Table of Contents

[Fibonacci Number 2](#_Toc92612774)

[GridTraveler Memorization 6](#_Toc92612775)

[Memorization Recipe 10](#_Toc92612776)

[canSum Memorization 11](#_Toc92612777)

# Fibonacci Number

Write a function 'fib(n)' that takes in a number as an argument.

The function should return the n-th number of the Fibonacci sequence.

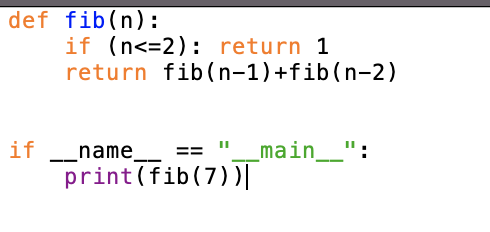
**Working of Fibonnaci Sequence:**

The 1st and 2nd number of the sequence is 1.

To generate the next number of the sequence, we sum the previous two.

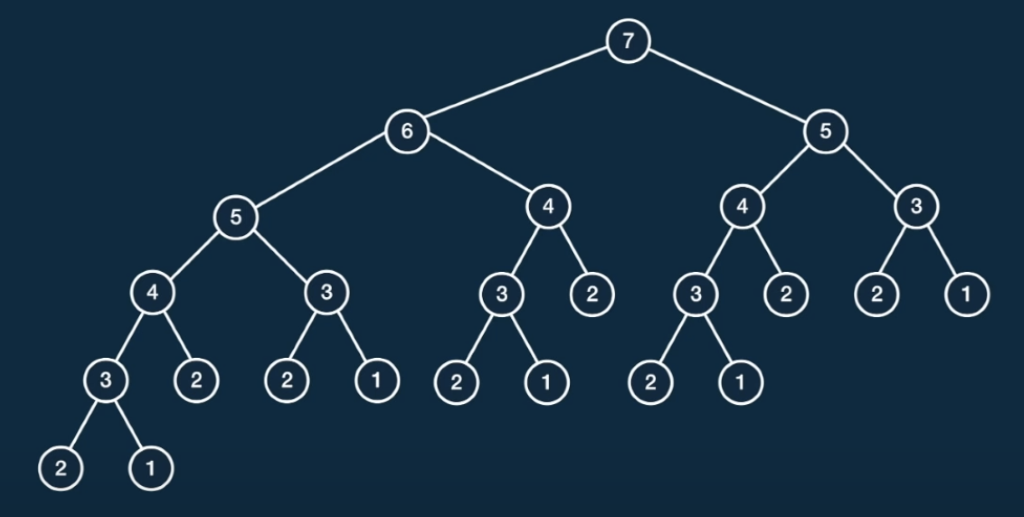
Example: 1,1,2,3,5,8,13,21,34,…

**Program for Fibonacci sequence:**

This code has one problem: which is As we want to get the Fibonacci number of higher nth term then it takes longer time to get the value .

This problem is due to the time complexity

**Visualizing this code in tree structure:**



For example: fib(7)  =  13

In tree structure:

Node 2 and 1 is a base case and it return 1 directly.

So, now we use bottom-up approach in which we add returning values of two branching nodes and get the value of their parent node. In this way, we get to the root node and finally get the answer by adding values of two branching nodes.

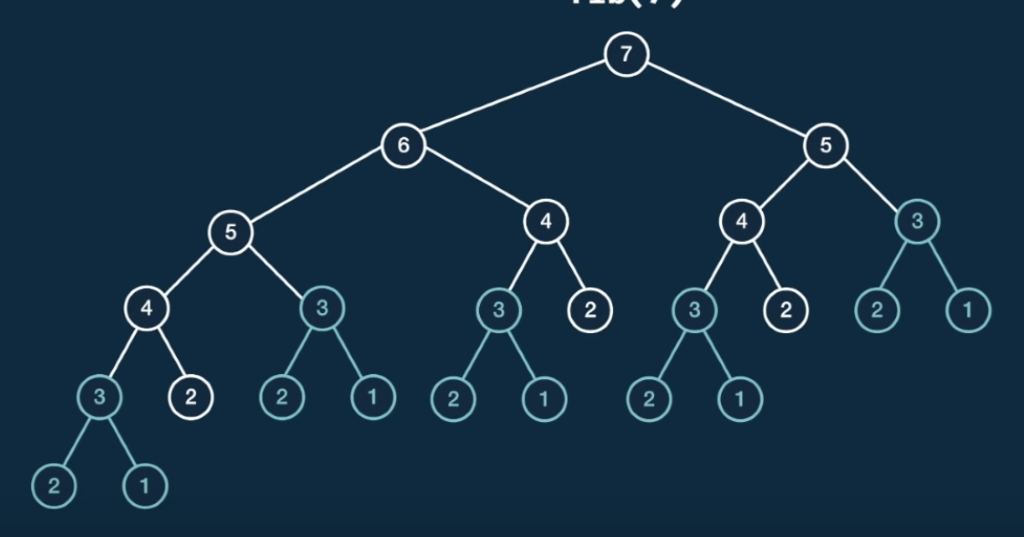
**Time Complexity of Above Code using Big-O Notation:**

Classic recursion implementation of fib is O(2n)

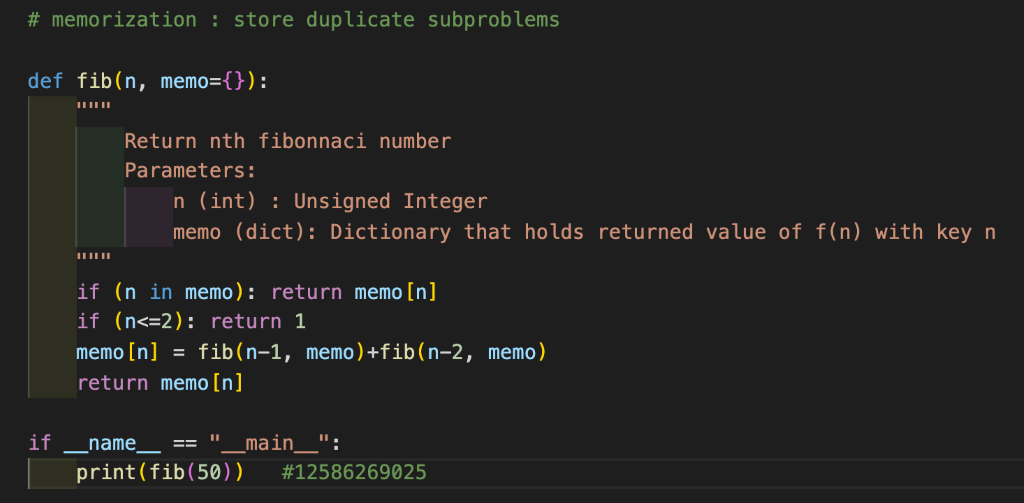
This is because we have two recursive calls in the function and each has a time complexity of O(n) and the size of nodes in the level increases by 2n.

So, with O(2n) time complexity, fib(50) is going to take 250steps which is roughly 1.12e+15 and this takes long time to complete.

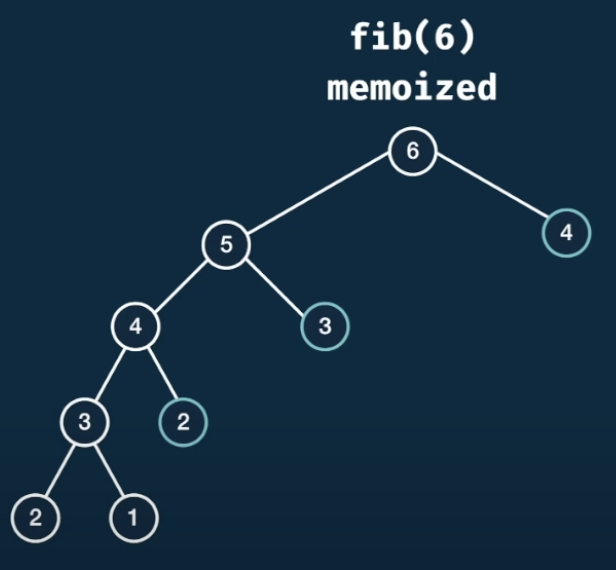
Making above code faster as the major bottleneck for the above code is time complexity.



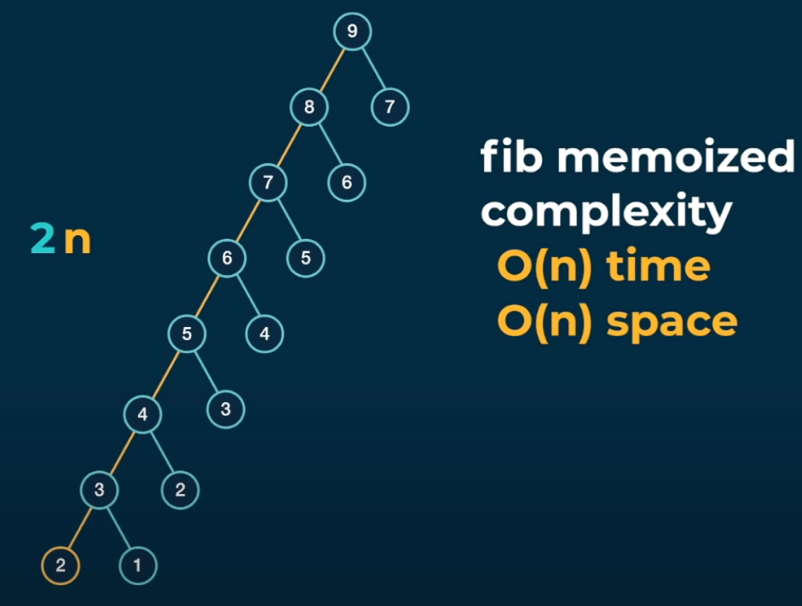
Looking at the above tree, we can a pattern that the sub-trees of node 3 (blue in color) is repeated, and similarly if we look at subtree of node 4 we can see that it is duplicated three times and also, subtree of node 5 is duplicated twice. So, each subtree is trying to answer the fib(5) which is not going to change. So, if we use the fib(5) calculated at the left node 5 then we can use the same value for the node 5 in the RHS which saves a huge time and this is called **Dynamic Programming.**



With memorization, tree of fib(6) is reduced to:



The time complexity of above code is **O(n)**and space complexity is **O(n).**



There are n pairs.

# GridTraveler Memorization

Say that you are a traveler on a 2D grid. You being in the top-left corner and your goal is to travel to the bottom-right corner. You may move down or right.

In how many ways you can travel to the goal on a grid with dimensions m \* n?

Write a function ‘gridTraveler(m,n)’ that calculates this?

Let’s say we are given gridTraveler(2,3) and is asking in how many ways can ways we can travel from top-left to botton-right in 2x3 grid which is 3. Lets visualize this:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | S |  |  | |  |  | E | | |  |  |  | | --- | --- | --- | | S |  |  | |  |  | E | | |  |  |  | | --- | --- | --- | | S |  |  | |  |  | E | |

So, these are the three ways to go from S to E.

If we are given 1x1 grid then there will be 1 unique way to travel, so in this case we do nothing.

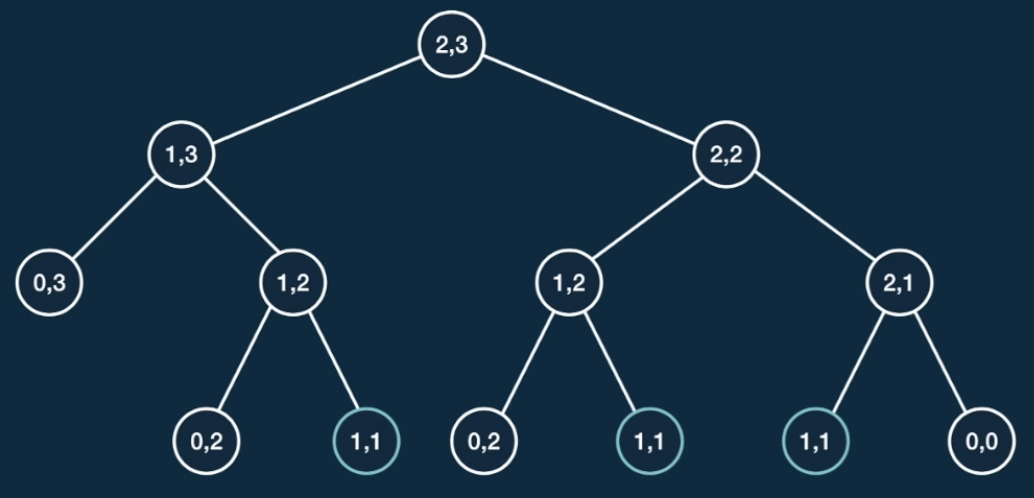
S/E

So, in some case we might be given a grid of 0x1, this is strange because if there 0 rows and 1 columns then the grid is empty which means there is 0 way to travel. In similar way, if we are given 1x0 then the same case i.e. there is no way to travel(0).

So, for gridTraveler(3,3) =

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | S |  |  | |  |  |  | |  |  | E | | At first, we have a playable area of 3x3. |
| |  |  |  | | --- | --- | --- | |  |  |  | | S |  |  | |  |  | E | | If we move down, then the playable area is reduced/shrinks to 2x3. So, now the question changes to how many ways can we travel in 2x3 grid. |
| |  |  |  | | --- | --- | --- | |  |  |  | |  | S |  | |  |  | E | | Again, moving to the right shrink’s grid into 2x2 grid. |
| |  |  |  | | --- | --- | --- | |  |  |  | |  |  |  | |  | S | E | | Further moving down, the grid shrinks into 1x2 grid |
| |  |  |  | | --- | --- | --- | |  |  |  | |  |  |  | |  |  | S/E | | Finally, the grid is minimized into 1x1 so there is only a way to travel |

Visualizing the gridTraveler(2,3) in tree-based approach:



1 + 2 = 3

1 + 1 = 2

1 + 0 = 1

1 + 0 = 1

1 + 0 = 1

1 + 0 = 1

0

0

0

0

1

1

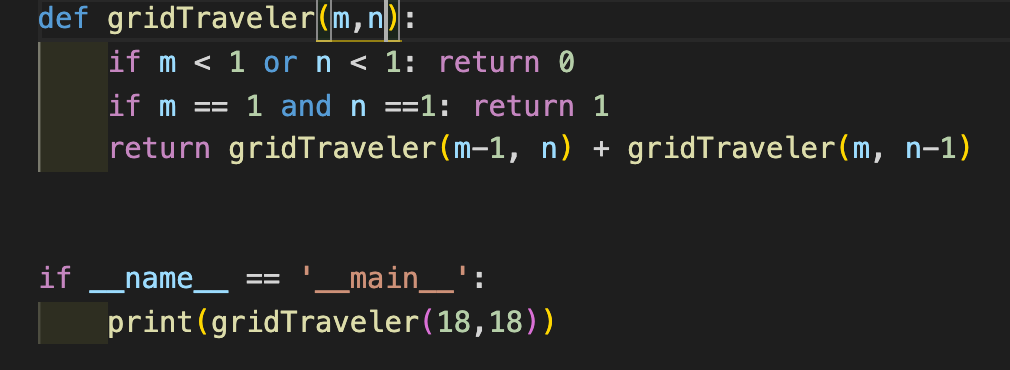
1

Right

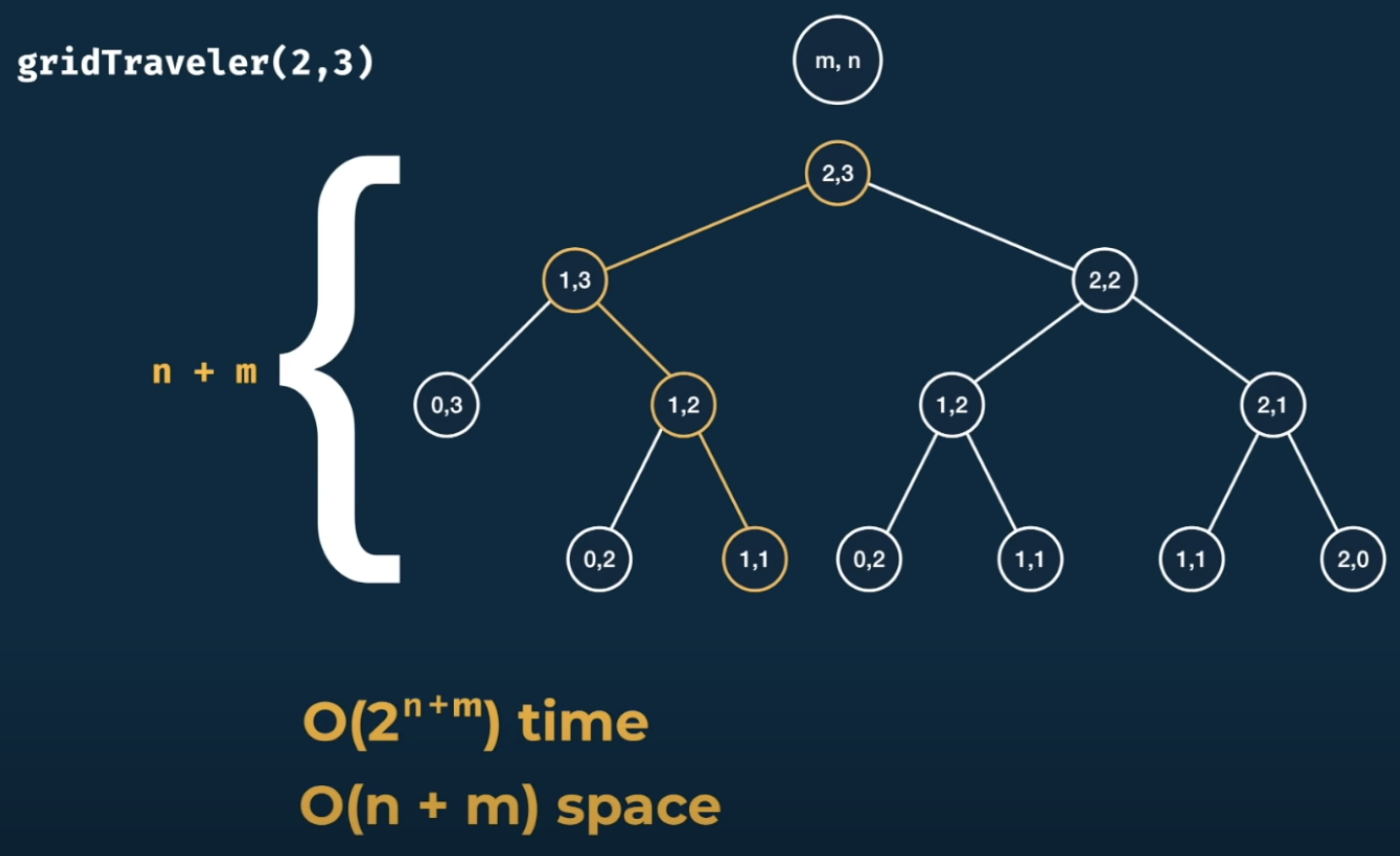
Down

So, by visualizing the recursive function in tree-based approach, we can see the output of 2x3 grid is 3 i.e. ways to travel.

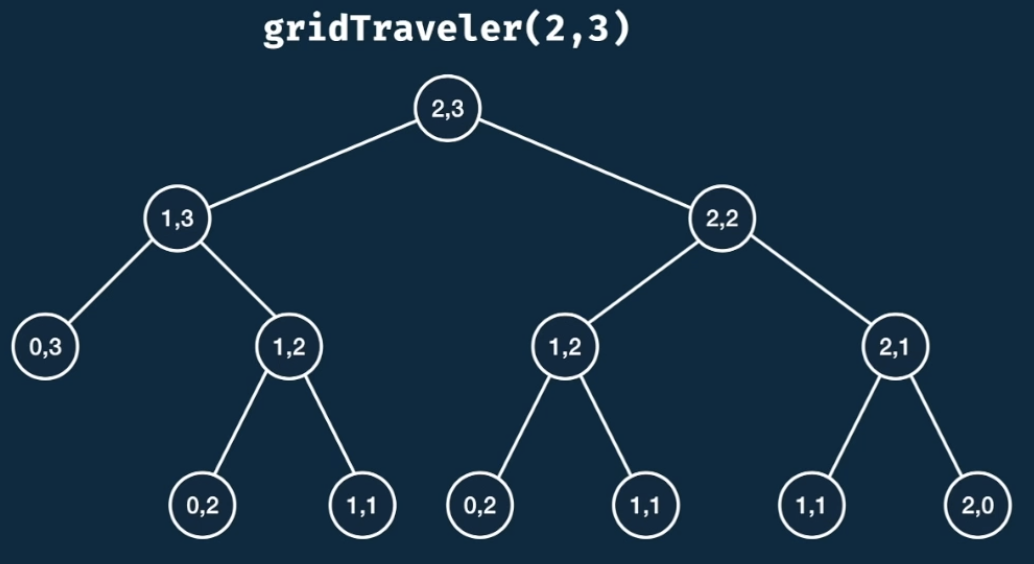
Also, by looking at the above tree, we can also find out which the combination of moves that we reach to the solution. For instance, taking down, right, right leads to the solution, another sets of moves is right, down, right and at last, taking right, right, down leads to the solution. Basically, there are three ways to reach the solution.



Time Complexity of above code:



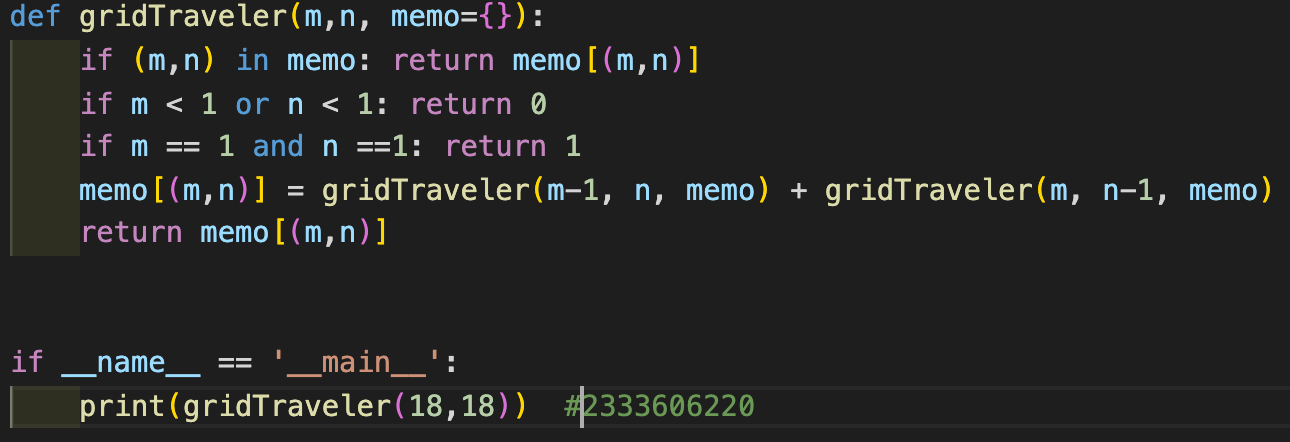
Let’s improve the time complexity:



Carefully looking at the tree, we can see some duplicated in the node (1,2) as there are two duplicates of this node. So, we can use memorization feature to prevent doing the work again. Also, looking at the node (2,1) it is equal to node (1,2) as the rows and columns are flipped out but the outputs the same result.

gridTraveler(a,b) = gridTraveler(b,a)

Implementing memorization in code:



Improved Time Complexity:

Let’s say we have to find the number of ways to travel in 4x3 grid using gridTraveler(4,3) function then (4,3) is a top level which is just (m,n) in general way.

Looking at the other nodes of the tree, they are going to be utmost 4,3 and others less than that so there is a range of values. So, the possible values of nodes for 4,3 grid are:

**m: {0, 1, 2, 3, 4 } n: {0, 1, 2, 3 }**

So, there are m choices for first number of node and n choices for second number of the node. And also, we know that we do not have to go through lots of duplicate nodes because of memorization so the number of possible nodes are **m \* n**.



# Memorization Recipe

1. **Make it work.**

* Visualize the problem as a tree
* Implement the tree using recursion
* Test it

1. **Make it efficient**

* Add a memo object
* Add a base case to return memo values
* Store return values into the memo

# canSum Memorization

Write a function ‘canSum(targetSum, numbers)’ that takes in a targetSum and an array of numbers as arguments.

The function should return a Boolean indicating whether or not it is possible to generate the targetSum using numbers from the array.

You may use an element of the array as many times as needed.

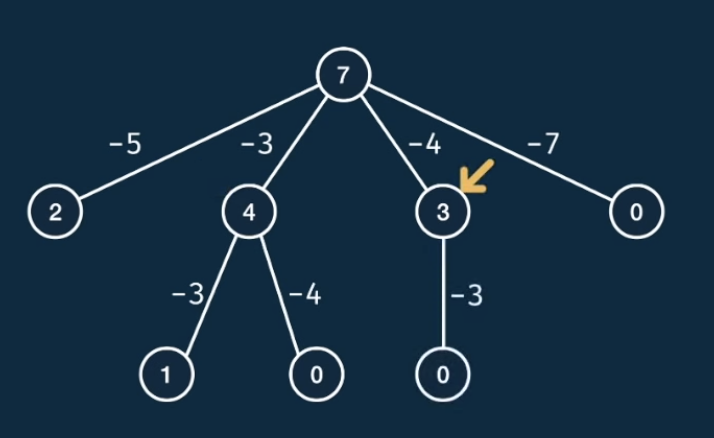
You may assume that all input numbers are nonnegative.

For example, canSum(7, [5,3,4,7]) = true as there are two ways to get targetSum 3+4 and 7

canSum(4, [2,4]) = false as targetSum cannot be achieved by summing the numbers in an array.

Let’s visualize a function canSum(7, [5,3,4,7])

Taking root node as a targetSum as every function call is going to receive a same array



**T + F = T**

**T + F = T**

**F**

**F**

**T**

**T**

**T**

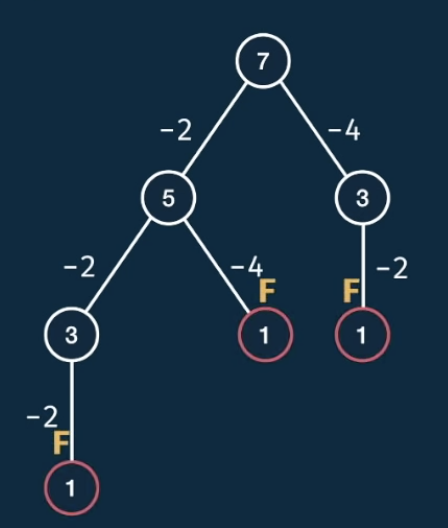
Now, shrinking the size of the problem. As there are 4 items in the array so we can branch the root node into four branches and subtracting the number from root node decreases the targetSum.

Carrying out the problem, but we have to be careful as on the left side we have a node with a target sum of 2 but we have a options 5,3,4,7 in a array which is not compatible as subtracting will give a negative target sum so, we cannot further flush out this target sum. So, the second node has target sum of 4 which can be further branched into two nodes which is 1 and 0. Lastly, we can branch 3 node further only once into 0 target sum.

Looking at all the base nodes, we can say that they all bottom-out in a base case as there is not further choices that we can take. So, carefully looking at the above tree structure we have a 0 base case which is really nice as in that base case we have found out that it can really generate the original target sum and these nodes should return true to their parent node and those nodes which are non-zero and cannot break down further then they should return false to their parent.

As stated in the question, either we can generate target sum by summing the numbers in an array. So, if one is True then the output is True.

For example: canSum(7,[2,4]) = False



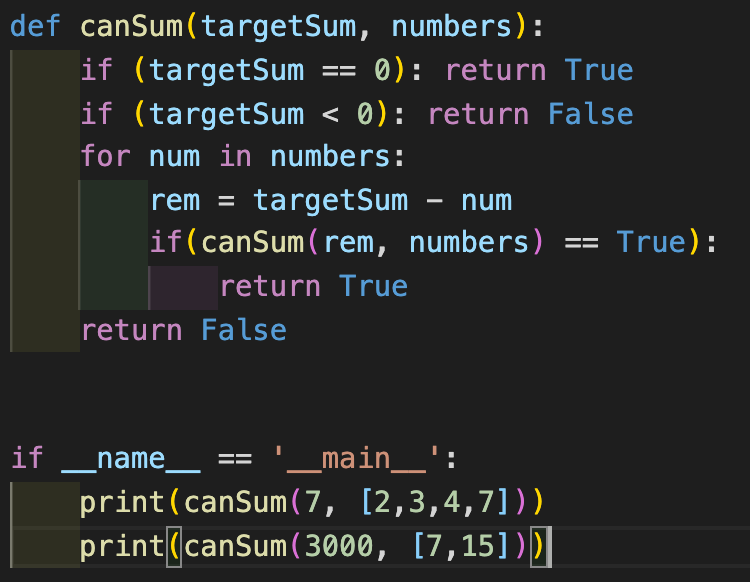
**F**

**F**

**F**

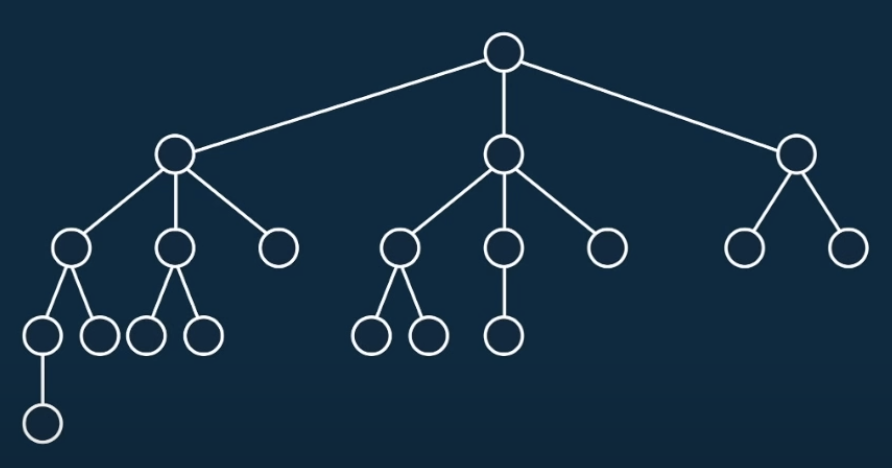
**F**

Implementation of canSum function using Python:



Time Complexity of above code:

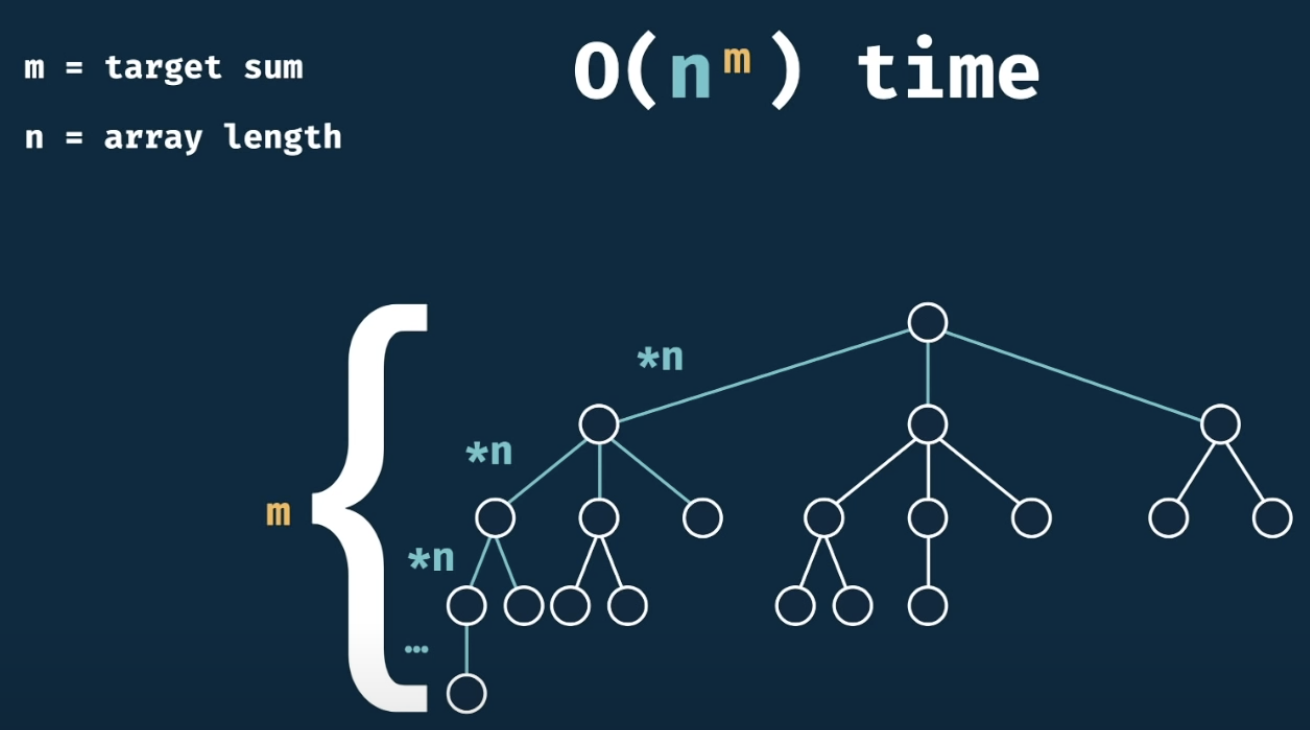
canSum(8, [2,3,5]) = True



This is the generalized structure of tree for the above function. This function has two inputs and let’s say:

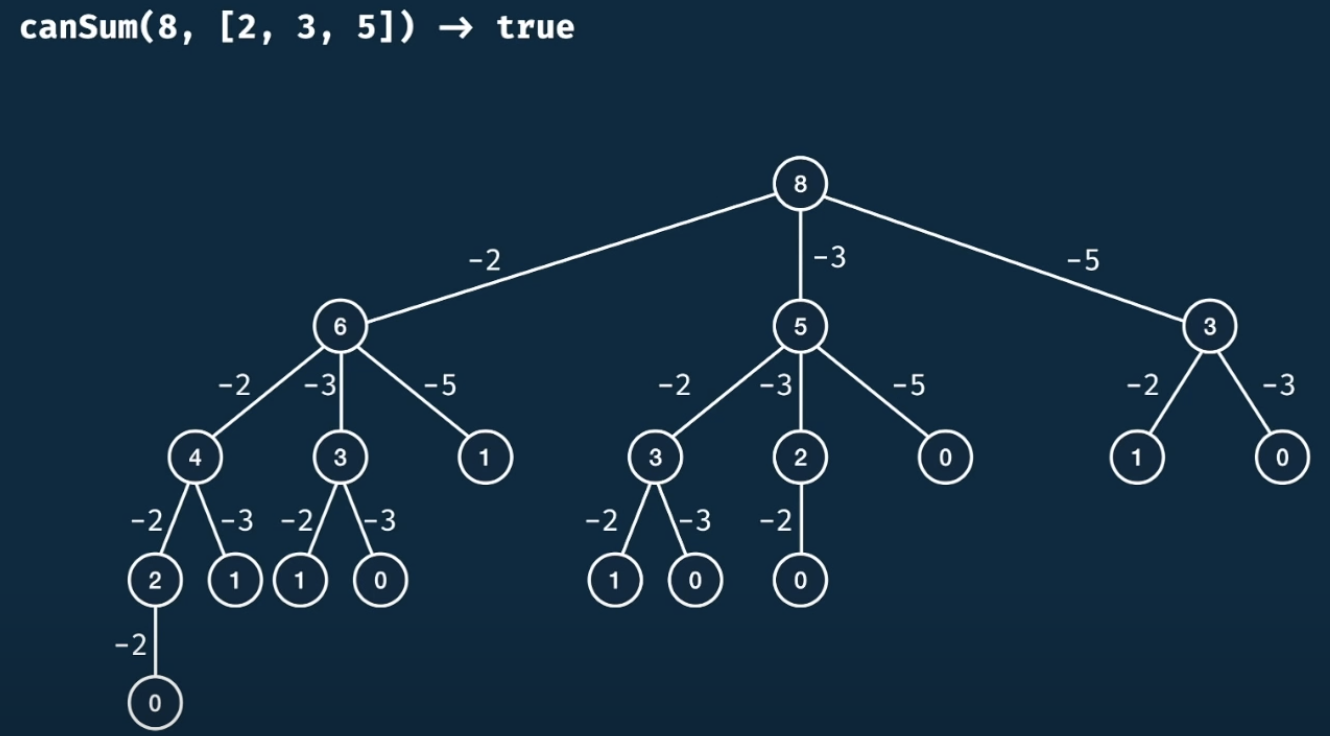
m = target sum n = array length

* Height of the tree is m as in the worst case we have subtract 1 again and again till we reach base node.
* The branching factor i.e. how the number of nodes changes from one level to another. In the above tree, the number of branching factor is 3 and generally it is the length of an array i.e. n.
* So, if we have m levels and from one level to next we multiply the number of nodes by n and this will give the exponential time complexity.



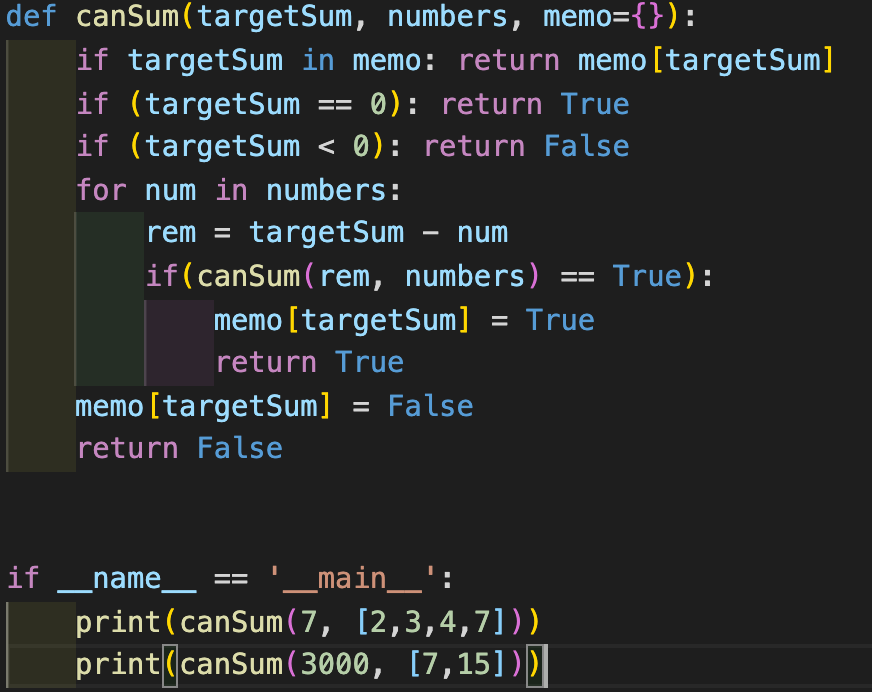
* From the above diagram, we can also derive the space complexity so the space used by the call stack is just a height of a tree i.e. O(m).

**Improving the time complexity**



Looking at the above tree, we can see that the target sum 3 node is repeated 3 times so we can store that target sum in a list and return value when that target sum comes up in a tree.

**Improved Code Using Memorization**



New complexity:

m = target sum n = array length

|  |  |
| --- | --- |
| Brute force | Memorized |
| O(nm) time  O(m) space | O(m \*n) time  O(m) space |

# howSum Memorization

Write a function ‘howSum(targetSum, numbers)’ that takes in a targetSum and an array of numbers as arguments.

The function should return an array containing any combination of elements that add up to exactly the targetSum. If there is no combination that adds up to the targetSum, then return null.

If there are multiple combination possible, you may return any single one.