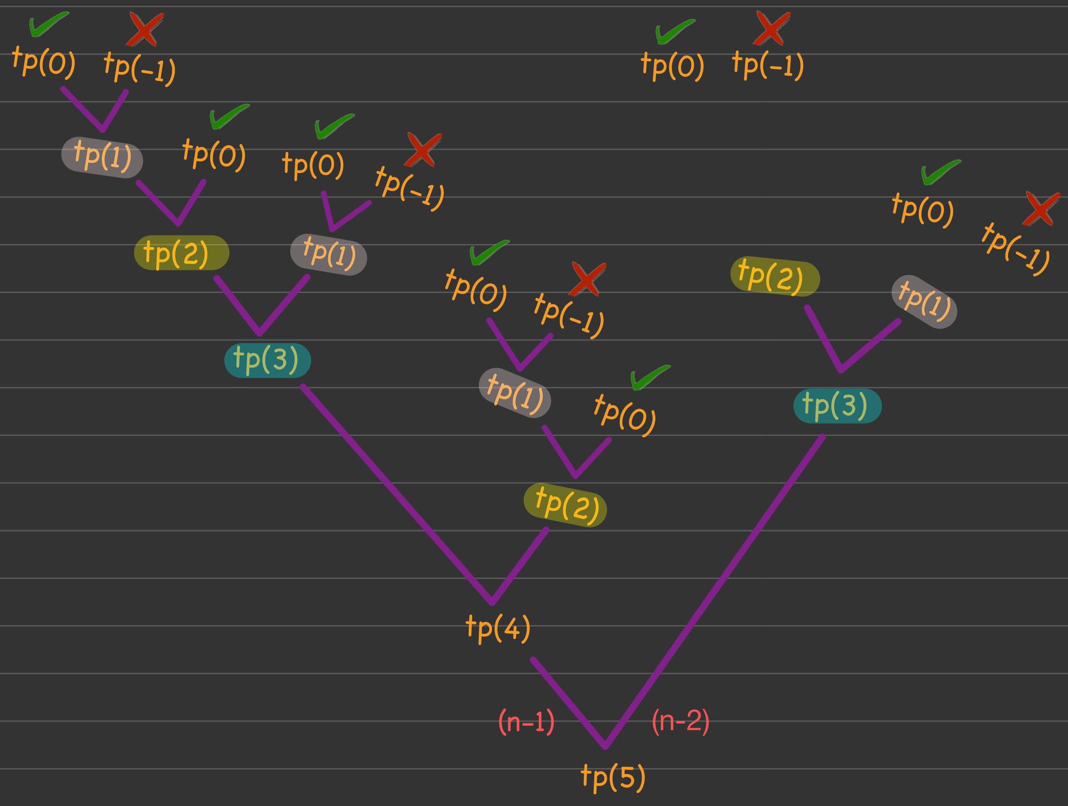
NOTE: the return values during tracing were just there to help you understand how we are getting the return values!

So our final tree diagram without return values would look like the following:

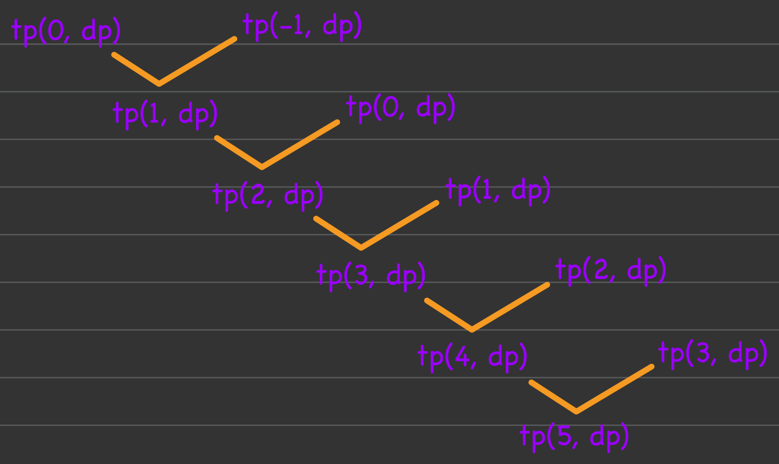


Now lets compare our dp approach tree diagram vs our recursive approach tree diagram! Hopefully you can see how dp is optimising the time as we are not recalculating already calculated values!

Recursive approach tree diagram



Dynamic Programming approach using memoization tree diagram



We can clearly see that in recursive approach we are always recalculating same values again and again meaning branches are being repeated but not in memoization approach! Also the time complexity for memoized approach is linear so O(n).