**How to understand and approach a problem:**1. Identify if you can break down the problem into smaller sub problem.

2. Write the recurrence relation if needed.

3. Draw the recursion tree.

4. About the tree:

\* see the flow of functions, how they are getting in stack.

\*Identify and focus on left tree calls and right tree calls.

\*Draw the recursion tree and pointers again and again using pen & paper.

\*Use a debugger to see the flow of the program.

5. See how the values are returned at each step , see where the function call will come out , and in the end , it will come out of the main function.

**Working with Variables:**

1. Arguments.

2.Return type.

3.Body of function.

**Types of recurrence relation:**

1. Linear recurrence relation. (fibbo)

2.Divide & Conquer recurrence relation. (binary search)

**Tip:**

1 .make sure to return the result of a function call of the return type . (if there is a return type , then return for each function call the result of the return data type).

**Subsequences:**

**Print all subsequences:**

**[]**

**/ \**

**[1] []**

**/ \ / \**

**[1,2] [1] [2] []**

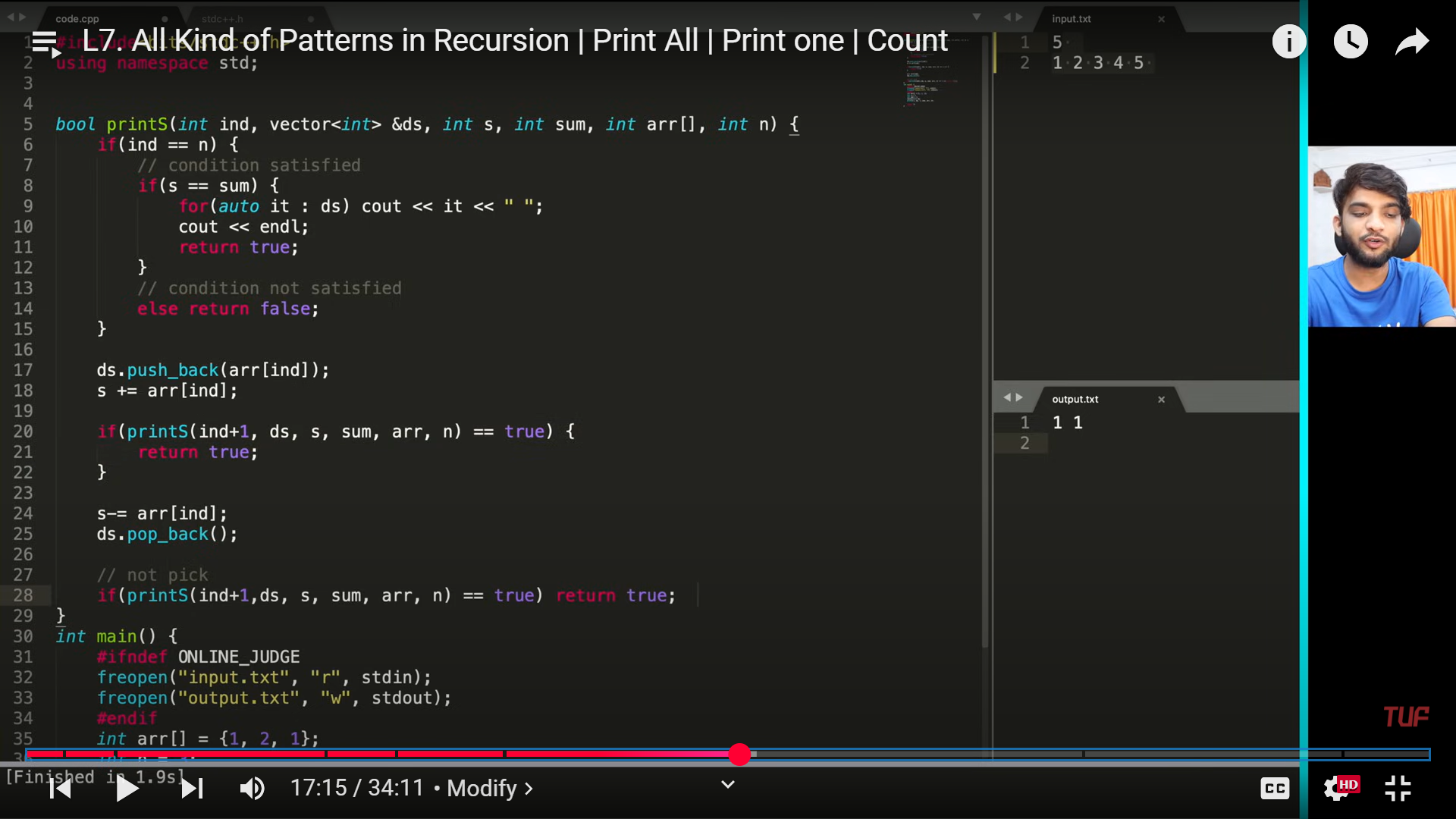
**/ \ / \ / \ / \**

**[1,2,3] [1,2] [1,3] [1] [2,3] [2] [3] []**

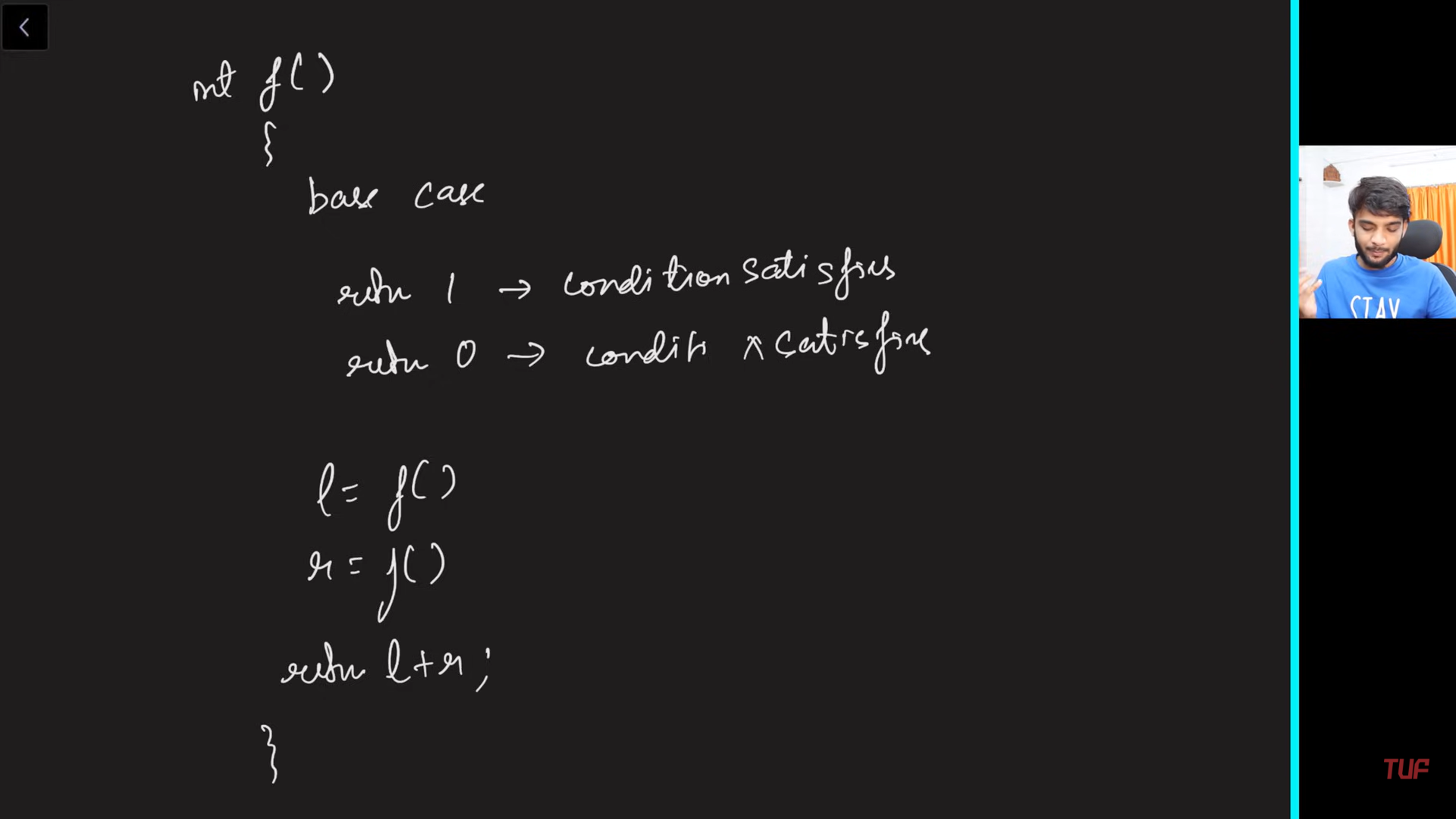
public List<List<Integer>> subsets(int[] nums) {  
 List<List<Integer>> ans = new ArrayList();  
 func(0,ans,new ArrayList<>(),nums,nums.length);  
 return ans;  
}  
public void func(int ind,List<List<Integer>> ans,List<Integer> temp,int[] nums,int n)  
{  
 if(ind==n)  
 {  
 ans.add(new ArrayList<>(temp));  
 return;  
 }  
 temp.add(nums[ind]);  
 func(ind+1,ans,temp,nums,n);  
 temp.remove(temp.size()-1);  
 func(ind+1,ans,temp,nums,n);  
}

* **Time Complexity: O(2^n)** (since we generate all 2^n subsets).
* **Space Complexity: O(n)** (max recursive call depth).

**Technique to print one answer:**



**Technique to count the subsequences:**



**Combination:**

**Pattern 1:**

**Pick or not pick.**

**Combination sum – 1:**

Problem Statement:

Given an array of distinct integers and a target, you have to return the list of all unique combinations where the chosen numbers sum to target. You may return the combinations in any order.

The same number may be chosen from the given array an unlimited number of times. Two combinations are unique if the frequency of at least one of the chosen numbers is different.

It is guaranteed that the number of unique combinations that sum up to target is less than 150 combinations for the given input.

Examples:

Example 1:

Input: array = [2,3,6,7], target = 7

Output: [[2,2,3],[7]]

Explanation: 2 and 3 are candidates, and 2 + 2 + 3 = 7. Note that 2 can be used multiple times.

7 is a candidate, and 7 = 7.

These are the only two combinations.

Example 2:

Input: array = [2], target = 1

Output: []

Explaination: No combination is possible.

The given **Java** code uses **backtracking** to find all unique combinations where numbers from the candidates array sum up to target. It allows **repeated usage** of the same number, meaning an element can be picked multiple times.

### **Key Idea (Backtracking + Recursion)**

1. Start from index 0 with an empty list temp.
2. At each index ind, we have **two choices**:
   * **Include** candidates[ind] (since repetition is allowed, we stay at the same index).
   * **Skip** candidates[ind] and move to the next index.
3. When target == 0, add the valid combination to the result list ans.
4. If ind == n (end of array), return without adding anything.
5. **Backtrack** (undo the last choice) after each recursive call.

## ****Step-by-Step Execution****

Let's take an example:

java

CopyEdit

int[] candidates = {2, 3, 6, 7};

int target = 7;

### **Recursive Calls Visualization**

scss

CopyEdit

func(0, 4, 7, [2,3,6,7], ans, [])

├── Include 2 → func(0, 4, 5, [2,3,6,7], ans, [2])

│ ├── Include 2 → func(0, 4, 3, [2,3,6,7], ans, [2,2])

│ │ ├── Include 2 → func(0, 4, 1, [2,3,6,7], ans, [2,2,2])

│ │ │ ├── Include 2 (target -1 ❌) → Backtrack

│ │ │ ├── Skip to 3 → func(1, 4, 1, [2,3,6,7], ans, [2,2,2]) (3 > 1 ❌)

│ │ ├── Include 3 (target 0 ✅) → Add [2,2,3] to ans

│ ├── Skip to 3 → func(1, 4, 3, [2,3,6,7], ans, [2,2])

│ │ ├── Include 3 → func(1, 4, 0, [2,3,6,7], ans, [2,3])

│ │ │ ├── (target == 0 ✅) → Add [2,3] to ans

│ │ ├── Skip to 6 → func(2, 4, 3, [2,3,6,7], ans, [2])

├── Skip to 3 → func(1, 4, 7, [2,3,6,7], ans, [])

│ ├── Include 3 → func(1, 4, 4, [2,3,6,7], ans, [3])

│ │ ├── Include 3 → func(1, 4, 1, [2,3,6,7], ans, [3,3])

│ │ ├── Skip to 6 → func(2, 4, 4, [2,3,6,7], ans, [3])

│ ├── Skip to 6 → func(2, 4, 7, [2,3,6,7], ans, [])

│ │ ├── Include 6 → func(2, 4, 1, [2,3,6,7], ans, [6])

│ │ ├── Skip to 7 → func(3, 4, 7, [2,3,6,7], ans, [])

│ │ │ ├── Include 7 (target 0 ✅) → Add [7] to ans

✅ **Valid combinations stored in ans:** [[2,2,3], [7]]

public List<List<Integer>> combinationSum(int[] candidates, int target) {  
 List<List<Integer>> ans = new ArrayList<>();  
 func(0,candidates.length,target,candidates,ans,new ArrayList<>());  
 return ans;  
}  
  
public void func(int ind,int n,int target,int[] candidates,List<List<Integer>> ans,List<Integer> temp)  
{  
 if(ind == n)  
 {  
 if(target == 0)  
 ans.add(new ArrayList<>(temp));  
 return;  
 }  
  
 if(candidates[ind] <= target)  
 {  
 temp.add(candidates[ind]);  
 func(ind,n,target-candidates[ind],candidates,ans,temp);  
 temp.remove(temp.size()-1);  
 }  
 func(ind+1,n,target,candidates,ans,temp);  
}

**Time Complexity Analysis**

* **Worst Case:** **O(2^t)** (where t is the target)
  + Since each number can be included multiple times, the recursion explores all possible ways.
* **Space Complexity:** **O(target / min(candidate))**
  + The depth of recursion is limited by the minimum candidate value.

**Pattern 2:**

**Pick from the index till the end.**

**Combination sum – 2:**

Problem Statement: Given a collection of candidate numbers (candidates) and a target number (target), find all unique combinations in candidates where the candidate numbers sum to target. Each number in candidates may only be used once in the combination.

Note: The solution set must not contain duplicate combinations.

Examples:

Example 1:

Input: candidates = [10,1,2,7,6,1,5], target = 8

Output:

[

[1,1,6],

[1,2,5],

[1,7],

[2,6]]

Explanation: These are the unique combinations whose sum is equal to target.

Example 2:

Input: candidates = [2,5,2,1,2], target = 5

Output: [[1,2,2],[5]]

Explanation: These are the unique combinations whose sum is equal to target.

Before starting the recursive call make sure to sort the elements because the ans should contain the combinations in sorted order and should not be repeated.

Initially, We start with the index 0, At index 0 we have n - 1 way to pick the first element of our subsequence.

Check if the current index value can be added to our ds. If yes add it to the ds and move the index by 1. while moving the index skip the consecutive repeated elements because they will form duplicate sequences.

Reduce the target by arr[i],call the recursive call for f(idx + 1,target - 1,ds,ans) after the call make sure to pop the element from the ds.(By seeing the example recursive You will understand).

if(arr[i] > target) then terminate the recursive call because there is no use to check as the array is sorted in the next recursive call the index will be moving by 1 all the elements to its right will be in increasing order.

Base Condition:

Whenever the target value is zero add the ds to the ans return.

public List<List<Integer>> combinationSum2(int[] candidates, int target) {  
 List<List<Integer>> ans = new ArrayList<>();  
 Arrays.sort(candidates);  
 func(0,target,candidates,ans,new ArrayList<>());  
 return ans;  
}  
  
public void func(int ind,int target,int[] candidates,List<List<Integer>> ans,List<Integer> temp)  
{  
 if(target == 0)  
 {  
 ans.add(new ArrayList<>(temp));  
 return;  
 }  
 for(int i=ind;i<candidates.length;i++)  
 {  
 if(i>ind && candidates[i] == candidates[i-1])continue;  
 if(candidates[i]>target)break;  
 temp.add(candidates[i]);  
 func(i+1,target-candidates[i],candidates,ans,temp);  
 temp.remove(temp.size()-1);  
 }  
}

**Time Complexity:O(2^n\*k)**

Reason: Assume if all the elements in the array are unique then the no. of subsequence you will get will be O(2^n). we also add the ds to our ans when we reach the base case that will take “k”//average space for the ds.

**Space Complexity:O(k\*x)**

Reason: if we have x combinations then space will be x\*k where k is the average length of the combination.

**Combination sum – 3:**

public List<List<Integer>> combinationSum3(int k, int n) {  
 List<List<Integer>> ans = new ArrayList<>();  
 func(1,k,n,ans,new ArrayList<>());  
 return ans;  
}  
  
public void func(int ind,int k,int target,List<List<Integer>> ans,List<Integer> temp)  
{  
 if(temp.size()>k||target<0)return;  
 if(temp.size() == k && target == 0)  
 {  
 ans.add(new ArrayList<>(temp));  
 }  
  
 for(int i=ind;i<=9;i++)  
 {  
 temp.add(i);  
 func(i+1,k,target-i,ans,temp);  
 temp.remove(temp.size()-1);  
 }  
}

 **Time Complexity:** O( \binom{9}{k} ) (approximately O(2^k), but better due to pruning).

 **Space Complexity:** O( \binom{9}{k} ) (to store valid combinations) + O(k) (recursive call stack).

**Subset sum – 1:**

Using the logic and concept from subsequences topic learned before.

public ArrayList<Integer> subsetSums(int[] arr) {  
 ArrayList<Integer> ans = new ArrayList<>();  
 int n = arr.length;  
 func(0,0,n,arr,ans);  
 return ans;  
}  
  
public void func(int ind,int sum,int n,int[] arr,ArrayList<Integer> ans)  
{  
 if(ind == n)  
 {  
 ans.add(sum);  
 return;  
 }  
 sum+=arr[ind];  
 func(ind+1,sum,n,arr,ans);  
 sum-=arr[ind];  
 func(ind+1,sum,n,arr,ans);  
}

**Time Complexity:** O(2^n). Each index has two ways. You can either pick it up or not pick it. So for n index time complexity for O(2^n) .

**Space Complexity:** O(2^n) for storing subset sums, since 2^n subsets can be generated for an array of size n

**Subset sum – 2:**

Using the logic and concept from Combination sum-2 topic learned before.

public List<List<Integer>> subsetsWithDup(int[] nums) {  
 Arrays.sort(nums);  
 List<List<Integer>> ans = new ArrayList<>();  
 func(0,nums,ans,new ArrayList<>());  
 return ans;  
}  
  
public void func(int ind,int[] nums,List<List<Integer>> ans,List<Integer> temp)  
{  
 ans.add(new ArrayList<>(temp));  
 for(int i=ind;i<nums.length;i++)  
 {  
 if(i>ind && nums[i] == nums[i-1])continue;  
 temp.add(nums[i]);  
 func(i+1,nums,ans,temp);  
 temp.remove(temp.size()-1);  
 }  
}

**Time Complexity:** O(2^n) for generating every subset and O(k) to insert every subset in another data structure if the average length of every subset is k. Overall O(k \* 2^n).

**Space Complexity:** O(2^n \* k) to store every subset of average length k. Auxiliary space is O(n) if n is the depth of the recursion tree.

**Generate Parantheses:**

Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

Example 1:

Input: n = 3

Output: ["((()))","(()())","(())()","()(())","()()()"]

Example 2:

Input: n = 1

Output: ["()"]

The idea is to add ')' only after valid '('

We use two integer variables left & right to see how many '(' & ')' are in the current string

If left < n then we can add '(' to the current string

If right < left then we can add ')' to the current string

public List<String> generateParenthesis(int n) {  
 List<String> ans = new ArrayList<>();  
 func(ans,0,0,"",n);  
 return ans;  
}  
  
public void func(List<String> ans,int left,int right,String temp,int n)  
{  
 if(temp.length() == n\*2)  
 {  
 ans.add(temp);  
 return;  
 }  
  
 if(left<n)  
 {  
 func(ans,left+1,right,temp+"(",n);  
 }  
  
 if(right<left)  
 {  
 func(ans,left,right+1,temp+")",n);  
 }  
}

## ****Time Complexity:****

1. **Total Number of Valid Parentheses Combinations**
   * The number of valid sequences is given by the **Catalan Number**, which is: Cn=1n+1(2nn)C\_n = \frac{1}{n+1} \binom{2n}{n}Cn​=n+11​(n2n​)
   * This counts the number of valid balanced parentheses sequences.
2. **Backtracking Process**
   * Each valid sequence has 2n characters.
   * At each step, we decide whether to add an ( or ), but we prune invalid branches early.
   * This gives an **approximate time complexity** of: O(4n/n)O( 4^n / \sqrt{n} )O(4n/n​)
   * This follows from the asymptotic approximation of Catalan numbers: Cn≈4nn3/2C\_n \approx \frac{4^n}{n^{3/2}}Cn​≈n3/24n​

### **Final Time Complexity**

O(4n/n)O( 4^n / \sqrt{n} )O(4n/n​)

which grows **exponentially** but is much smaller than O(2^(2n)) due to pruning.

## ****Space Complexity:****

1. **Recursive Call Stack**
   * The maximum depth of the recursion is O(n), as we place at most 2n characters.
2. **Result Storage (List<String> result)**
   * There are C\_n valid sequences stored in the result list.
   * Each sequence takes O(n) space.
   * The total space for storing results is: O(Cn×n)=O((4n/n)×n)=O(4n/n)O( C\_n \times n ) = O( (4^n / \sqrt{n}) \times n ) = O( 4^n / \sqrt{n} )O(Cn​×n)=O((4n/n​)×n)=O(4n/n​)

### **Final Space Complexity**

O(4n/n)O( 4^n / \sqrt{n} )O(4n/n​)

which is the same as the time complexity because the output itself dominates space usage.

### **Summary of Complexity**

| **Complexity Type** | **Value** |
| --- | --- |
| **Time Complexity** | O( 4^n / sqrt(n) ) |
| **Space Complexity** | O( 4^n / sqrt(n) ) (for storing results) |

**Pow (x,n) :**

**Binary exponentiation :**

Implement pow(x, n), which calculates x raised to the power n (i.e., xn).

Example 1:

Input: x = 2.00000, n = 10

Output: 1024.00000

Example 2:

Input: x = 2.10000, n = 3

Output: 9.26100

Example 3:

Input: x = 2.00000, n = -2

Output: 0.25000

Explanation: 2-2 = 1/22 = 1/4 = 0.25

Constraints:

-100.0 < x < 100.0

-231 <= n <= 231-1

n is an integer.

Either x is not zero or n > 0.

-104 <= xn <= 104

public double myPow(double x, int n) {  
 return calculate(x,(long)n);  
}  
  
public double calculate(double x,long n)  
{  
 if(n == 0)return 1;  
 if(n<0)return 1/calculate(x,-n);  
  
 if(n%2==0)  
 {  
 return calculate(x\*x,n/2);  
 }  
 else  
 {  
 return x\*calculate(x\*x,n/2);  
 }  
}

This code efficiently calculates xnx^n using a technique called **binary exponentiation**, also known as **exponentiation by squaring**. Let's break it down step by step to understand its intuition and logic:

### **1. Handling Base Cases**

java

if (n == 0) {

return 1;

}

* **Reason**: Any number raised to the power of 0 is always 1 (x0=1x^0 = 1).

### **2. Dealing with Negative Exponents**

java

if (n < 0) {

return 1.0 / binaryExp(x, -n);

}

* **Reason**: For negative exponents, x−nx^{-n} is mathematically equivalent to 1/xn1 / x^n. To compute this, the function takes the reciprocal of xnx^n. However, since n is a long, negating it avoids integer overflow for edge cases like n = Integer.MIN\_VALUE.

### **3. Recursive Reduction of the Exponent**

The core idea of binary exponentiation lies in reducing the problem size at every step by halving the exponent:

java

if (n % 2 == 1) {

return x \* binaryExp(x \* x, (n - 1) / 2);

} else {

return binaryExp(x \* x, n / 2);

}

#### **Even** nn**:** xn=(x2)n/2x^n = (x^2)^{n/2}

* If nn is even, the power can be computed by squaring xx and halving the exponent. This drastically reduces the problem size in each step.
* Example: x8=((x2)2)2x^8 = ((x^2)^2)^2.

#### **Odd** nn**:** xn=x⋅(x2)(n−1)/2x^n = x \cdot (x^2)^{(n-1)/2}

* If nn is odd, subtract 1 (making it even) and multiply by xx once.
* Example: x5=x⋅(x4)=x⋅((x2)2)x^5 = x \cdot (x^4) = x \cdot ((x^2)^2).
* The if (n % 2 == 1) condition handles the odd case, and the else branch handles the even case.

### **4. Key Insights**

* **Time Complexity**: Instead of multiplying xx nn times (which takes O(n)O(n) time), this method reduces the exponent by half at every recursive step. This gives a time complexity of O(log⁡n)O(\log n), making it ideal for large exponents.
* **Space Complexity**: As the algorithm is recursive, the space complexity is O(log⁡n)O(\log n) due to the recursion stack. For extremely large values of nn, an iterative approach can reduce the space to O(1)O(1).

The reason for converting int n to long n is to handle edge cases where the value of n might lead to overflow. Specifically, when n is the most negative value of a 32-bit signed integer (Integer.MIN\_VALUE), its value is -2,147,483,648.

Here's why this causes a problem:

* In Java, the range of int is from −231-2^{31} to 231−12^{31} - 1. If you negate Integer.MIN\_VALUE (i.e., calculate -n), the result exceeds the positive range of int (2312^{31}), leading to integer overflow. As a result, -n will remain -2,147,483,648, which is incorrect and can cause incorrect behavior in your recursion logic.

By converting n to long, the range is extended to −263-2^{63} to 263−12^{63} - 1. This prevents overflow when negating n, as -n will now be properly handled within the limits of long.

### Example of the Issue:

java

int n = Integer.MIN\_VALUE; // -2147483648

int negatedN = -n; // Still -2147483648 due to overflow

### Correct Handling with long:

java

long n = Integer.MIN\_VALUE; // -2147483648

long negatedN = -n; // Now 2147483648, correctly handled

By using long, the algorithm ensures correctness for all valid inputs.

**Count Good Numbers:**

A digit string is good if the digits (0-indexed) at even indices are even and the digits at odd indices are prime (2, 3, 5, or 7).

For example, "2582" is good because the digits (2 and 8) at even positions are even and the digits (5 and 2) at odd positions are prime. However, "3245" is not good because 3 is at an even index but is not even.

Given an integer n, return the total number of good digit strings of length n. Since the answer may be large, return it modulo 109 + 7.

A digit string is a string consisting of digits 0 through 9 that may contain leading zeros.

Example 1:

Input: n = 1

Output: 5

Explanation: The good numbers of length 1 are "0", "2", "4", "6", "8".

Example 2:

Input: n = 4

Output: 400

Example 3:

Input: n = 50

Output: 564908303

public long MOD = 1\_000\_000\_007;  
public int countGoodNumbers(long n) {  
 long odd = n/2;  
 long even = (n+1)/2;  
 return (int)(calculate(5,even) \* calculate(4,odd) % MOD);  
}  
public long calculate(long x, long n){  
 if (n == 0) return 1;  
 if (n % 2 == 0) {  
 long temp = calculate((x \* x) % MOD, n / 2);  
 return temp % MOD;  
 }  
 else {  
 long temp = (x \* calculate((x \* x) % MOD, n / 2)) % MOD;  
 return temp % MOD;  
 }  
}

### **Intuition**

1. **Number of Choices for Each Position**:
   * **Even Positions**: For digits in even positions, there are **5 choices** (0,2,4,6,80, 2, 4, 6, 8).
   * **Odd Positions**: For digits in odd positions, there are **4 choices** (2,3,5,72, 3, 5, 7).
2. **Dividing the Positions**:
   * For nn, if nn is even, there are n/2n/2 even positions and n/2n/2 odd positions.
   * If nn is odd, there are (n+1)/2(n+1)/2 even positions and n/2n/2 odd positions (since the sequence starts with an even position).
3. **Total Combinations**:
   * The total number of sequences can be calculated as:

\text{Total} = 5^{\text{#even positions}} \cdot 4^{\text{#odd positions}}

1. **Modular Arithmetic**: Since nn can be very large, the resulting value of 5^{\text{#even positions}} \cdot 4^{\text{#odd positions}} can exceed the computational limits. To manage this, use modular arithmetic with MOD=109+7\text{MOD} = 10^9 + 7.

### **Algorithm**

To solve the problem, we can leverage **binary exponentiation** to compute large powers efficiently under a modulus:

1. **Calculate Powers**:
   * Compute 5^{\text{#even positions}} \mod MOD using binary exponentiation.
   * Compute 4^{\text{#odd positions}} \mod MOD using binary exponentiation.
2. **Combine Results**:
   * Multiply the results of the two powers under the modulus:

### **Time and Space Complexity**

* **Time Complexity**: O(log n), as binary exponentiation is logarithmic.
* **Space Complexity**: O(log n), as recursive stack space.

**Palindrome Partitioning:**

**Generate Permutations:**

**N Queen:**

**Rat in a maze:**