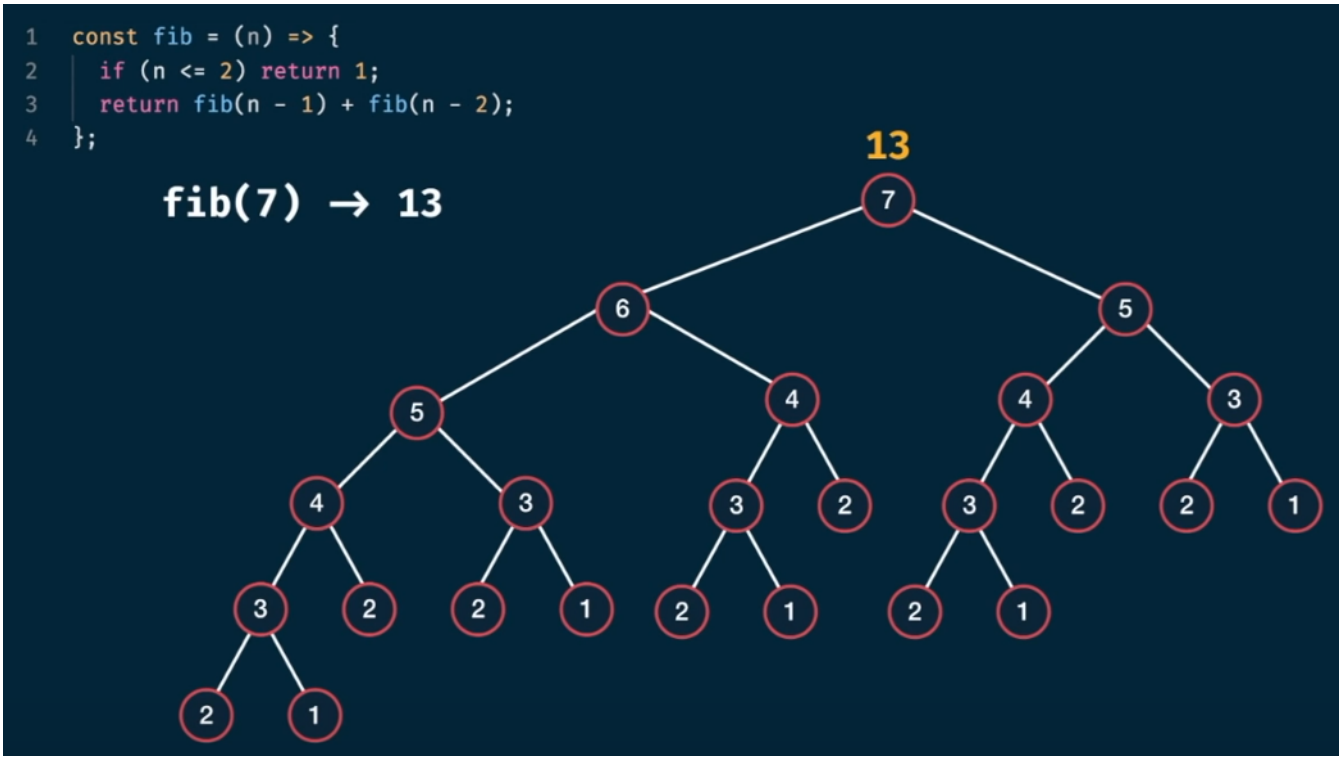
DP. Memoization

Fibonacci Sequence

```
int fib(int n){
   if(n <= 2){
       return 1;
   }
   else {
       return fib(n-1) + fib(n-2);
   }
}
```

```
const fib = (n) =>{
    if(n <= 2) return 1;
    return fib(n-1) + fib(n-2);
}

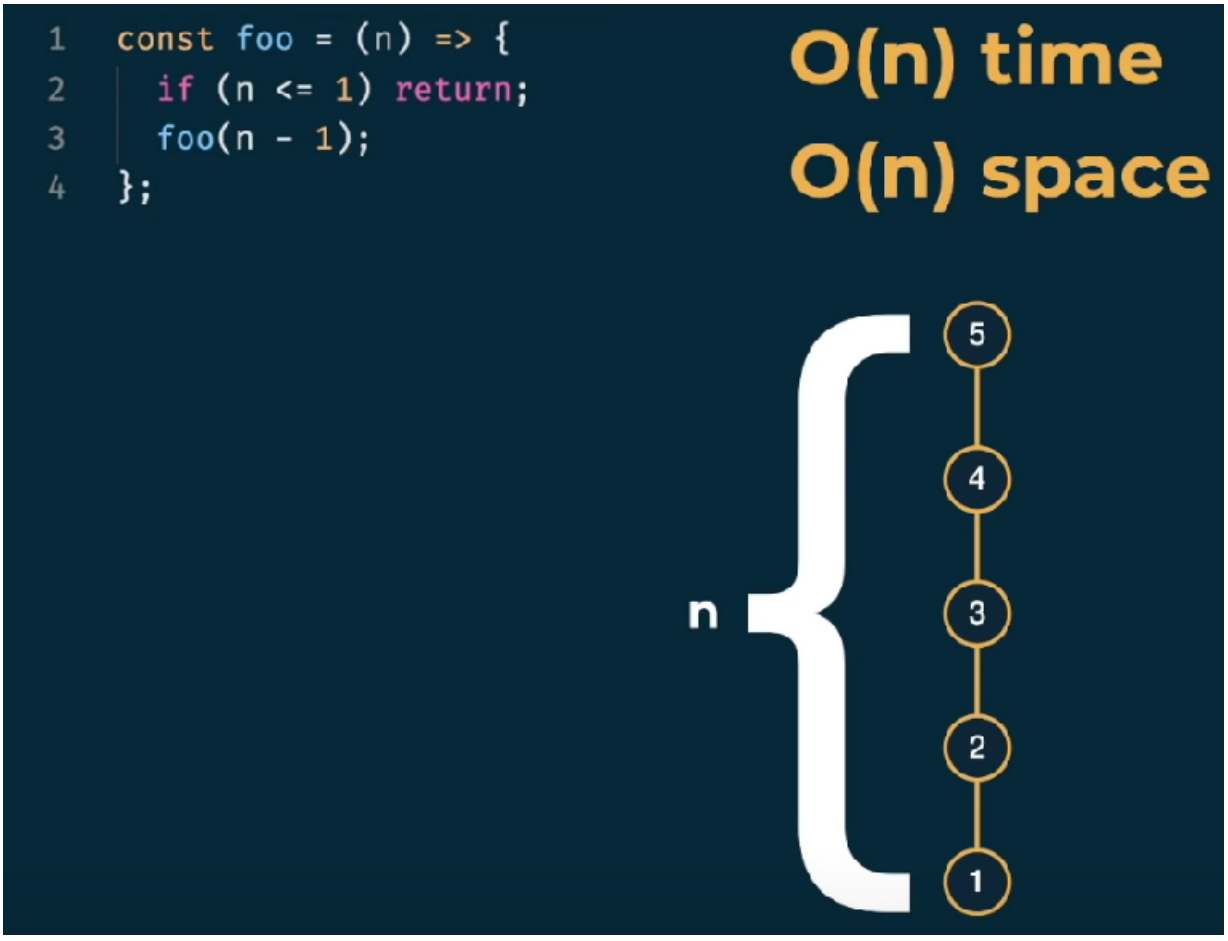
console.log(fib(7));
```



- Time complexity of fib function is `O(2^n)`

Time and space complexity of recursive function

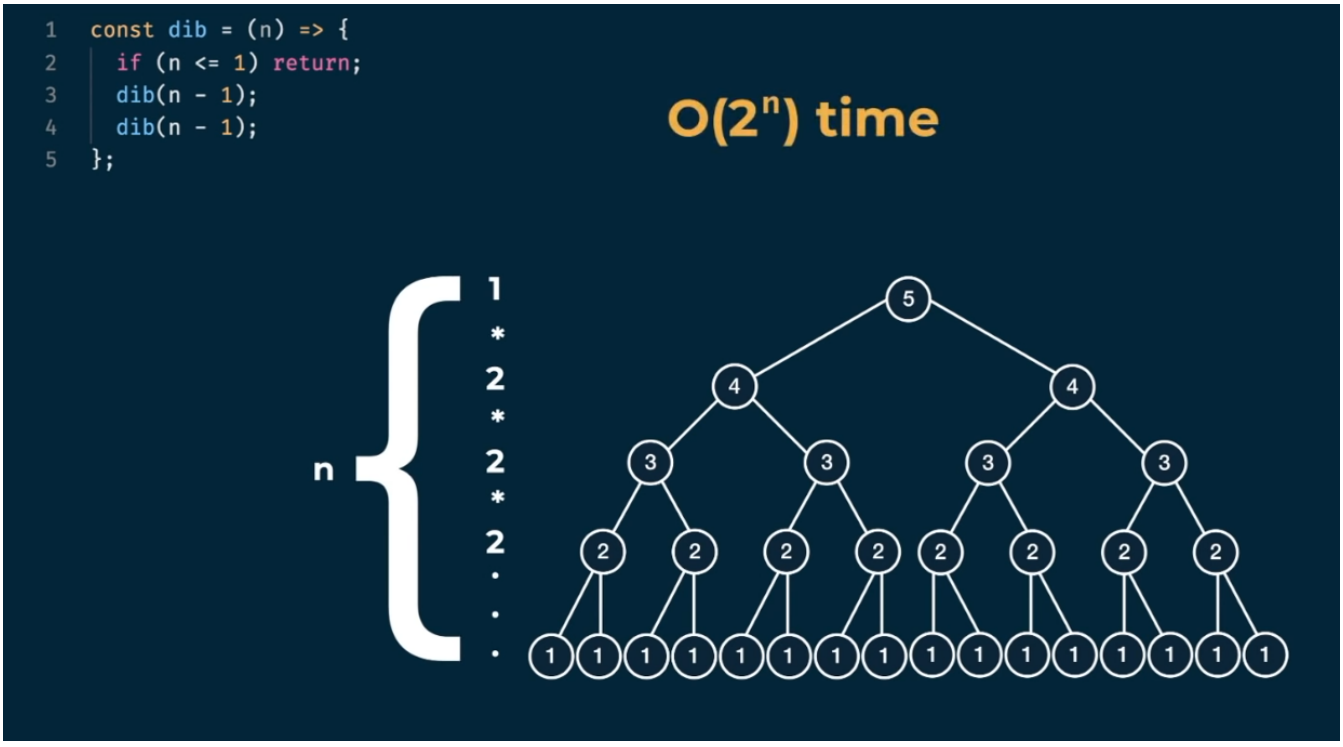
- Analysing time complexity of a recursive function
-

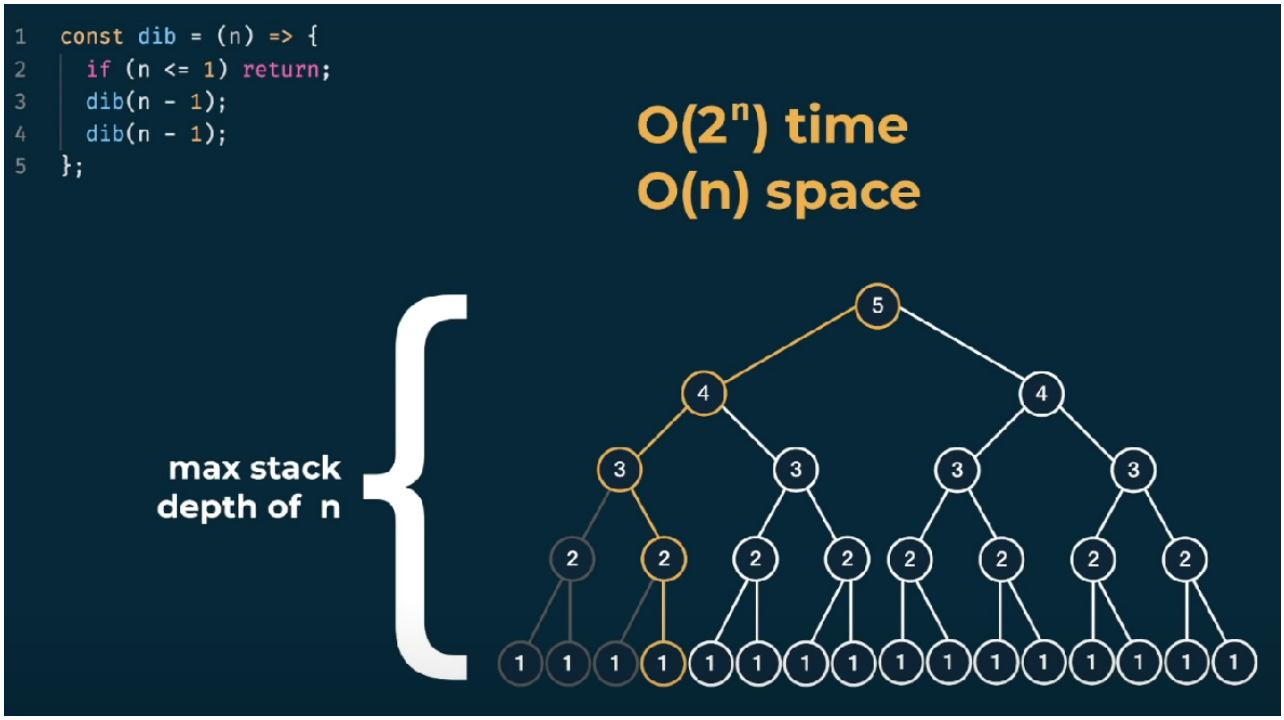


- Space complexity
  - when we analyse space complexity of recursive functions We should include any of the additional stack space
- for a function

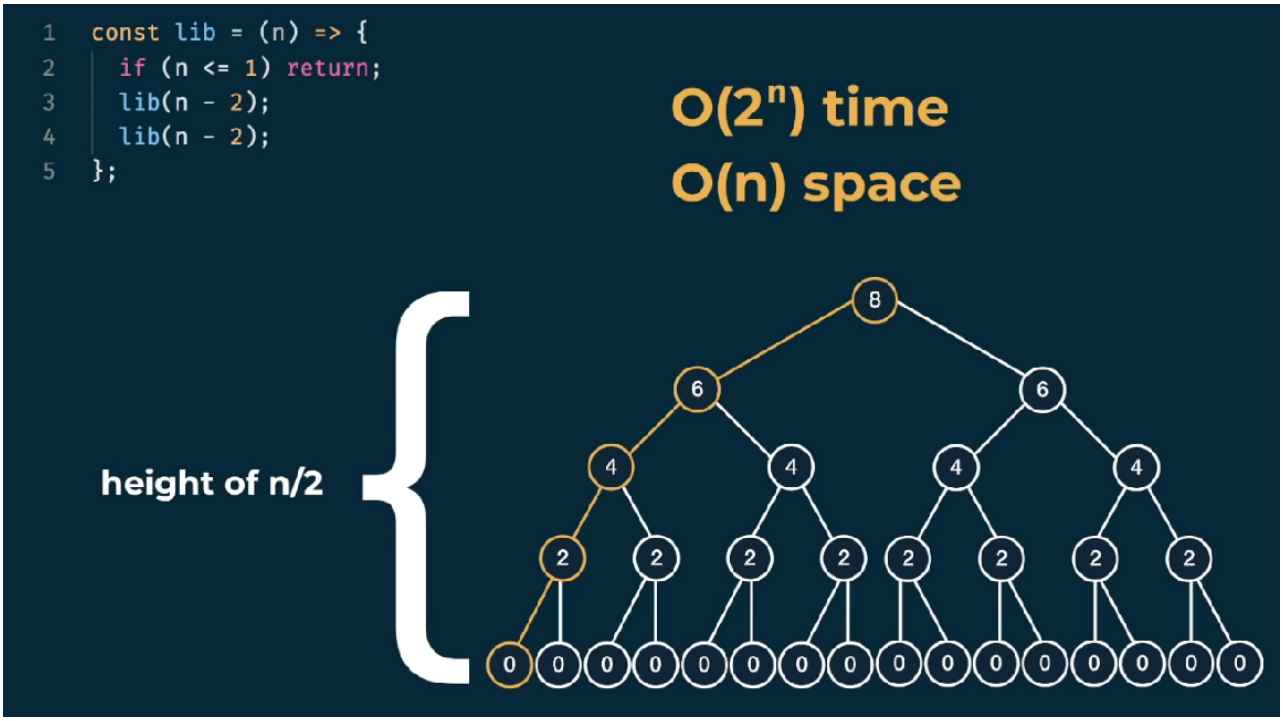
```
void dib(int n){
    if(n <= 1){
        return;
    }
    dib(n-1);
    dib(n-1);
}
```

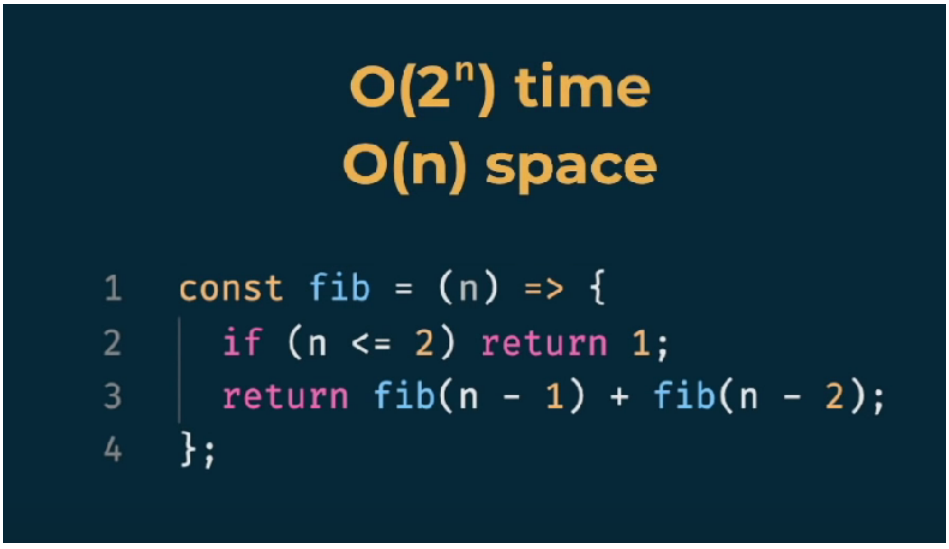
- Height is the distance between the root node and the furthest leaf node
- At every level we are calling the functions twice doubling the number of function calls
- for `n` levels our time complexity becomes `O(2^n)`



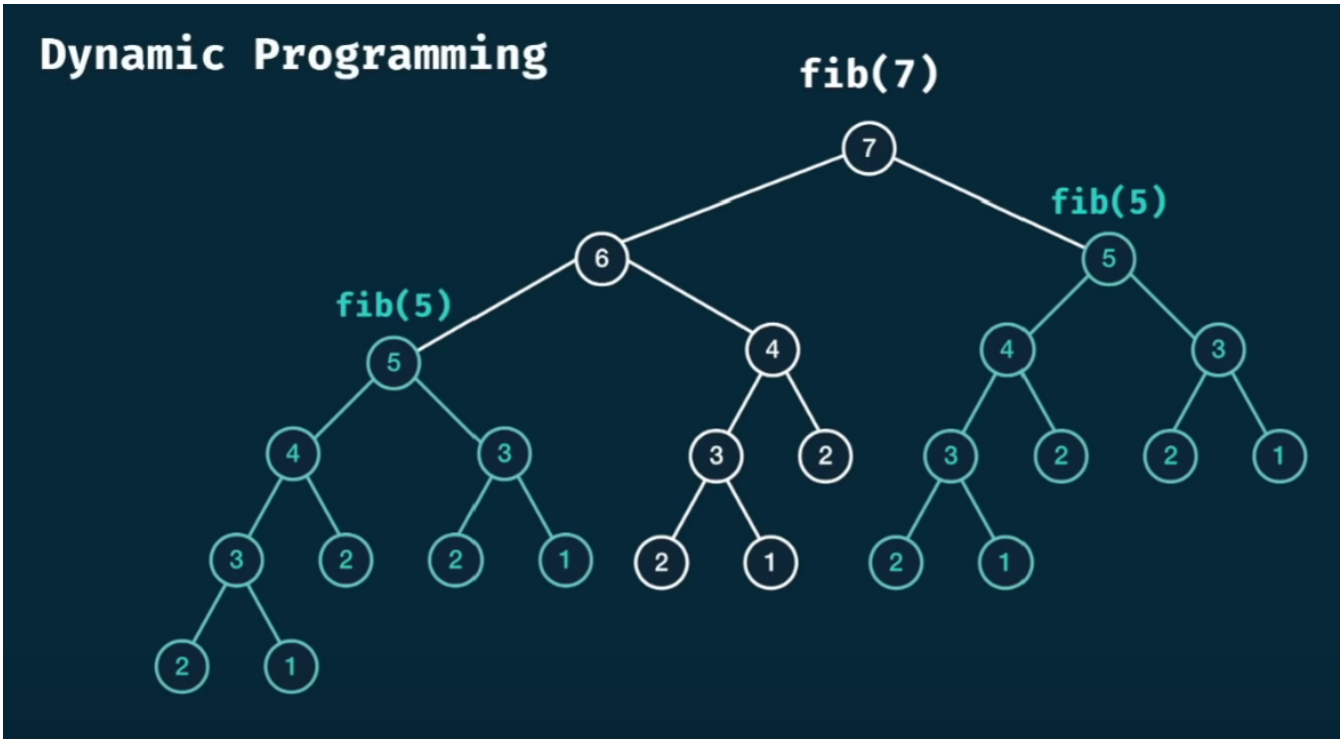
- At any point we use only 5 stacks for `dib(5)`
- Because after reaching five stack calls the functions returns
- it is only after return from left one we enter the right one.
- 

- Time complexity is `O(n^2)` and space complexity is `O(n)`

- 

- Our Fibonacci function has `O(2^n)` time complexity and `O(n)` space complexity
- 

Dynamic programming definition



- When we have some larger problem, we can decompose into smaller instances of the same problem. This concept of break down is called **Dynamic Programming**

memoization

- We store duplicate subproblems as we can get those results later on.
- To implement a memoization we pick a fast access data structure
- Usually a HashMap or a JavaScript object
- JavaScript passes data to function by reference

Memoization for Fibonacci

```
int fib(int n){
    // create a memo
    static unordered_map<int, int> memo;
    // find the number
    auto it = memo.find(n);
    if(it != memo.end()){
        return memo[n];
    }

    if(n <= 2){
        return 1;
    }

    // add to the memo
    memo[n] = fib(n-1) + fib(n-2);

    // return from memo
    return memo[n];
}
```

```
const fib = (n, memo = {}) =>{
    // check in memo
    if(n in memo) {
        return memo[n];
    }

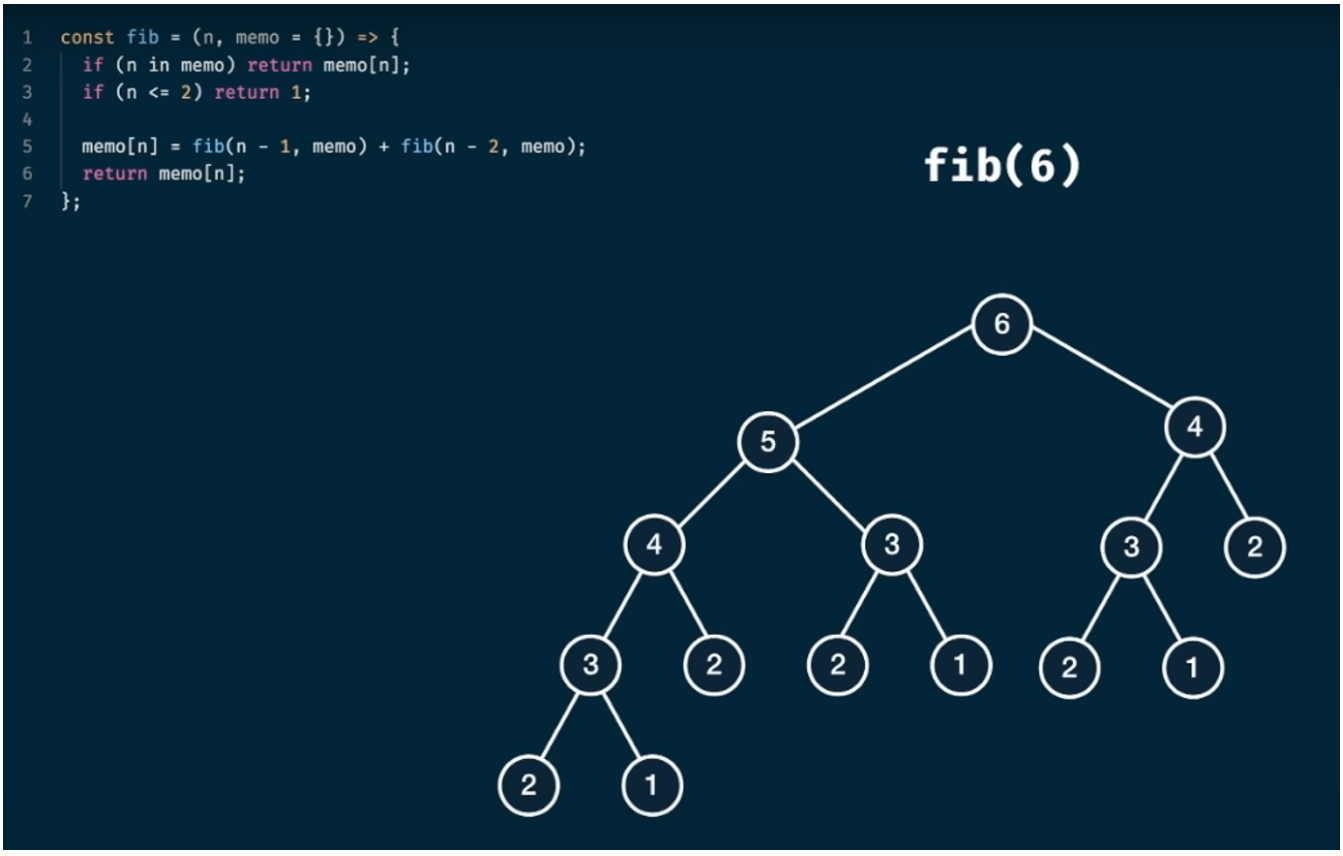
    if(n <= 2) {
        return 1;
    }

    memo[n] = fib(n-1, memo) + fib(n-2, memo);
    return memo[n];
}

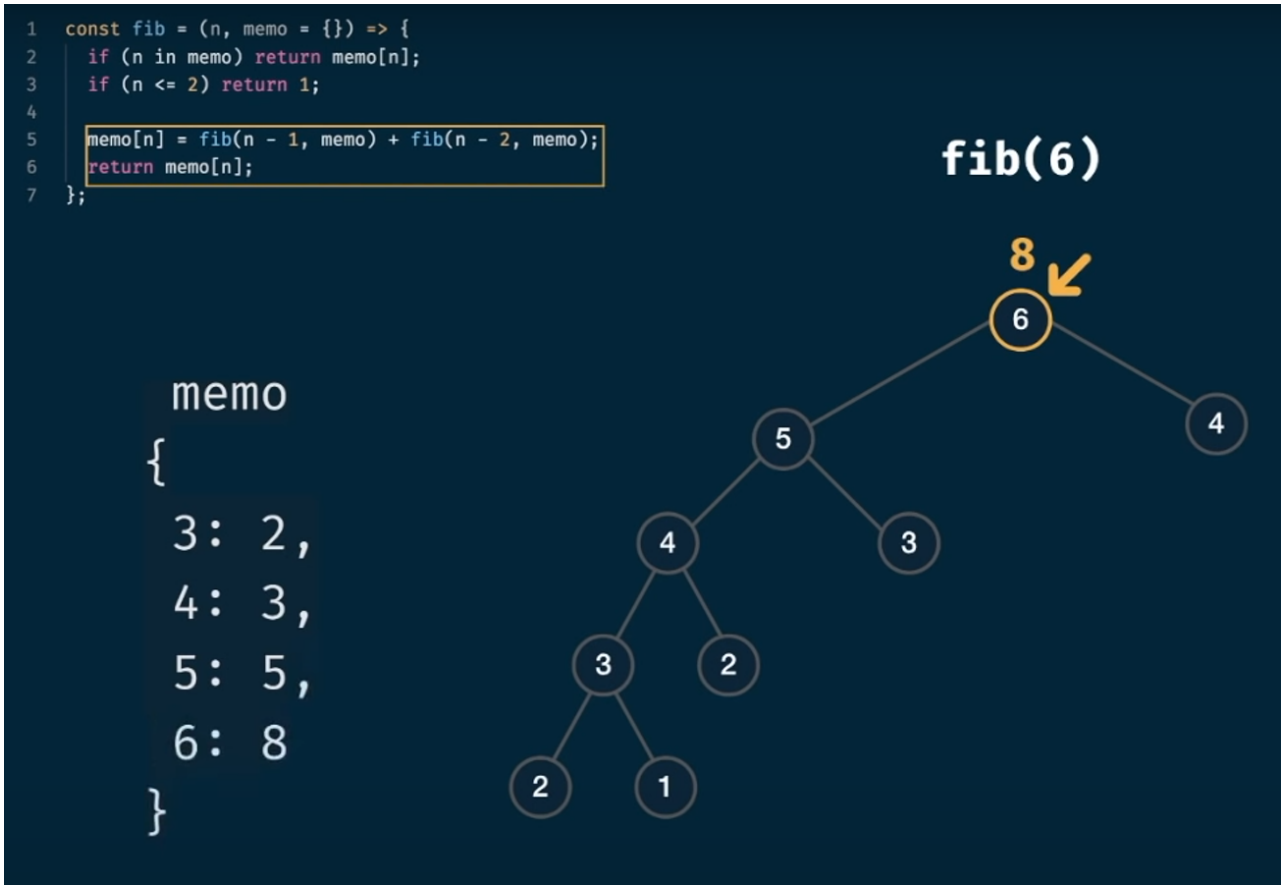
console.log(fib(30));
```

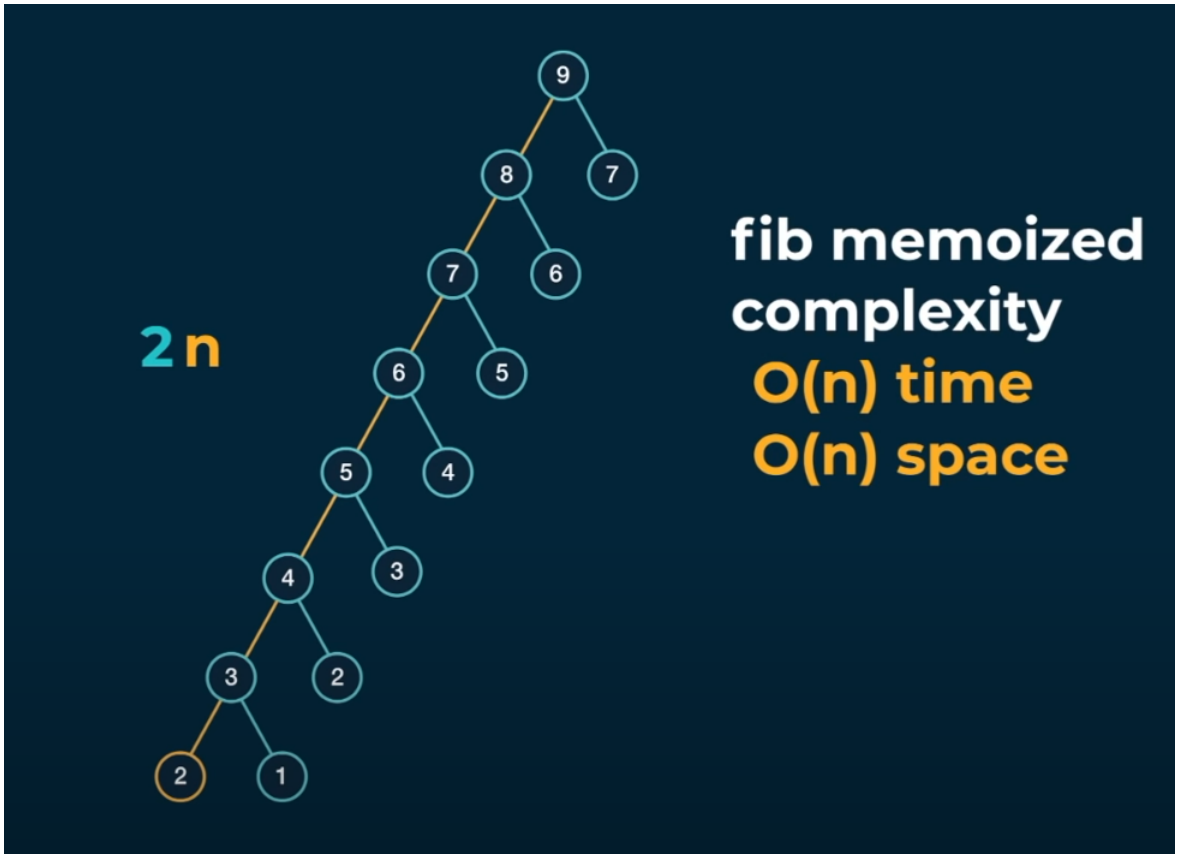
Memoized solution recursion tree

1. Without Memorization



2. With memoization

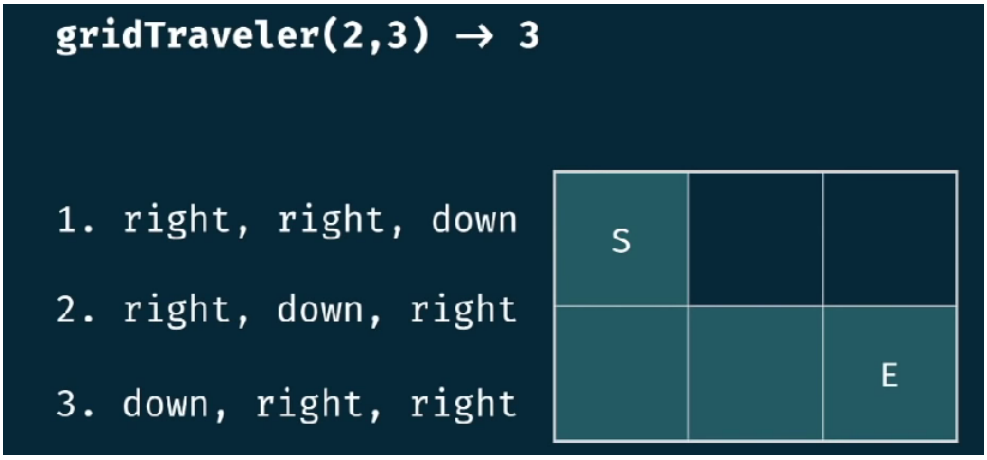


- Overall we have roughly `2n` nodes
- 

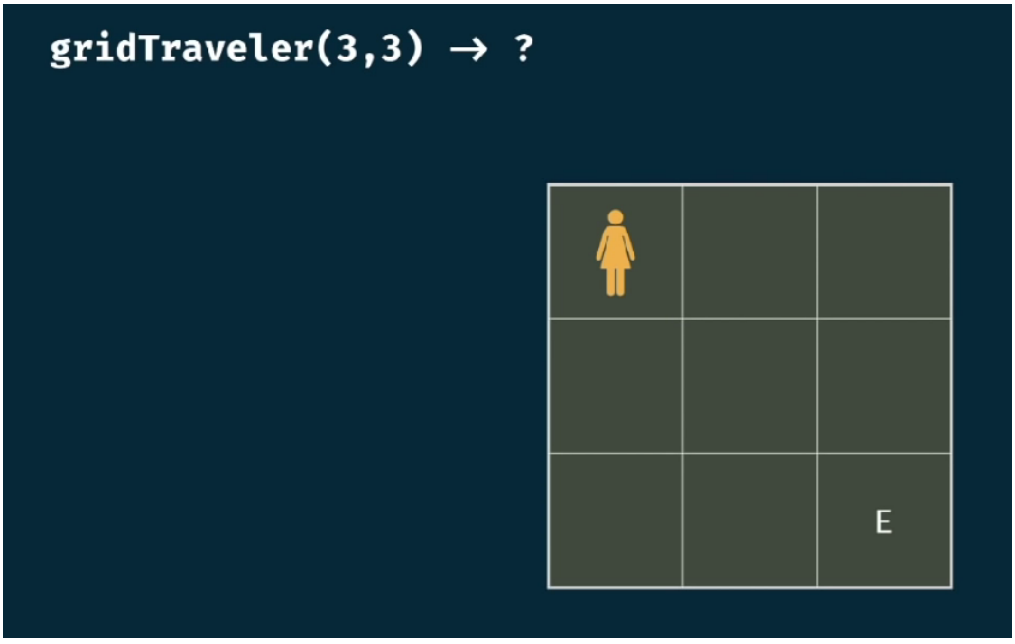
- `Fib(9)` has this structure
- New time complexity is `O(n)` and space complexity is `O(n)`

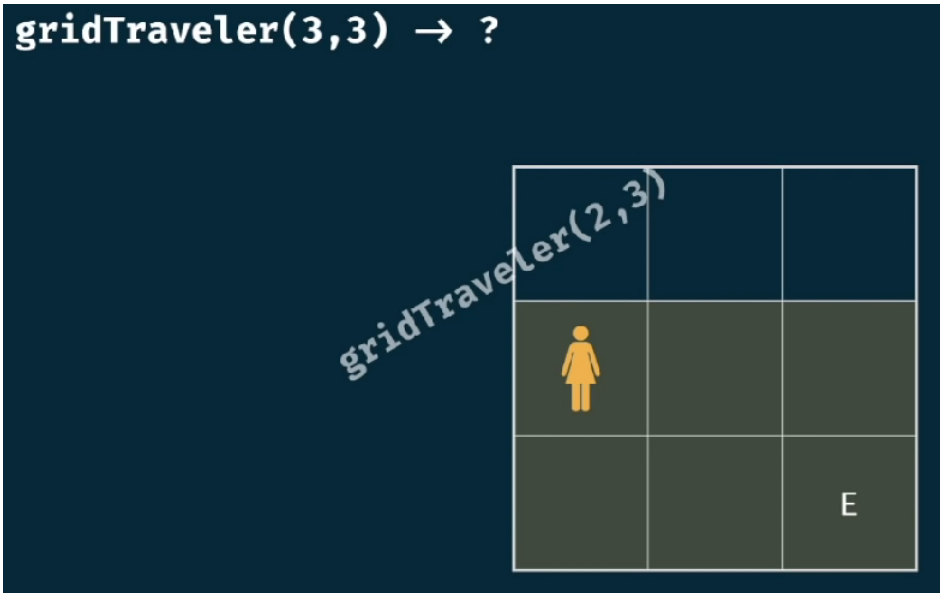
Grid Traveller

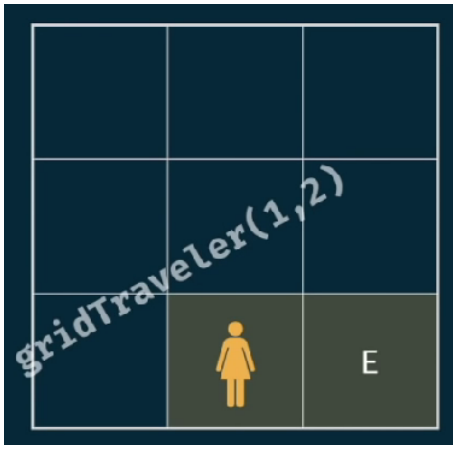
- You are a traveller on a 2D Grid. You begin in the top-left corner and your goal is to travel to the bottom right corner. You may only move down or right.
- In how many ways can you travel to the goal on a grid with dimensions m*n?
-

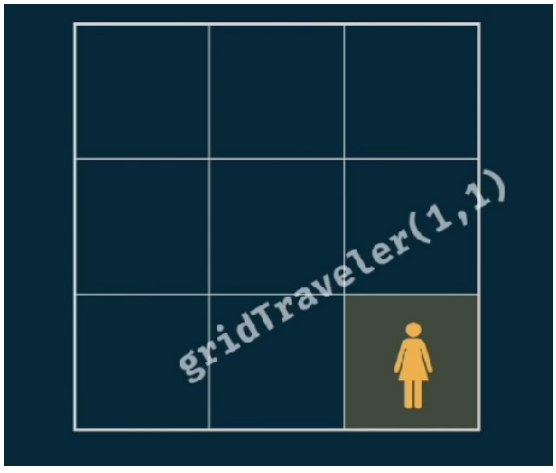


- Problems like this.
  - Always start with the smallest possible scenario
    - `gridTraverler(1,1) -> 1` since start is already at end
      - do nothing
    - `gridTraveler(0,1)` -> 0
      - invalid
    - `gridTraveler(3,3)`
      -
  - These can be thought of as base cases and can be used to construct the larger solution
- for a `gridTraveler(3,3)`
  - As me move down the problem reduces to `gridTraveler(2,3)`
-

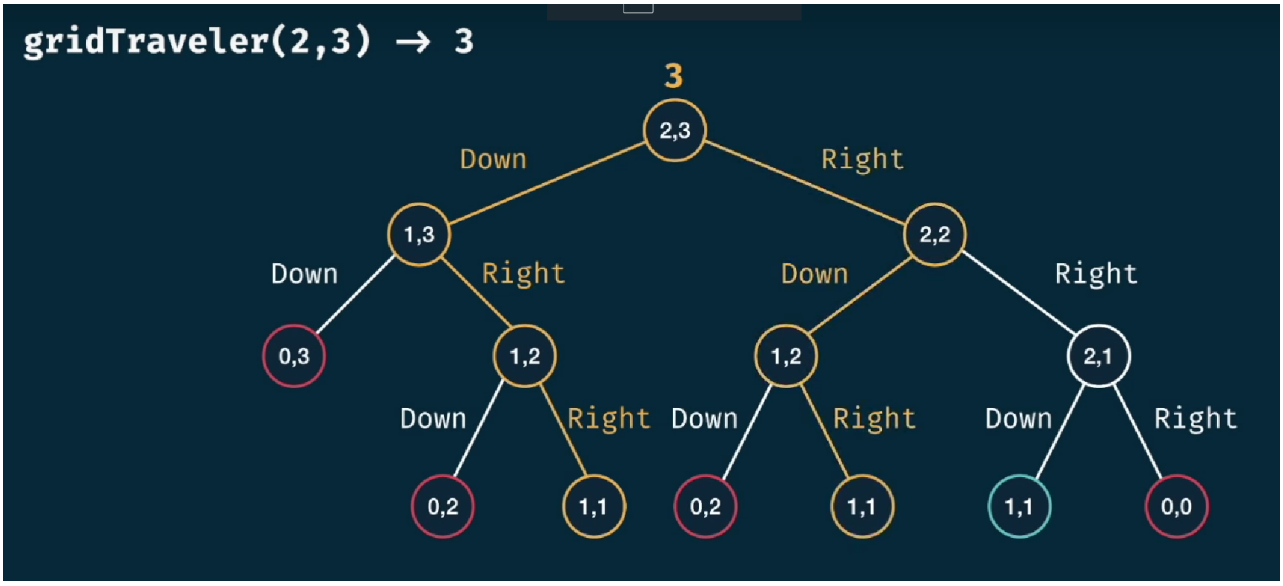


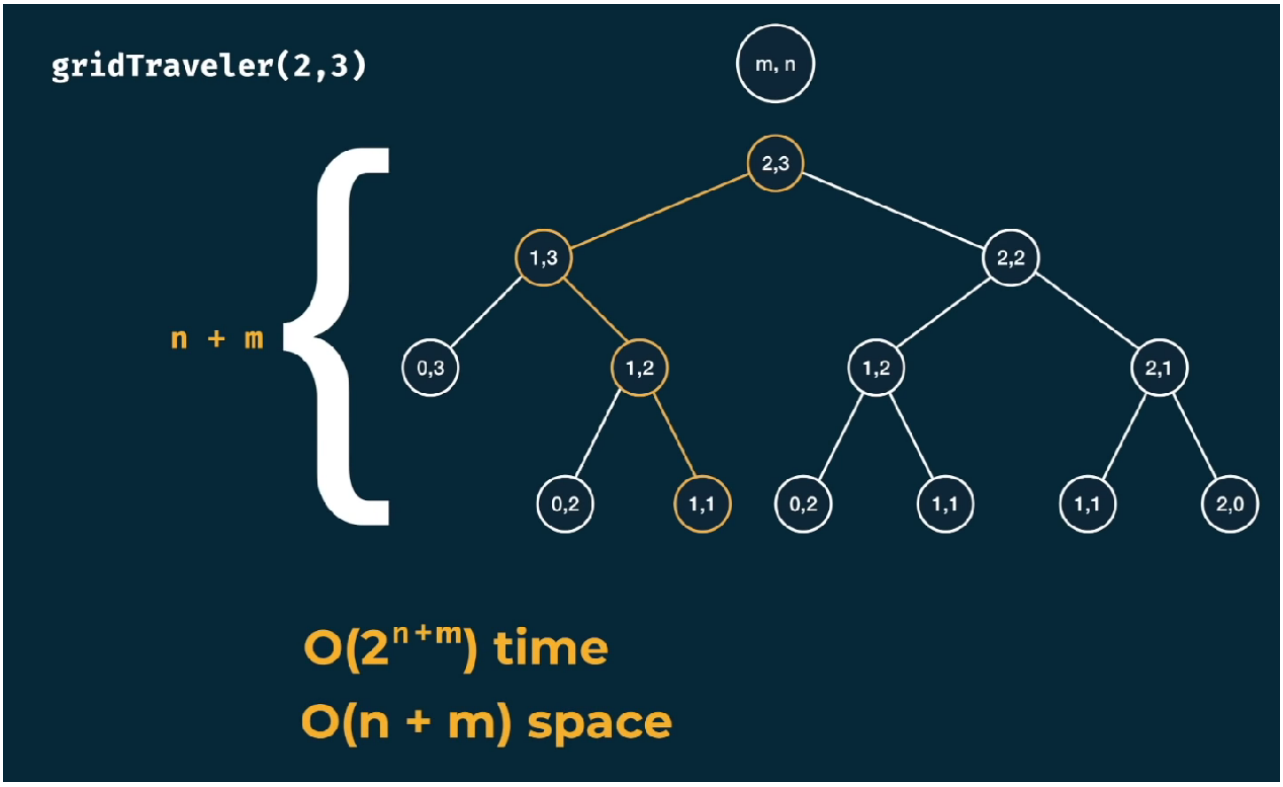
- 

- 

- 

- Any problem that looks like recursion always visualize a tree
- At every recursive call we are either reducing by 1 row or by 1 column
- From the tree structure we can already tell which tree nodes were formed by which combination of moves.
- which combination gives a positive answer and which moves lead us to dead end
-



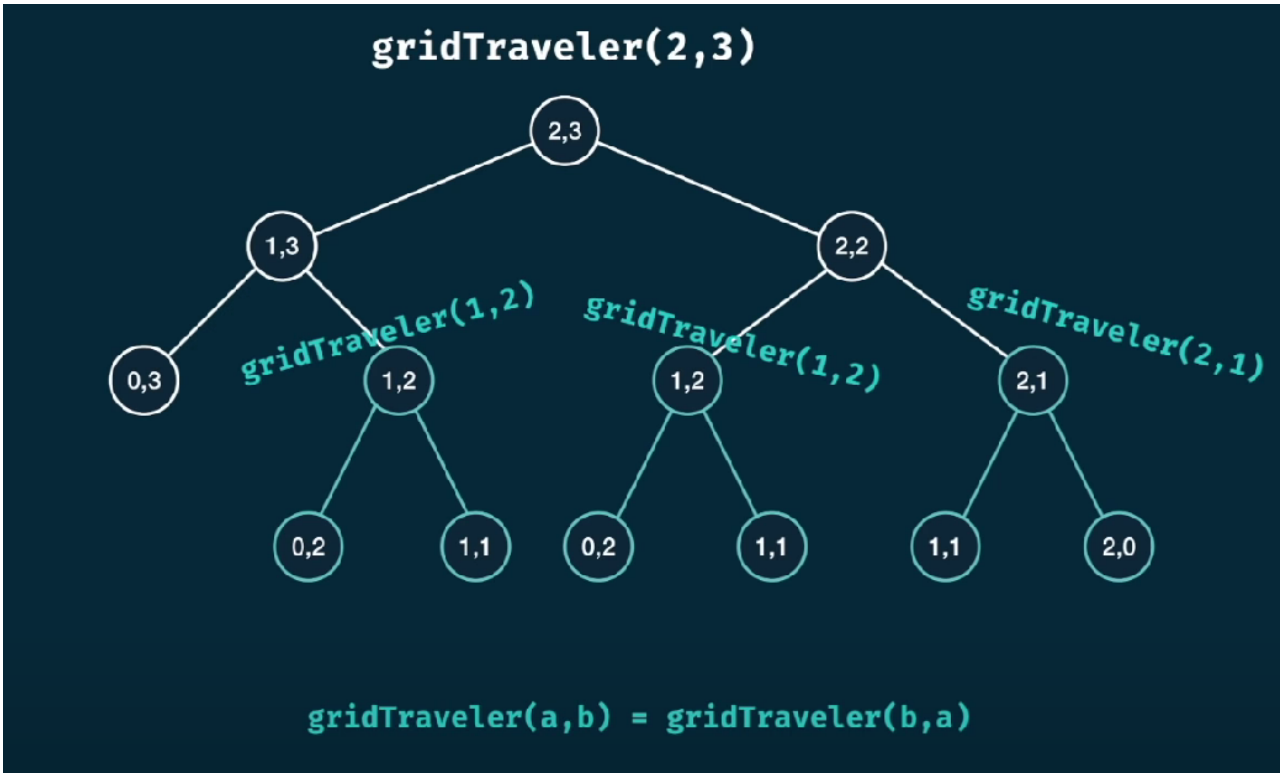
- The tree looks like a binary tree because at every point there are two possible ways to traverse the the grid either down or right
- This tree is now composed of two different factors : `m` and `n`.
- We must take in account the other parameter
- The height of the tree
  - To do that we travel furthest and see the base case
  - There are two base cases
    - either one of them is 0 and we return
    - Or both are 1 and we count as 1
  - There are `m+n` levels
- 

Brute force recursive solution

```
const gridTraveller = (m,n) =>{
    if(m === 1 && n === 1) return 1;
    if(m === 0 || n === 0) return 0;

    return gridTraveller(m-1, n) + gridTraveller(m, n-1);
}
```

Making improvements to counting number of ways we can reduce the recursive

- We can notice there are duplicate subtrees
- `2,1` is same as `1,2`
- 

Strategy

- To solve these recursive dynamic programming problems we need a brute force solution to the original recursive problem
- A well formed recursion

For memoization

- We take the exact return recursive statement and assign it to the `memo[key]`
- and return that value based key

Solution

```
const gridTraveller = (m,n, memo={}) =>{
    let key = m + ','+n;
    if(key in memo) return memo[key];
    if(m === 1 && n === 1) return 1;
    if(m === 0 || n === 0) return 0;

    memo[key] = gridTraveller(m-1, n, memo) + gridTraveller(m, n-1, memo);
    return memo[key];
}
```

- A pair cannot be used as key in unordered_list
- ```
  // wrong solution
  int gridTraveller(int n, int m){
      // create memo table
      static unordered_map<pair<int, int>, int> memo;
      // find the pair
      auto it = memo.find({n, m});
      if(it != memo.end()){
          return memo[{n,m}];
      }

      if(m == 1 && n == 1) return 1;

      if(m == 0 || n == 1) return 0;

      memo[{n,m}] = gridTraveller(m-1, n) + gridTraveller(m, n-1);
  }
  ```

```
int gridTraveller(int n, int m){
    // crate a key
    string key = to_string(n) + " , "+ to_string(m);
    // create memo table
    static unordered_map<string, int> memo;
    // find the pair
    auto it = memo.find(key);
    if(it != memo.end()){
        return memo[key];
    }

    if(m == 1 && n == 1) return 1;

    if(m == 0 || n == 0) return 0;

    memo[key] = gridTraveller(m-1, n) + gridTraveller(m, n-1);
    return memo[key];
```
```

```
}
```

Space and Time complexity

- For a case of `gridTraveler(4,3)`
  - `m: {0,1,2,3,4}`
  - `n: {0,1,2,3}`
- Total number of nodes possible `m*n`
- Memoized version has time and space complexity of
  - Time : `O(m*n)`
  - Space: `O(n+m)`

Memoization Recipe

Two Step process

1. **Make it work**
   1. *Visualize the problem as a tree*
   2. *Implement the tree using recursion (brute force)*
   3. *Test it (should give correct results, it may be slow)*
2. **Make it efficient**
   1. *Add a memo object (key represent argument to the function, values represent return values), it must be shared among all recursive calls*
   2. *Add a base case to return memo values*
   3. *Store return values into the memo*

Don't try to implement an efficient algorithm from get go. Get a brute force working solution using recursion then implement using memoization.

- first look for correctness in your solution it could be brute force or anything. As long as it can get you correct results
- Then once that is achieved then try to optimize it.
- Leaves of the tree are the base cases

canSum

- Write a function `canSum(targetSum, numbers)` that takes in a targetSum and an array of numbers as an arguments.
- The function should return a boolean indicating whether or not it is possible to generate the targetsum using numbers from the array
- You can use an element of the array as many times as needed
- You may assume that all input numbers are nonnegative

```
bool canSum(int target, vector<int> & v){
    cout << "canSum(" << target << ")" << endl;
    if(target == 0){
```

```
        return true;
    }

    if(target < 0){
        return false;
    }

    for(auto a: v){
        int remainder = target -a;
        if(canSum(remainder, v)){
            return true;
        }
    }

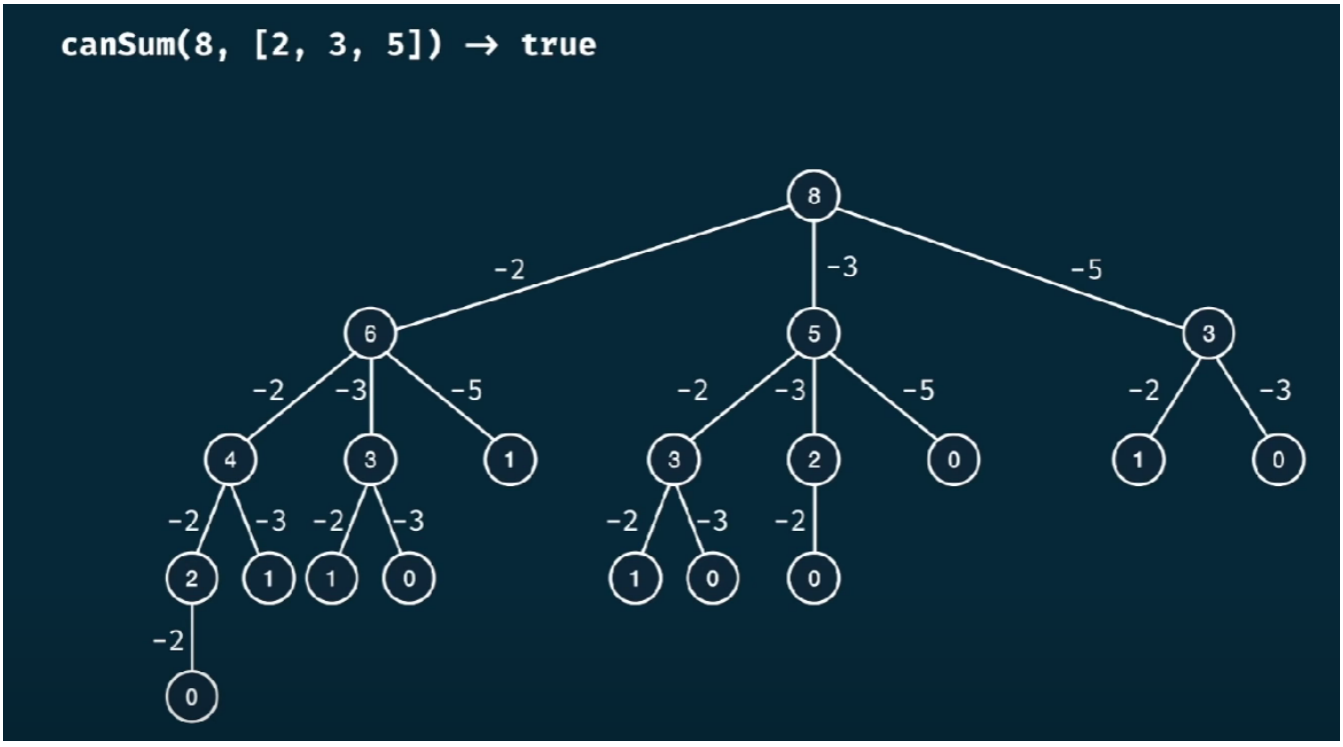
    return false;
}
```

```
const canSum(targetSum, numbers){
    if(targetSum === 0) return true;
    if(targetSum < 0 ) return false;

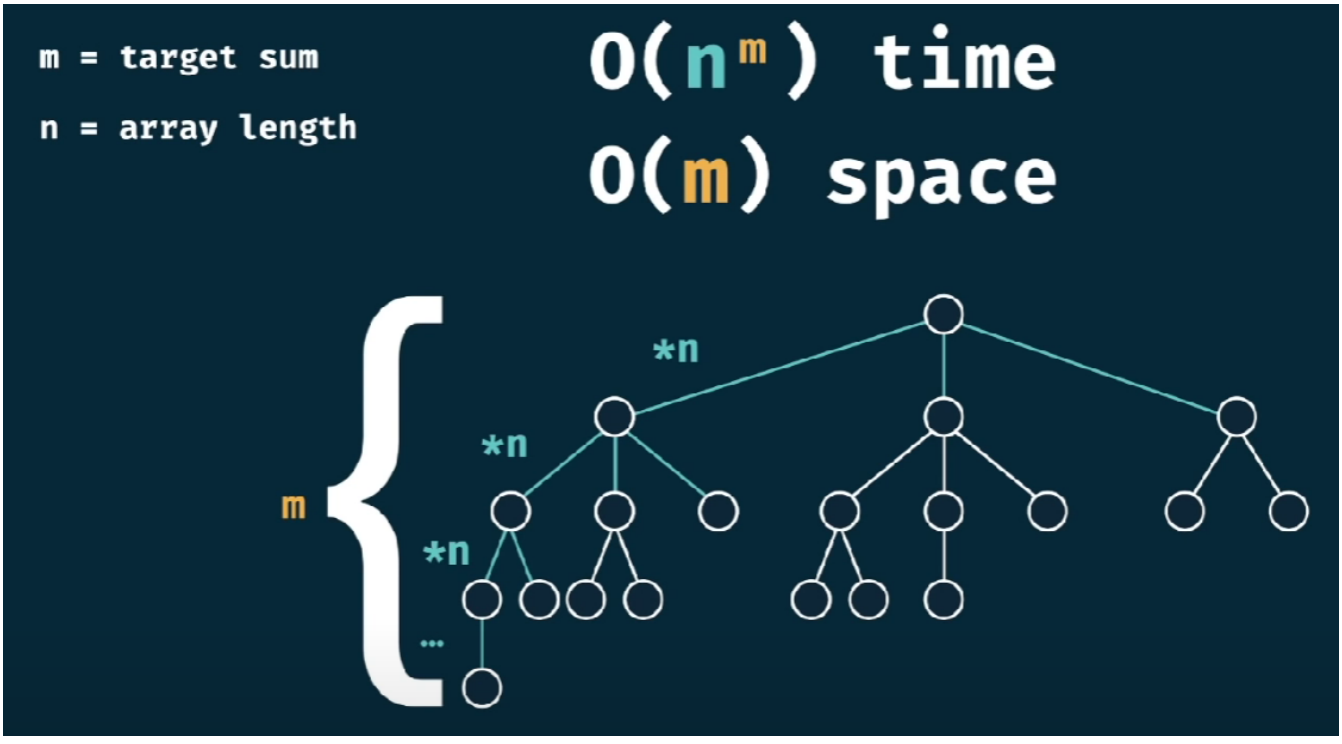
    for(let num of numbers){
        const remainder = targetSum - num;
        if(canSum(remainder, numbers) === true){
            return true;
        }
    }

    return false;
}
```

Time and space complexity of brute force



- Two factors determine the size of the tree
  - `m` = target sum
  - `n` = array length
- Height of tree
  - Even if we subtract 1 at every step to reach the target sum the maximum height would be targetsum
  - `m` is the height of the tree
- Branches
  - The maximum branching factor is the number of elements in the array: `n`



- The time complexity becomes `O(n^m)`
- Space used will be maximum height of the tree `O(m)`

Memoized solution

```
bool canSum(int target, vector<int> & v){

    static unordered_map<int,bool> memo;

    unordered_map<int, bool>::iterator it = memo.find(target);

    if(it == memo.end()){
        return memo[target];
    }

    if(target == 0){
        return true;
    }

    if(target < 0){
        return false;
    }

    for(auto a: v){
        int remainder = target -a;
        memo[remainder] = canSum(remainder, v);
        if(memo[remainder]){
            return true;
        }
    }

    memo[target] = false;
    return false;
}
```

```
const canSum = (target, numbers) =>{
    if(target == 0) return true;
    if(target < 0) return false;

    for( let num of numbers){
        const remainder = target - num;
        if(canSum(remainder, numbers) === true){
            return true;
        }
    }

    return false;
}
```

```
const canSum = (target, numbers, memo = {}) =>{
```

```
    if(target in memo) return memo[target];

    if(target == 0) return true;
    if(target < 0) return false;

    for( let num of numbers){
        const remainder = target - num;
        memo[remainder] = canSum(remainder, numbers, memo);

        if(memo[remainder]){
            return true;
        }

    }

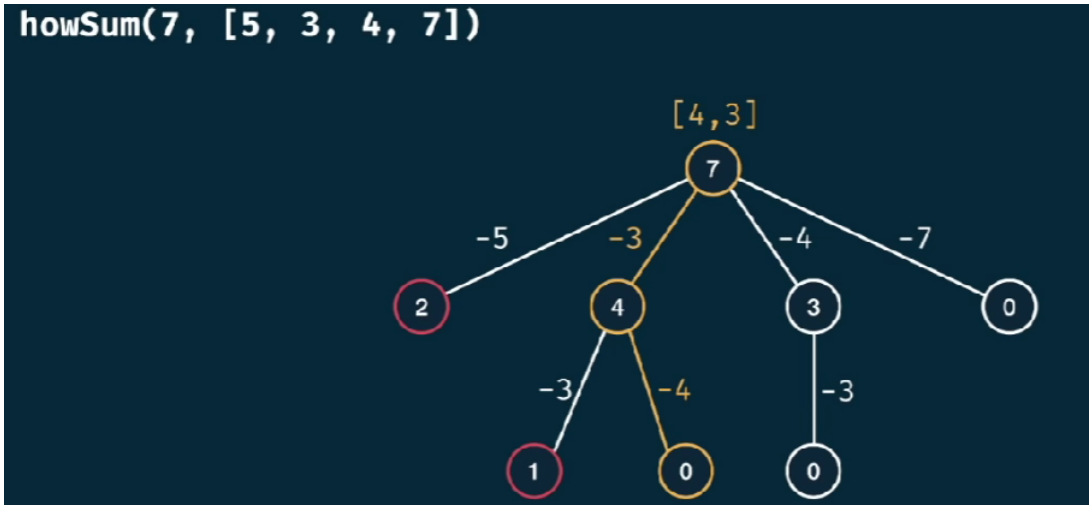
    memo[target] = false;
    return memo[target];
}
```

Time and space complexity of memoized solution

- `m` the target sum is the height of the array
- `n` the size of the array determines the branches
- Time complexity if `O(n*m)`
    - We still need to branch `n` times to collect the values for the memo object
- Space complexity is height `O(m)`

howSum(targetSum, numbers)

- 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.
- We can return as soon as we find a combination
- Example
    - a
        - `howSum(8, [2,3,5]) -> [2,2,2,2]`
        - `howSum(8, [2,3,5]) -> [3,5]`
    - b
        - `howSum(7, [2,4]) -> null`
    - c

- `howSum(0, [1,2,3])-> []`
- 

Recursive solution

```
const howSum = (target, numbers) =>{

    if(target === 0){
        return [];
    }

    if(target < 0){
        return null;
    }

    for(let c of numbers){
        const remainder = target - c;
        const result = howSum(remainder, numbers);
        if(result !== null){
            return [ ...result, c];
        }
    }

    return null;

}

console.log(howSum(7, [2,3]));
```

```
// c++ 17 and ownwards
// g++ -std=c++17 index.cpp -o index
#include <iostream>
#include <unordered_map>
#include <algorithm>
#include <vector>
#include <utility>
#include <string>
```

```cpp
#include <optional>

using namespace std;


optional<vector<int>> howSum(int sum, vector<int> &v){
    if(sum < 0){
        return nullopt;
    }

    if(sum == 0){
        return vector<int>();
    }

    for(auto a: v){
        int remainder = sum -a;
        optional<vector<int>> result = howSum(remainder, v);
        if(result.has_value()){
            result->push_back(a);
            return *result;
        }
    }

    return nullopt;

}


int main(){

    vector<int> v = {2, 3};

    optional<vector<int>> v2 = howSum(7, v);
    if(v2.has_value()){
        for(auto a: *v2){
            cout << a;
        }
    }
    return 0;
}
```

- You can use `optional<vector<int>>` to either return a `vector<int>` or return a `nullopt` which represents a null value.
- `optional<vector<int>> v2` can be checked for if there is a value by using `.has_value()`

space and time complexity for howSum

- `m` = target sum
- `n` = length of vector
- `time` = `O(n^m *m)`

    - Target gum determines the height of the tree
    - vector length determine number of branches at every height
- Space complexity
    - `m` is the number of max stack size at any point
    - space = `O(m)`

memoized solution

```
const howSum = (target, numbers, memo = {}) =>{

    if(target in memo){
        return memo[target];
    }

    if(target === 0){
        return [];
    }

    if(target < 0){
        return null;
    }

    for(let c of numbers){
        const remainder = target - c;
        const result = howSum(remainder, numbers, memo);
        if(result !== null){
            memo[target] = [ ...result, c];
            return memo[target];
        }
    }

    memo[target] = null;
    return null;
}
```

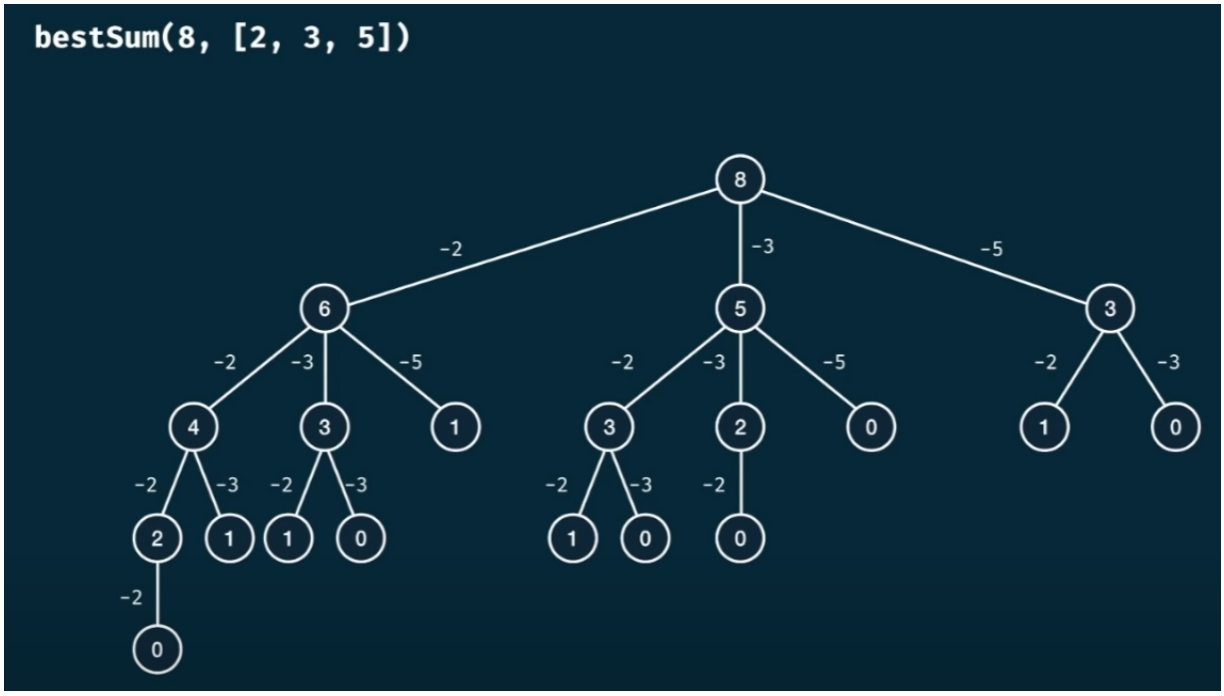
```

optional<vector<int>> howSum(int sum, vector<int> &v){

    static unordered_map<int, optional<vector<int>>> memo;

    auto it = memo.find(sum);

    if(it != memo.end()){
        return it->second;
    }

    if(sum < 0){
        return nullopt;
    }
```

```
    if(sum == 0){
        return vector<int>();
    }

    for(auto a: v){
        int remainder = sum -a;
        optional<vector<int>> result = howSum(remainder, v);
        if(result.has_value()){
            result->push_back(a);
            memo[sum] =  *result;
            return memo[sum];
        }
    }

    memo[sum] = nullopt;
    return nullopt;

}


int main(){

    vector<int> v = {15, 2};

    optional<vector<int>> v2 = howSum(3000, v);
    if(v2.has_value()){
        for(auto a: *v2){
            cout << a << " ";
        }
    }
    return 0;
}
```

memoized version time and space

- Time
  - Tree height is `n`
  - `m` is the vector size
  - copying the array with cost `m`
  - So total time is `O(n*m*m)`
  - Time: `O(n*m^2)`
- Space
  - We must consider maximum number of recursive call at any point: `m`
  - And all unique values of the memo. The maximum length is `m`
  - Space `O(m^2)`

bestSum(targetSum, numbers)

- Write a function `bestSum(targetSum, numbers)` that takes in a targetSum and an array of numbers as arguments.
- The function should return an array containing the shortest combination of numbers that add up to exactly the targetSum
- If there is a tie for the shortest combination, you may return any one of the shortest.
- Eg
  - `bestSum(7, [5,3,4,7])`
    - `[3,4]`
    - `[7]` result
  - `bestSum(8, [2,3,5])`
    - `[2,2,2,2]`
    - `[2,3,2]`
    - `[3,5]`
  - 

- **Recursive solution**

```
const bestSum = (targetSum, numbers) =>{
    if(targetSum === 0){
        return [];
    }

    if(targetSum < 0){
        return null;
    }

    let lastResult = null;

    for(let num of numbers){
        let remainder = targetSum - num;
```

```
        let result = bestSum(remainder, numbers);
        if( result !== null){
            let current = [...result, num];
            if(lastResult === null || current.length < lastResult.length){
                lastResult = current;
            }
        }
    }

    return lastResult;
}
```

```
optional<vector<int>> bestSum(int targetSum, vector<int>& numbers){
    if(targetSum == 0){
      return vector<int>();
    }

    if(targetSum < 0){
      return nullopt;
    }

    optional<vector<int>> minimumLength = {}; // nullopt

    for(auto a: numbers){

      int remainder = targetSum -a;

      optional<vector<int>> result = bestSum(remainder, numbers);

      if(result.has_value()){
        result->push_back(a);
        if(minimumLength == nullopt || result->size() < minimumLength->size()){
          // * minimumLength = * result; does not work
          minimumLength = result;  // only this works
        }
      }
    }

    //printOpt(minimumLength);
    return minimumLength;
}
```

Time and space complexity

- Space
  - height of tree is `m` targetsum
  - for every recursive call we are making an array of size `m`
  - Space : `O(m)`

- Time complexity
  - Height of tree is `m`
  - Array size: `n`
  - For every call we are iteration through an array of size `m` because it is the size of the array we are returning so if any number is 1 then we could have an array of just 1 `m` times
  - `O(n^m * m)`

Memoized solution

```
const bestSum = (targetSum, numbers, memo = {}) =>{

    if(targetSum in memo){
        return memo[targetSum];
    }

    if(targetSum === 0){
        return [];
    }

    if(targetSum < 0){
        return null;
    }

    let lastResult = null;

    for(let num of numbers){
        let remainder = targetSum - num;

        let result = bestSum(remainder, numbers, memo);
        if( result !== null){
            let current = [...result, num];
            if(lastResult === null || current.length < lastResult.length){
                lastResult = current;
            }
        }
    }

    memo[targetSum] = lastResult;
    return lastResult;
}

console.log(bestSum(100, [1,2,5,25]));
console.log(bestSum(7, [5, 3,4,7]));
```

```
optional<vector<int>> bestSum(int targetSum, vector<int>& numbers){

    static unordered_map<int, optional<vector<int>>> memo;

    unordered_map<int, optional<vector<int>>>::iterator it = memo.find(targetSum);
```

```
    if(it != memo.end()){
        return memo[targetSum];
    }

    if(targetSum == 0){
      return vector<int>();
    }

    if(targetSum < 0){
      return nullopt;
    }

    optional<vector<int>> minimumLength = {}; // nullopt

    for(auto a: numbers){

      int remainder = targetSum -a;

      optional<vector<int>> result = bestSum(remainder, numbers);

      if(result.has_value()){
        result->push_back(a);
        if(minimumLength == nullopt || result->size() < minimumLength->size()){
          // * minimumLength = * result; does not work
          minimumLength = result;  // only this works
        }
      }
    }

    memo[targetSum] = minimumLength;
    return minimumLength;
}
```

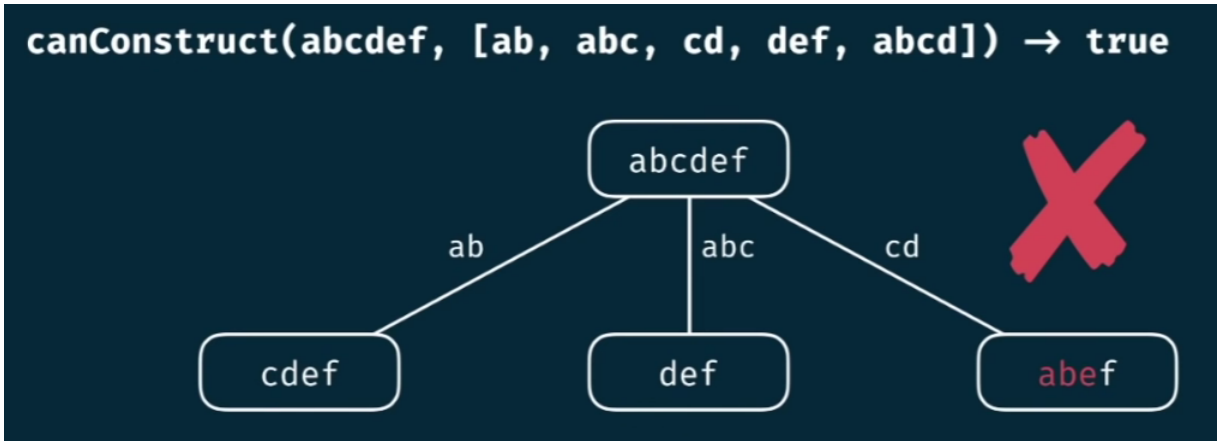
Time and space complexity

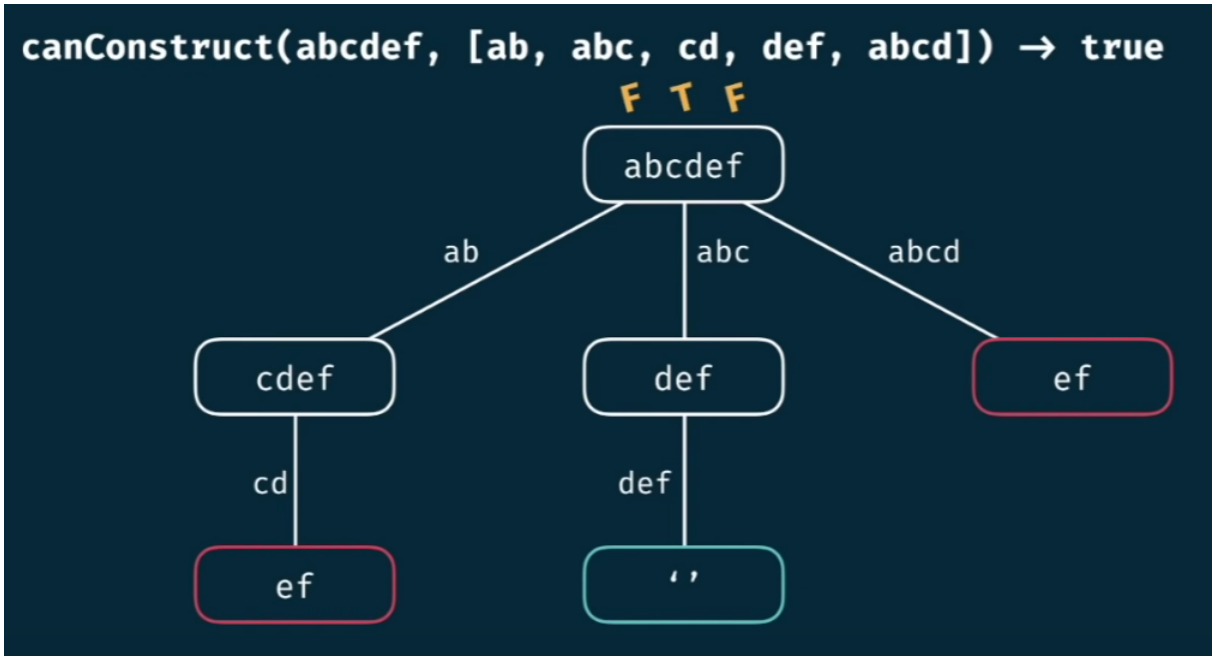
- Time complexity `O(m*n)`
- Complexity: `O(m^2 *n)`

canSum vs howSum vs bestSum

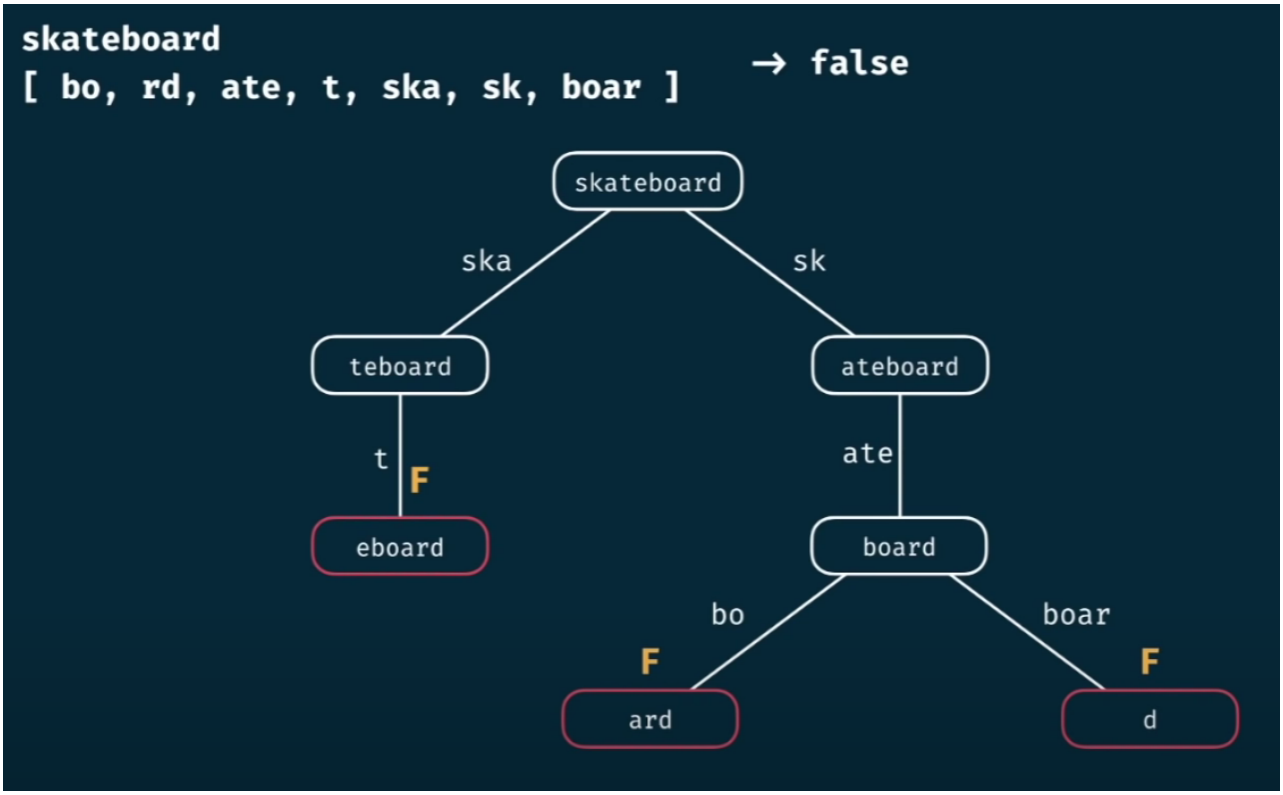
- canSum -> Decision Problem
- howSum -> Combinatoric Problem
- bestSum -> Optimation Problem

canConstruct

- Write a function `canConstruct(target, wordBank)` that accepts a target string and an array of strings
- The function should return a boolean indicating whether or not the `target` can be constructed by concatenating elements of the `wordBank` array
- You may resuse element of `wordBank` as many times as needed
- `canConstruct(abcdef, [ab, abc, cd, def, abcd]) -> true`
  - `abc+def`
- base case
  - `canConstruct('', [cat, dog, mouse]) ->true`
- Do Not take from the middle only from front or back
- 

- 

-


skateboard
[bo, rd, ate, t, ska, sk, boar]
→ false
skateboard
ska
sk
teboard
ateboard
t
F
ate
eboard
board
bo
boar
F
F
ard
d

-