Recursion

function calls

→ while the function is not finished or executing it will remain in the stack

→ when a function finishes executing it is removed from the stack and the flow of the program is restored to where the function was called

Base condition

→ condition where our recursion will stop making more function calls

→ eventhough the function is same, every time the function is called, the function will be added to the stack each time (maybe with a different argument)

→ At some point, memory of stack will exceed the limit and this error is called StackOverflow

```
recursion(arguments){
    Base condition
    Body
    recursive call
}
```

Structure

Why Recursion

→ it helps us in solving complex problems in a simpler way

→ you can convert any recursion solution into iteration and viceversa

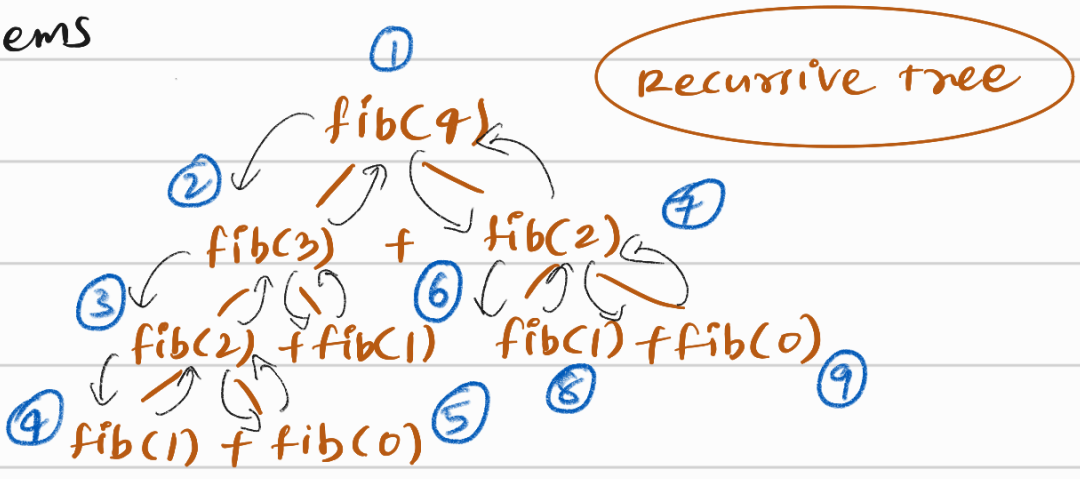
→ space complexity is not constant because of the recursive function calls

Visualising Recursion

→ Recursion Tree

Find n^{th} Fibonnaci number using Recursion

→ see if you can brake the problem into smaller problems



$$fib(n) = fib(n-1) + fib(n-2)$$

↓

Reccurence relation

→ the base condition is represented by answers we already know, here we know

$$fib(1) = 1; \quad fib(0) = 0;$$

→ if the last line in recursive function is the recursive call then it is called tail recursion

```
print(n){
    sout(n);
    print(n-1); → last line of recursive function
}
```

How to approach recursion

→ identify if you can break the problem into smaller problems

→ write reccurence relation if needed

→ Draw the recursive tree

→ About the tree

→ See the flow of program, how they are getting into stack

→ Identify the flow of left and right tree calls

→ Draw the tree and pointer again and again using pen & paper

→ use a debugger to see the flow of the program

→ See how the values are returned at each step, See where the function call will come out of At the end, you will come out of the main function

Working with variables

→ Arguments

→ return type

→ Body of function

Binary search using recursion

→ comparing middle element O(1)

→ dividing into 2 half

$$F(N) = O(1) + F(N/2)$$

↓

Reccurence Relation

Types of Reccurence relation

→ linear reccurence relation (fib)

→ divide and conquere reccurence (BS)

Variables

→ variables that are needed to pass in the future function calls put in arguments

→ if not needed in future function calls (or) only valuable in the function call, put it in the body

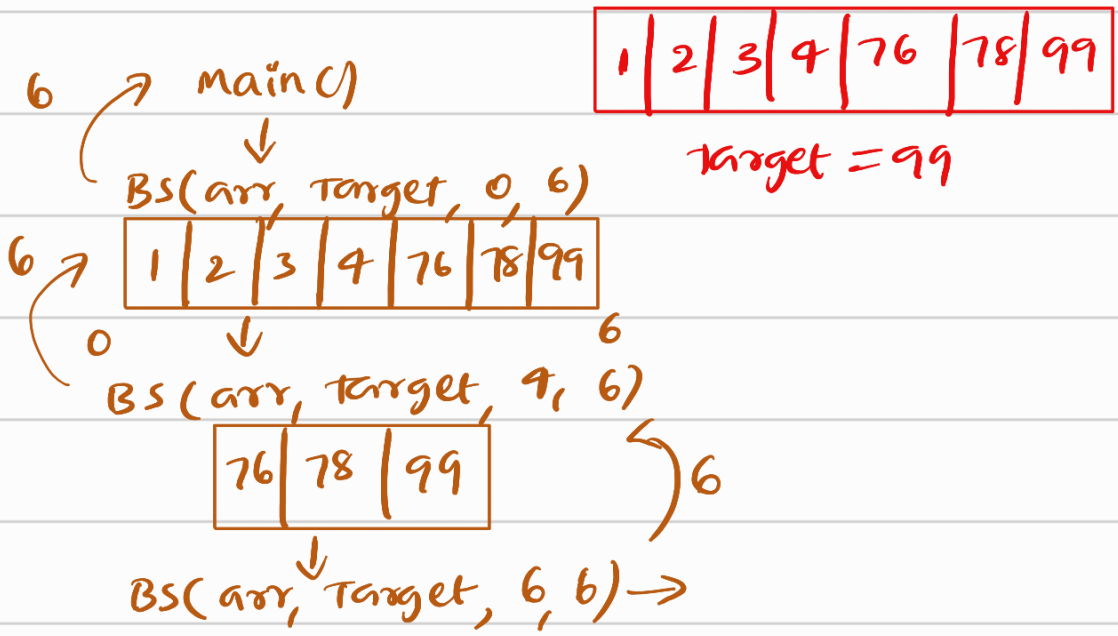
→ make sure to return the result of a function call of the return type

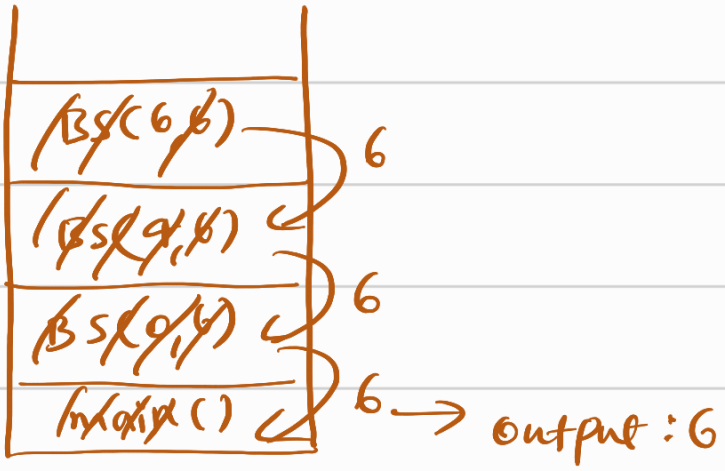
if return type is int → return recursive call

if return type is void → no need to return

→ return the type and return the sub-recursive

calls





→ fibonnaci (fib(n) = fib(n-1) + fib(n-2)

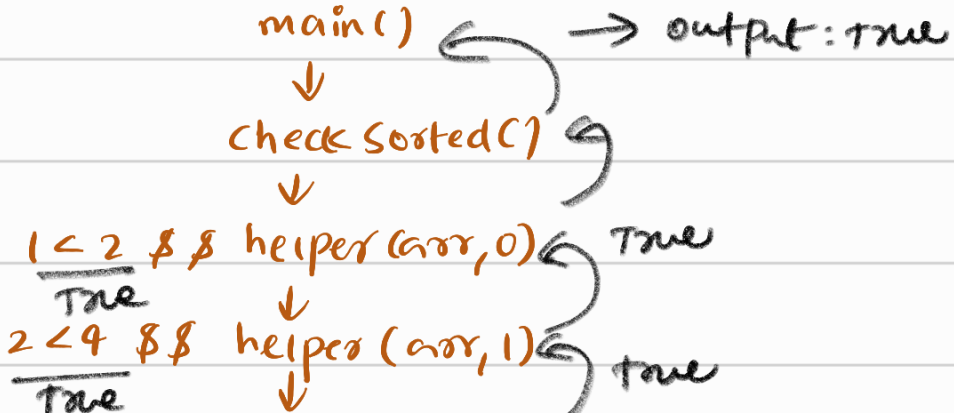
→ N to 1, 1 to N, Both

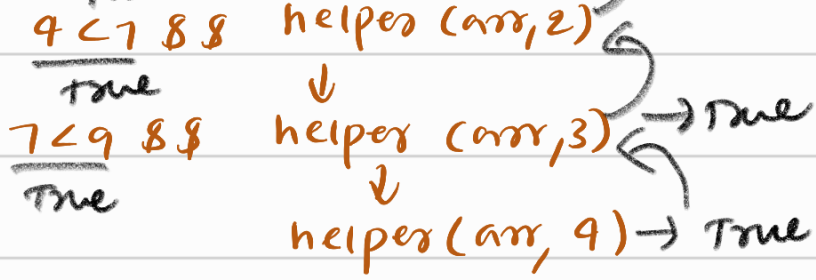
→ factorial (fact(n) = n fact(n-1)

→ sum of digits in a number

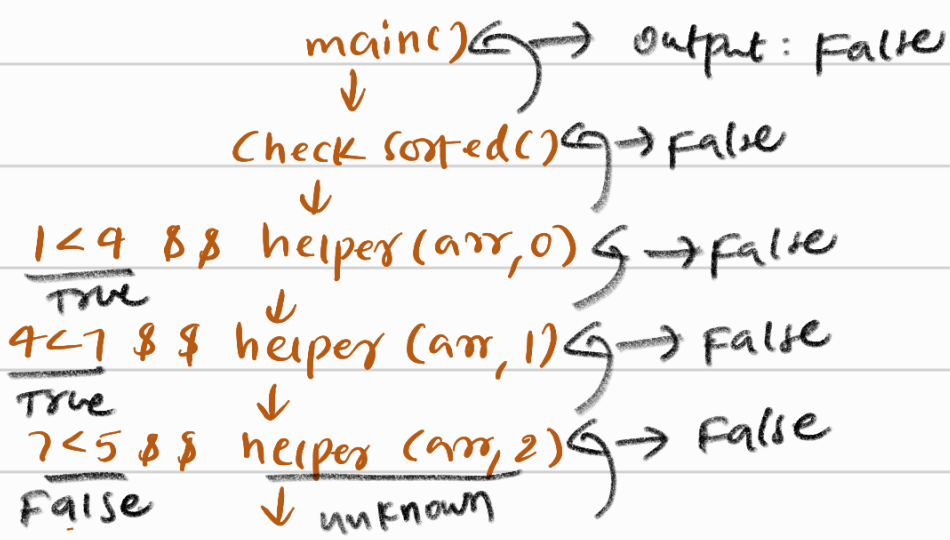
→ count no. of zeros in a given integer

→ check if given array is sorted {1, 2, 4, 7, 9}





{1, 4, 7, 5, 11}



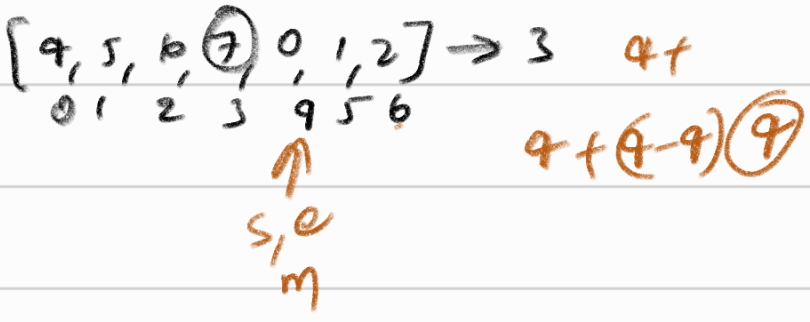
→ search for an element in an array

{3, 2, 1, 18, 9} 18

→ Return the list, don't take it in the argument

(the indexes of the target element)

→ Rotated Binary search for sorted array

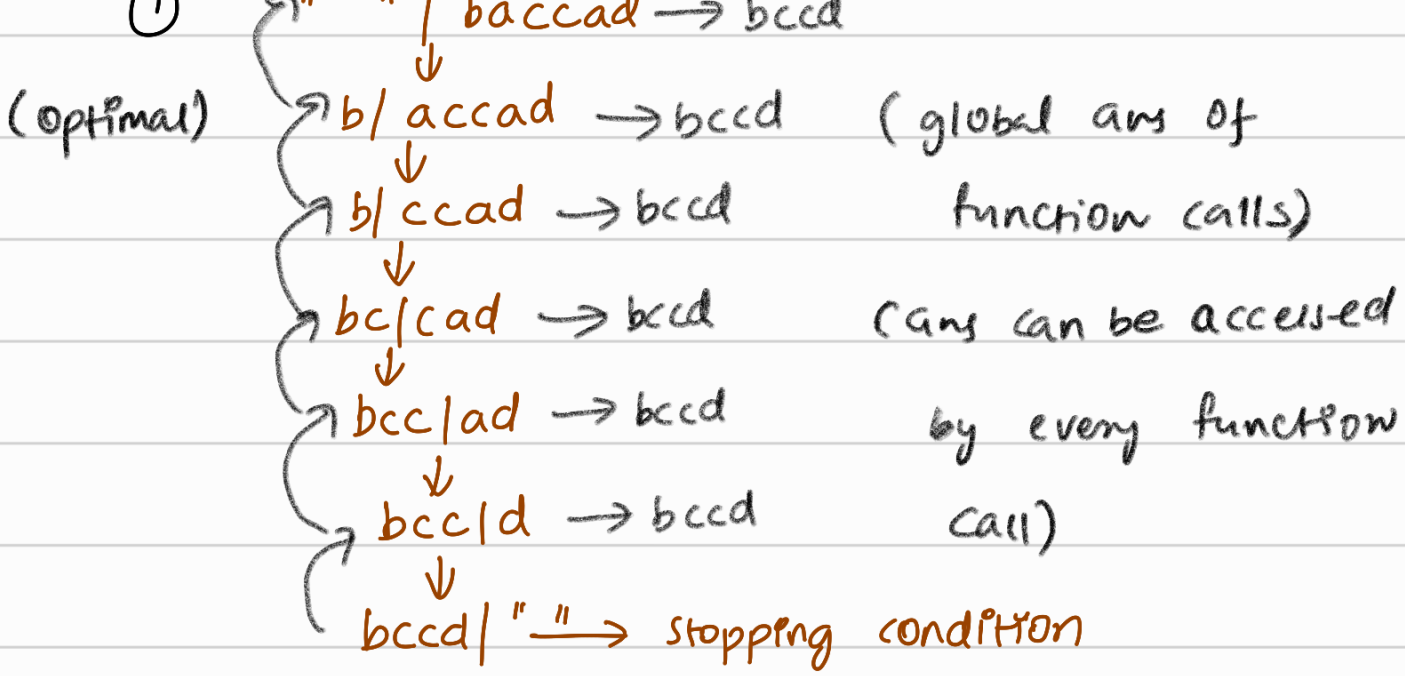


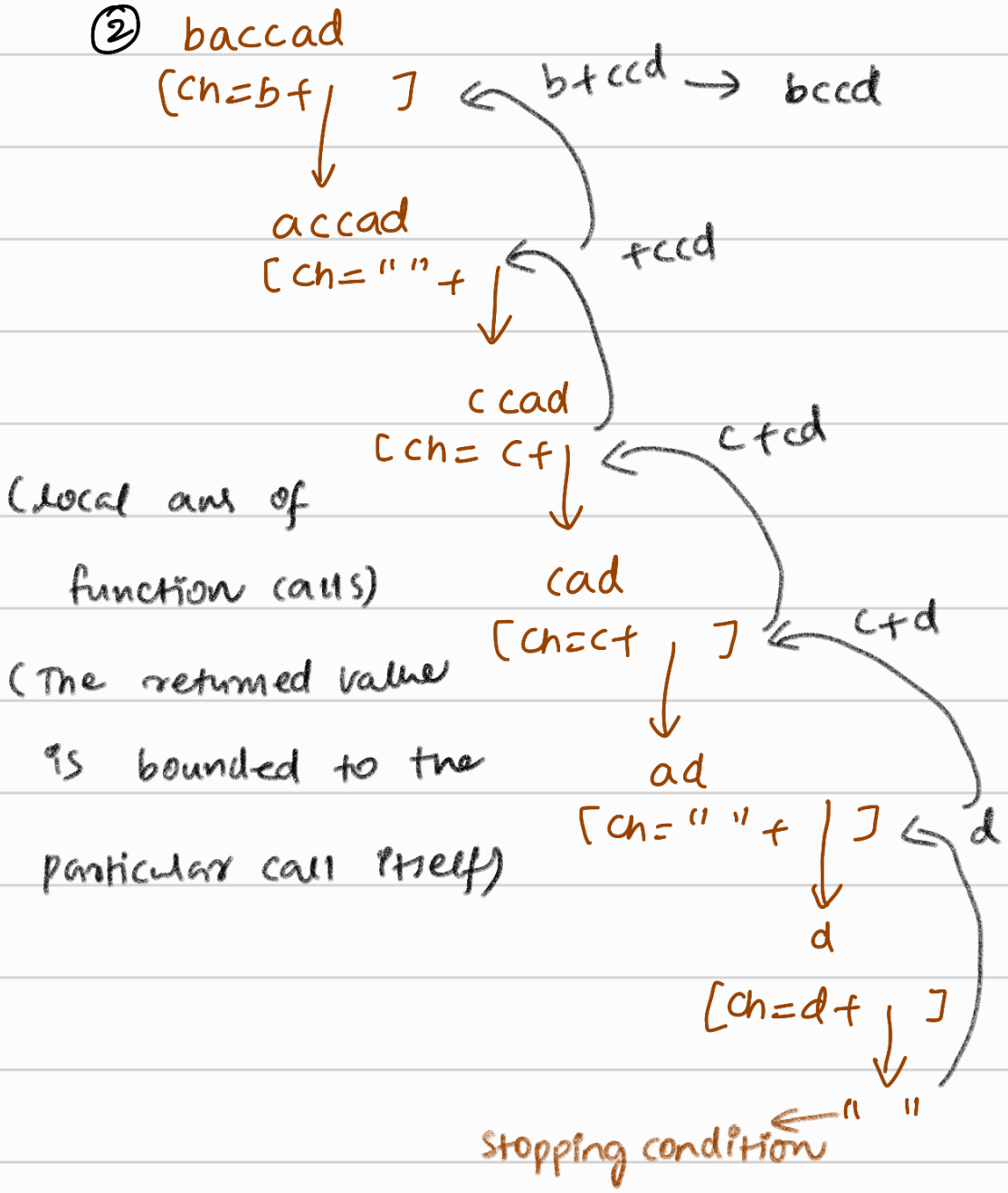
→ Strings

Str = "baccad" ans = "bccd"

Approach 1: pass the ans string in argument

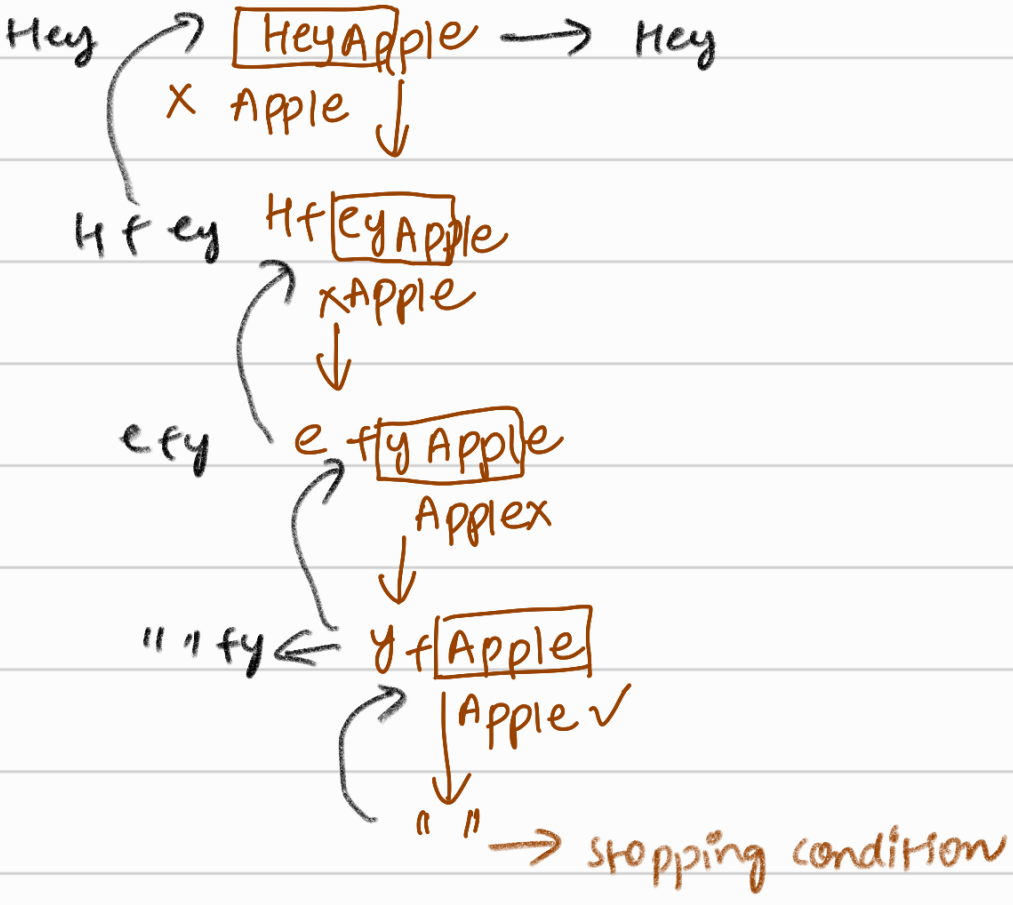
Approach 2: create the ans variable in function



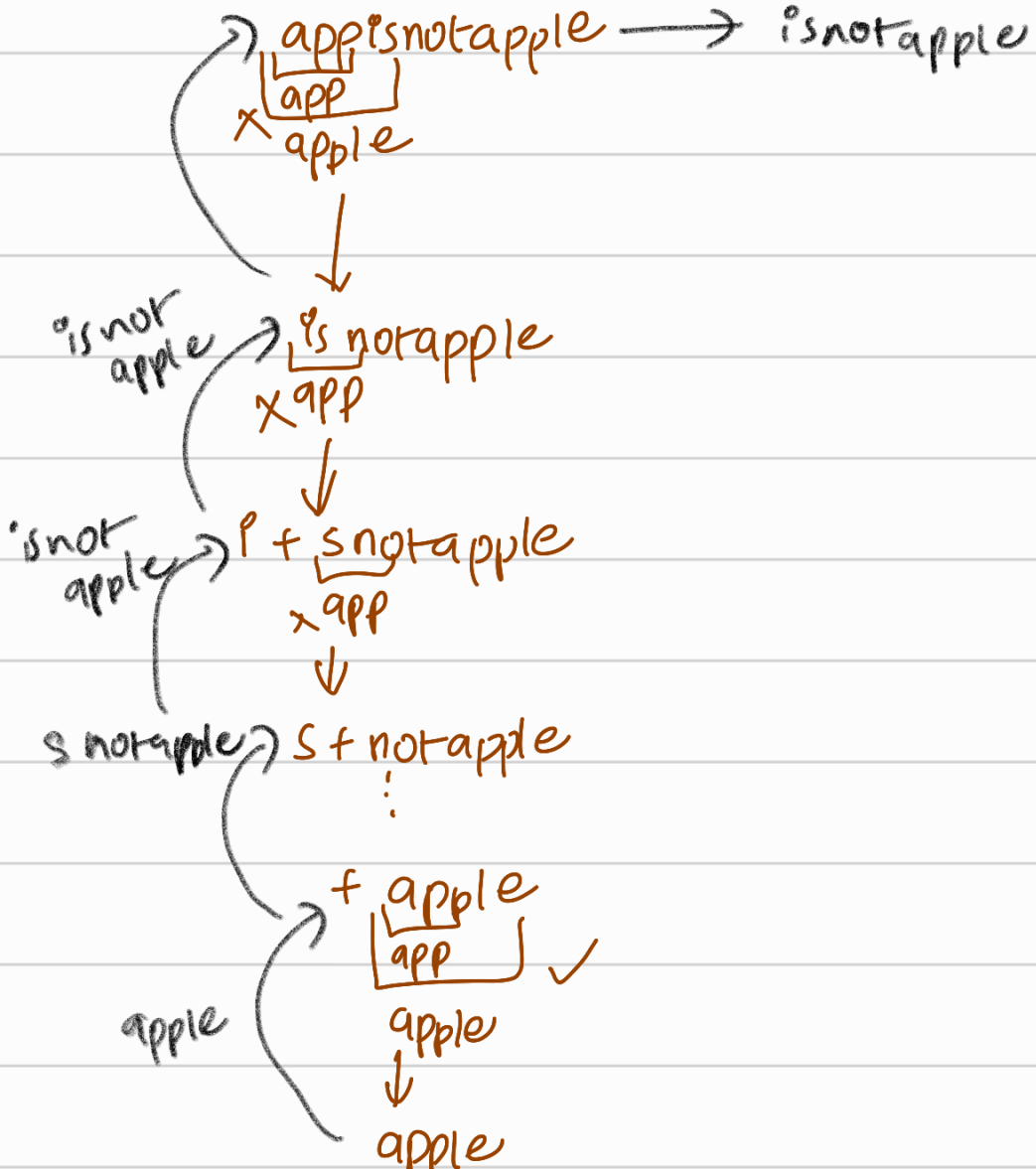


→ skip a given seqence of characters from a string

Str="Hey Apple" skip="Apple" (F)



→ skip "app" when it's not apple



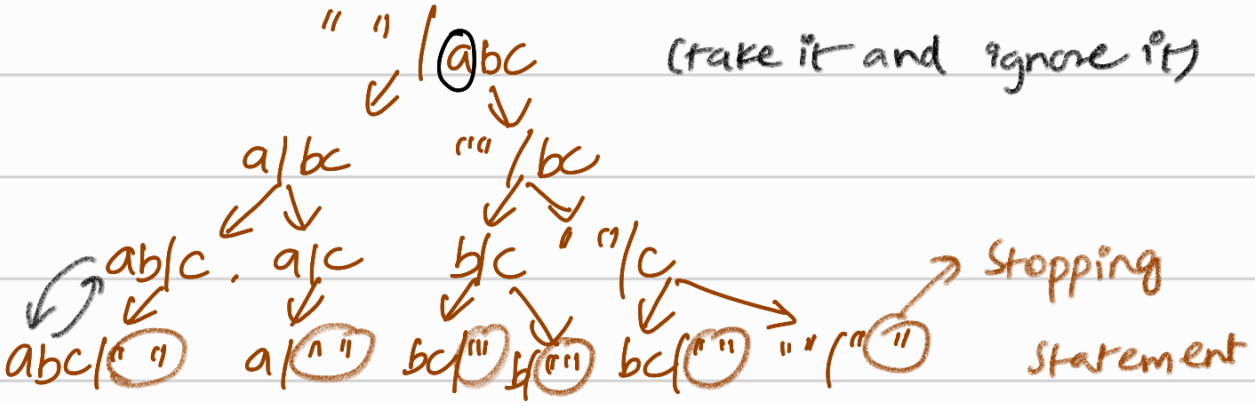
Subsets

→ permutations and combinations

→ subsets – Non-adjacent collection

[3,5,9] → [3,9], [3,5], [3], [], [5], [9], [], [5,9]

→ This pattern of ignoring and considering an element is called subset pattern



→ print the all subsets of string including ASCII values

→ To remove duplicates from the subset, when you find a duplicate, only add it in the newly created subsets of previous step.

→ Because of the above step, duplicates have to be together (so, sort the array before finding subsets)

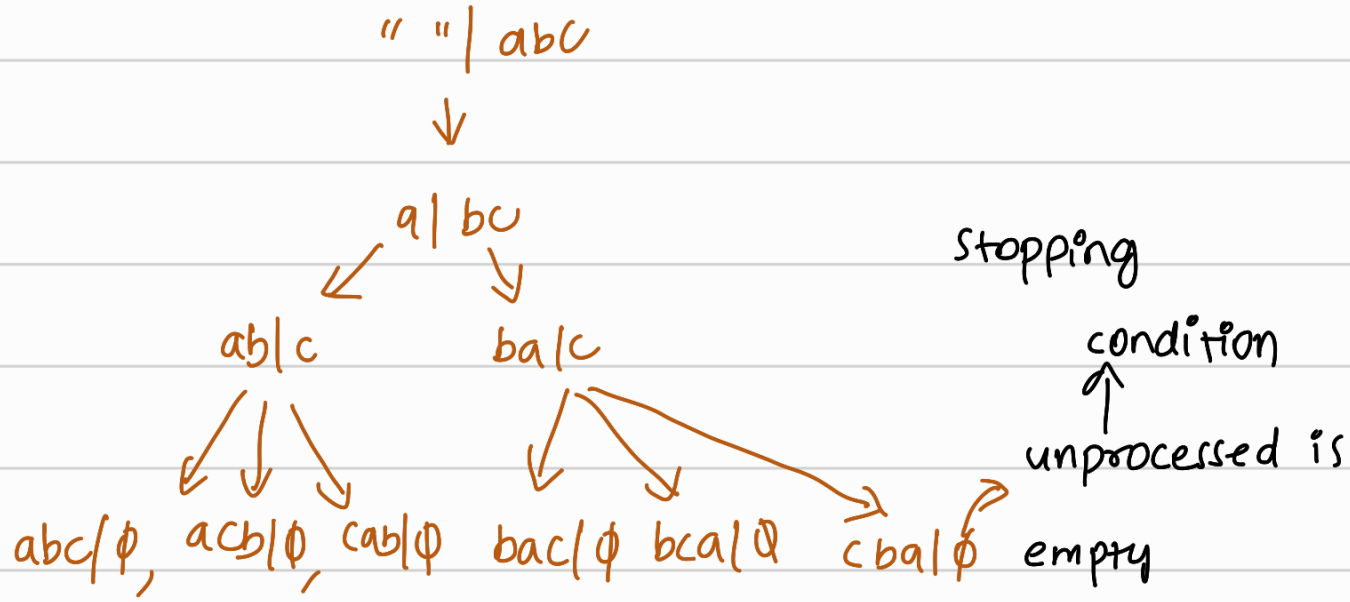
→ maintain two pointers start and end and in the next step dup is found add the element from end+1

→ permutations

S = "abc"

permutations = [abc, acb, bac, bca, cab, cba]

→ permutation is basically inserting at char at all the possible positions

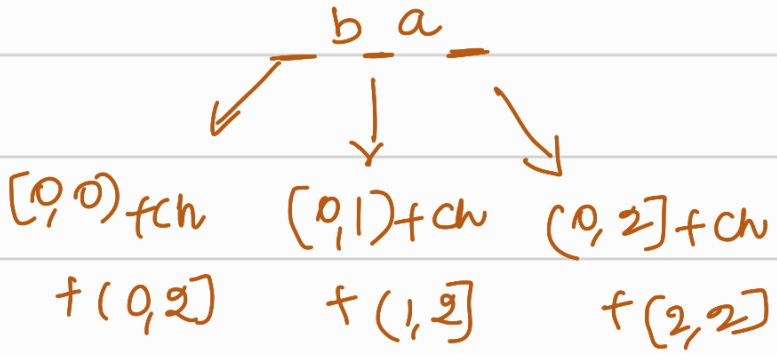


→ The number of permutations possible for a string length n is n!

→ we have variable no. of recursive calls in each step

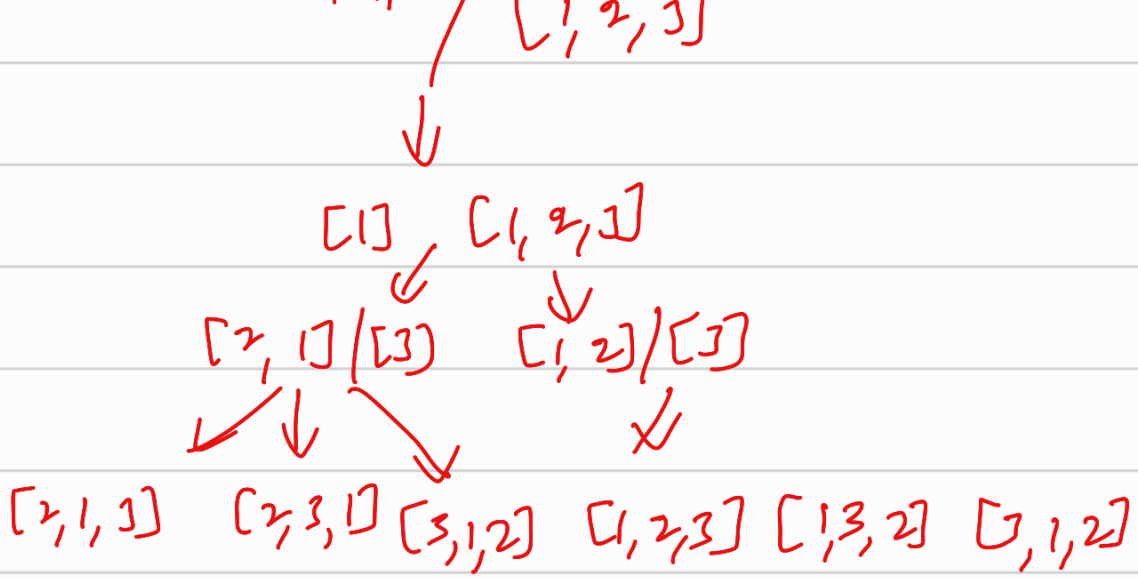
Sometimes, 1 recursion call, 2 and then 3 recursion calls

→ we have p.length()-i no. of recursive calls in each step



[1, 2, 3]

↓


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