* Divide and Conquor: It is an Algorithm of dividing a bigger problem into two equal subproblem & then two subproblem into 2-2 each sub-subproblems. As. Merge Sort & Quick sort are great example of it.

① Quick sort

ex: I/P:

0	1	2	3	4	5	6
8	3	4	1	20	50	30

Quick sort Algorithm:

↳ Pick one element & place it at right position. [Partition Logic]

↳ Then everything else leave it to Recursion.

* Partition Logic:

① choose any pivot element. it could be starting one. or ending one or any random element from an array.

② Now place that element at its right position.

③ Elements left to the pivot element should be smaller than pivot element. & elements right to the pivot element should be greater than pivot element.

ex:

0	1	2	3	4	5	6
8	3	4	1	20	50	30

pivot element

0	1	2	3	4	5	6
1	3	4	8	20	50	30

greater than pivot element

↳ pivot element at its right position

smaller than pivot element

0	1	2	3	4	5	6
8	1	3	4	20	50	30

Here it is said to be pivot because the elements left to that elements is in sorted order & elements right to that element is in sorted order. In simple the element which break the order of sorting called pivot element

Partition logic: ① choose pivot element i.e = 8.

② pivot at right place.

count.

8>1 yes count = ~~1~~ ~~2~~ 3

8>3 yes

8>4 yes

8>20 No swap(arr[pivotIndex], arr[start + count])

Here we have added start as we can use just count because for every Recursive call the value of start varies as its doesn't begin from 0. The array divided into sub parts so start value become different.

*Partition Logic:

① choose pivotIndex & pivotElement i.e start.

② Place pivot Element at its right position.

i.e arr[i] < pivotElement <

0	1	2	3	4	5	6
8	1	3	4	20	50	30

count ++;

i = indexes

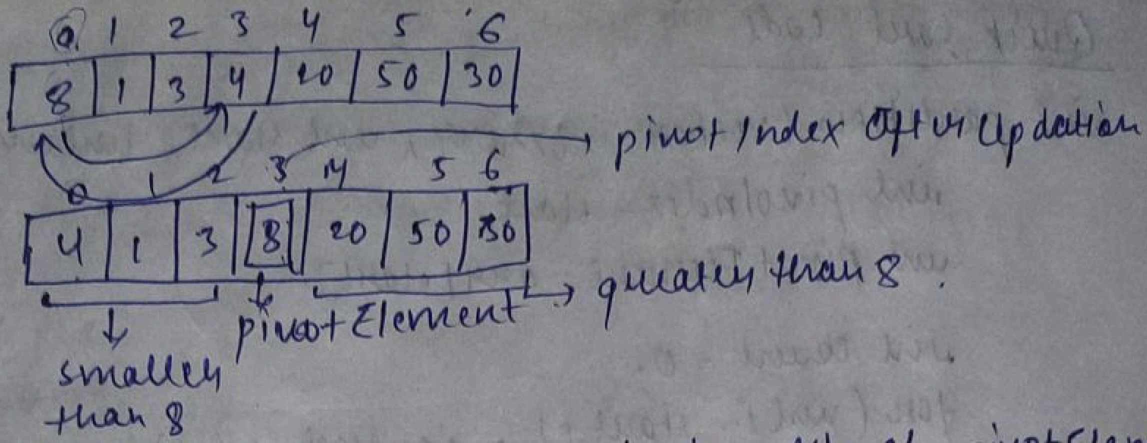
count = ~~1~~ ~~2~~ 3

8>20

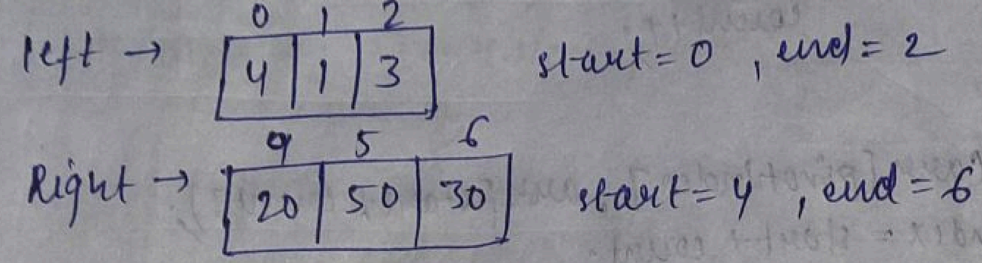
8 will be placed at 3 index & it is 4th element of an array.

③ swap (arr[pivotIndex], arr[s+count]).

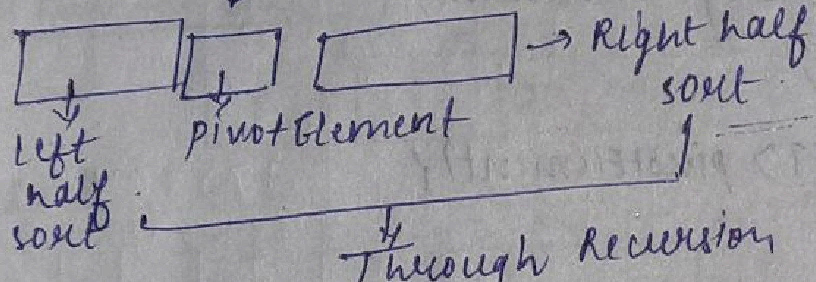
swap (arr[0], arr[3]);



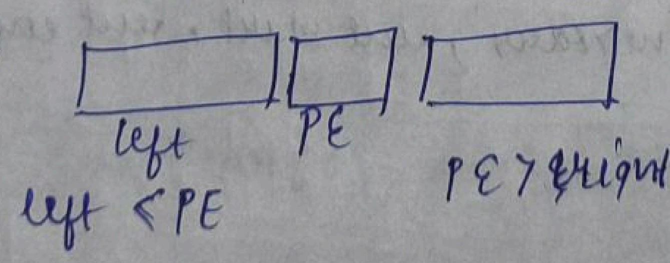
4) Recursive calls to sort left & right part of pivotElement



Quick sort
1) Partition logic
2) Recursive logic.



* Partition logic.
1) int pivotIndex = start
2) if (arr[i] < arr[pivotIndex]) {
       count ++;
   }
   → right place of pivot element.
3) swap (arr[pivotIndex] , arr[start + count]). → swap
4)



5) return pivot Index

Quick sort code

```
int partition (vector<int>&arr, int start, int end){
    int pivoIndex= start;
    int pivotElement = arr[start].

    int count = 0.
    for (int i= start+1 . i<= end ; i++){
        if (arr[i]< pivotElement) {
            count++;
        }
    }

    swap (arr[pivotIndex] , arr[start+ count]);
    pivotIndex = start+ count;

    int i= start.
    int j= end;
    while (i< pivotIndex && j> pivotIndex){
        while( arr[i] < pivot Element) {
            i++;
        }
        while (arr[j]> pivotElement){
            j--;
        }
        if (i< pivotIndex && j> pivotIndex){
            swapp( arr[i], arr[j]);
        }
    }
    return pivotIndex;
}
void quickSort (vector<int>&arr , int start, int end){
    if (start >= end) {
        return;
    }
```

```
int pivotIndex = partition(arr, start, end);
quickSort(arr, start, pivotIndex-1);
quickSort(arr, pivotIndex+1, end);
}
int main(){
    int num;
    cin>>num.
    vector<int> arr(num);
    for(int i=0; i<arr.size(); i++){
        cin>>arr[i];
    }
    int start=0, end= arr.size()-1;
    quickSort(arr, start, end);
    for(int i=0; i<arr.size(); i++){
        cout<< arr[i]<<" ";
    }
}.
```

Dry Run the code

0	1	2	3	4	5	6
8	1	3	4	20	50	30

s = 0 , e = 6

s → 0 ... 6 ← e

0	1	2	3	4	5	6
8	1	3	4	20	50	30

$s \neq e$

index = partition(arr, 0, 6)

pivotIndex = 0

" ele = arr[0] = 8.

arr[1] = 1 < 8 ✓ count = ~~1~~ ~~2~~ 3

= 3 < 8 ✓ ∴ = 3

4 < 8 ✓

20 < 8 ✗

swap(arr[0], arr[0+3])

s → 0 ... 6 ← e

0	1	2	3	4	5	6
8	1	3	4	20	50	30

swap

0	1	2	3	4	5	6
4	1	3	$\boxed{8}$	20	50	30

pivotele

pivot Index = ~~0~~ + 3 (s + count)

PI = 3

i = ~~0 1 2~~ 3 ⑥

j = ~~6 5 4~~ 3

~~3 < 3~~ && ~~6~~ > 3 ✓

4 < 8 ✓ 1 < 3 ✓ 3 < 8 ✓

k = 4 50 > 8 ✓ 20 > 8 ✓

30 > 8 ✓

Original Array

0	1	2	3	4	5	6
4	1	3	8	20	50	30

pass by refrence

PI = 3

quicksort(arr, 0, 2)

0	1	2
4	1	3

PI = 0 . PE = 4

arr[1] 1 < 4 ✓ count = ~~1~~ 2

arr[2] 3 < 4 ✓ = 2

swap (arr[0], arr[0+2])

0	1	2
4	1	3

→

0	1	2 → PI
3	1	4

PE

PI = s + count
= 0 + 2 = 2

i = 0 , j = 2

0 < 2 && 2 > 2

Original Array

0	1	2	3	4	5	6
3	1	4	8	20	50	30

P.B.R

PI = 2

quicksort (arr, 0, 1)

0	1
3	1.

PI = 0

PE = 3

i = 1

arr[1] < 3 count = 1

1 < 3

swap (arr[0] , arr[0+1])

0	1
3	1

→

0	1 → PI = s + count = 0 + 1
1	3

PE

i = 0 j = 1

0 < 1 && 1 > 1 X

PI = 1

Original Array

0	1	2	3	4	5	6
1	3	4	8	20	50	30

quicksort (arr, 0, 0)

0
1

s e

return
Base case

0	1	2	3	4	5	6	
1	3	4	8	20	50	30	O.A

P.B.R

Recursive call stack

qs(s,e)	s=0, e=0 B.C return
qs(s,e)	s=0, e=1, PI=~~1~~
qs(s,e)	s=0, e=2, PI=~~2~~
qs(s,e)	s=0, e=6 PI=3
main	

$PI = \boxed{3}$

quicksort (arr, 4, 6)

4	5	6
20	50	30

s=4, e=6
PI=4, PE=20
i=5 arr[5] 50<20 ✗ count=0
arr[6]

swap (arr[4], arr[s+count])
arr[4], arr[4+0]

4	5	6
20	50	30

PI=4 PE=20

i=4 & j=6
i<4 && 6>4 ✗

PI = 4

0	1	2	3	4	5	6	
1	3	4	8	20	50	30	OA

PBR

quicksort (arr, 5, 6)

quicksort (arr, 5, 6)

5	6
50	30

s = 5, e = 6

pI = 5
pE = 50

i = 5+1 = 6
arr[6] < pE   count = 1
30 < 50

swap (arr[5], arr[5+1])   arr[s+count]
5+1 = 6
arr(5), arr[6]

5	6
50	30

→

5	6
30	50

pI = 6

0	1	2	3	4	5	6
1	3	4	8	20	30	50

OR

PBR

quicksort (arr, 6, 6)

50

s = 6, e = 6

single Element
B.C
return.

0	1	2	3	4	5	6
1	3	4	8	20	30	50

O.R

Pass By Refrence

Recursive Call stack

qs(s,e) — s = 6, e = 6 B.C = return
qs(s,e) — s = 5, e = 6, pI = 6
qs(s,e) — s = 4, e = 6 pI = 4
main().

1	3	4	8	20	30	50

ORIGINAL ARRAY

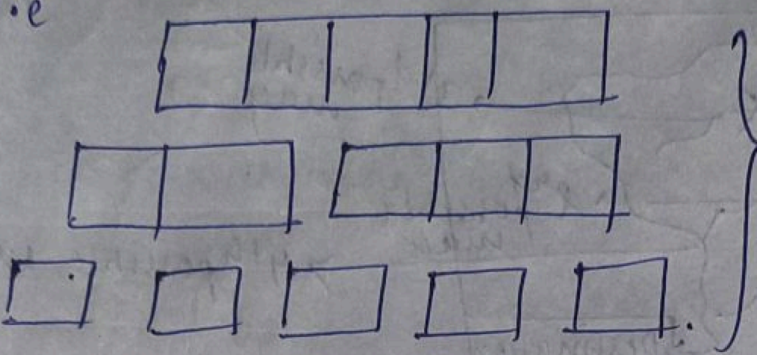
PRINT THE ARRAY
IN MAIN FUNCTION

* Time complexity:

0	1	2	3	4	5	6	7
1	5	5	6	8	20	30	60

As partition logic ~~the~~ the T.C will be
$O(2n) \rightarrow O(n)$

As the array divided into ~~par~~ subarray
i.e



$\log n$

$\frac{n}{2^a} = 1$

$n = 2^a$

$\boxed{a = \log n}$

T.C = $O(n \log n)$
for the best case

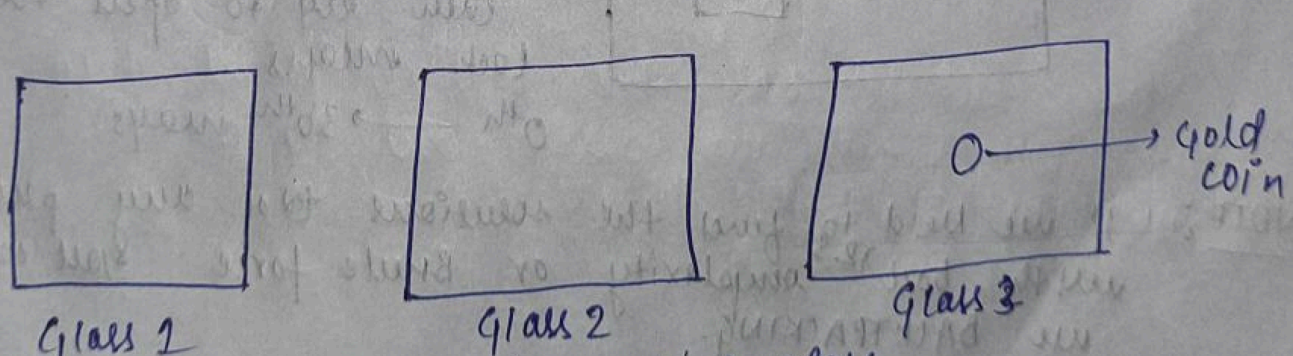
Here equals to 1 until the array become single element as single element is sorted order.

T.C for worst case i.e
when array in reverse order.
T.C = $O(n^2)$

* Space complexity: $O(\log n)$

* Backtracking: It is specific form of Recursion.

- It will check all possible solution.
- Once one solution is checked, it never goes back.



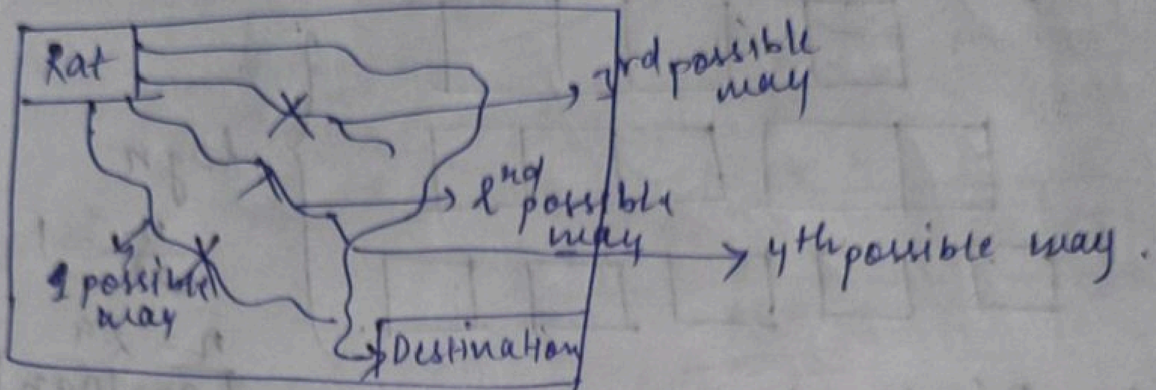
To find gold coin is in which glass.

~~/ / glass 1 / / glass 2.~~

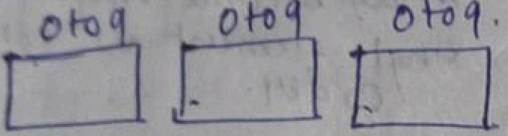
It will check glass 1 if found return else check in glass 2 found return ~~b~~ else it won't go back to glass 1. then go to glass 3 found return.

* Famous Question

Rat in Maze:



* Password find in Lock

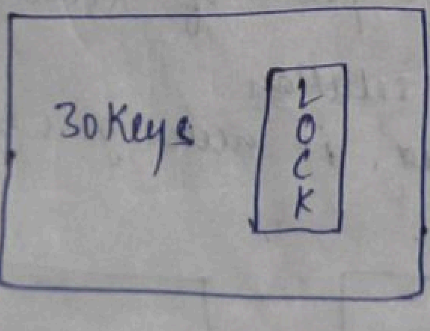


In each block you can put number from 0 to 9

→ so you start from 000 $\xrightarrow{\text{till}}$ 999 possible value you can check.

DESI TARIKA

* Escape Room:



You have Lock & have 30 keys. You have to open the lock.

→ so we can check for each key to open the lock, means

0th → 30th ways.

* NOTE: If we need to find the solutions ~~to~~ of any problem with bad T.C complexity or Brute force you can use BACKTRACKING.

* when do we use BackTracking?
→ we use when we don't have optimum solution of any problem when you can only go for brute force then we will use backTracking.

* Permutations of string:

Input string "abc"

All permutation "abc, bac, bca, cab, cba, acb".

string = "xy"
Permutation = "xy, yx".

string = "abcd".

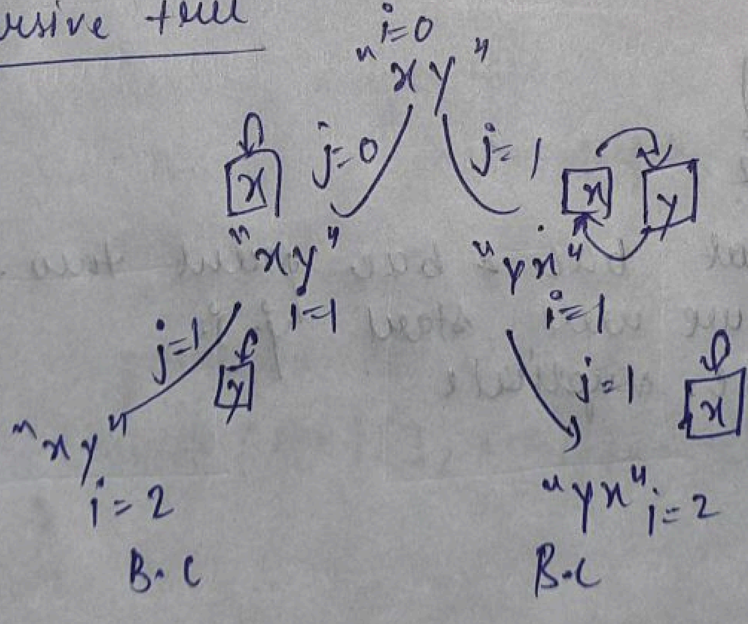
abcd	bacd	cabd	dabc
abdc	badc	cadb	dacb
acbd	bcad	cbad	dbac
acdb	bcdc	cbdc	dbca
adcb	bdac	cdab	dcab
adbc	bdca	cdba	dcbc

It can't be done with inc/exc ~~proway~~

string str = "abc".
abc, bac, bca, cab, cba, acb

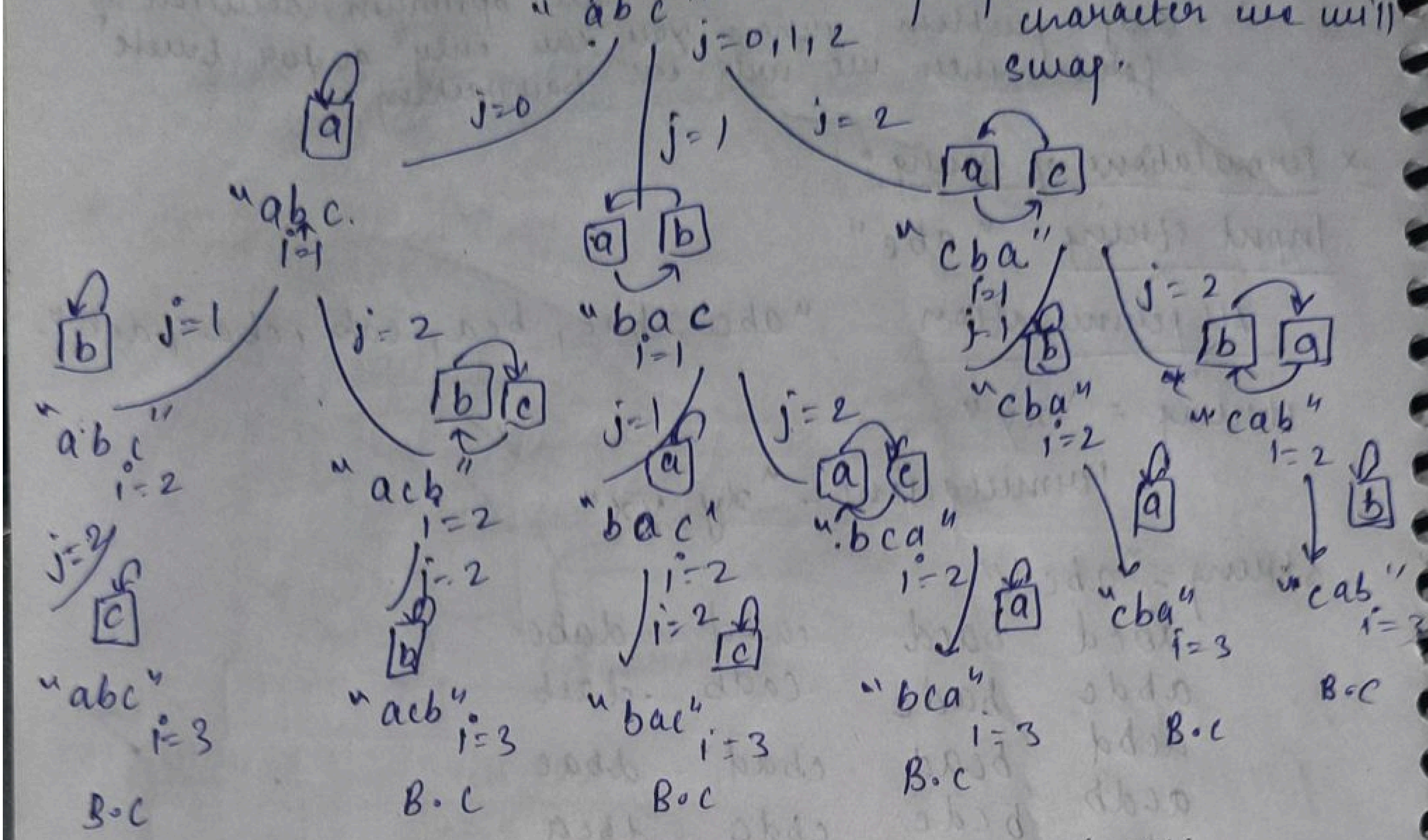
In each block each character comes.

* Recursive tree

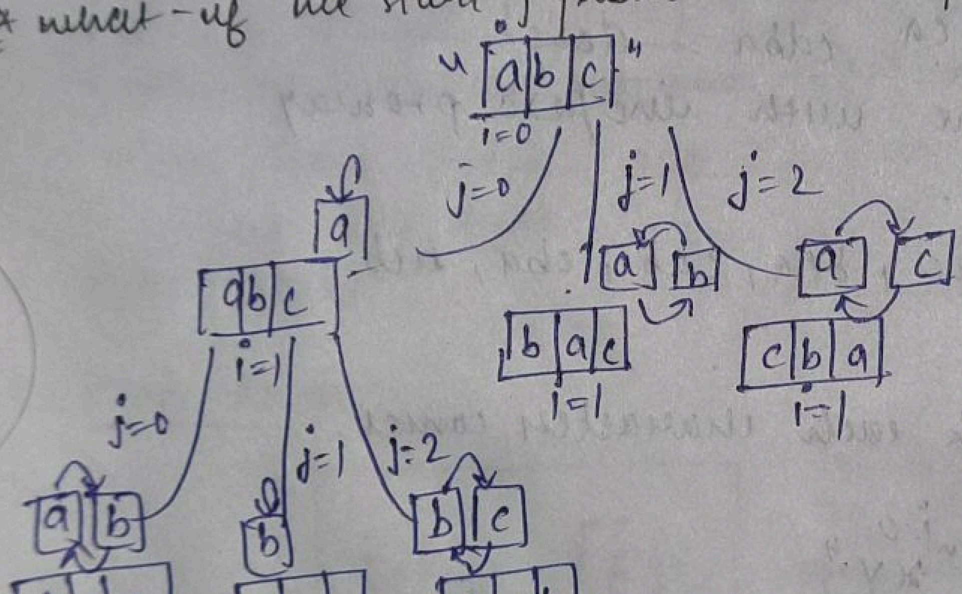


* Recursive Tree

i = position
j = from which character we will swap.



* what-if we start j from 0 ~~with~~ for each case.

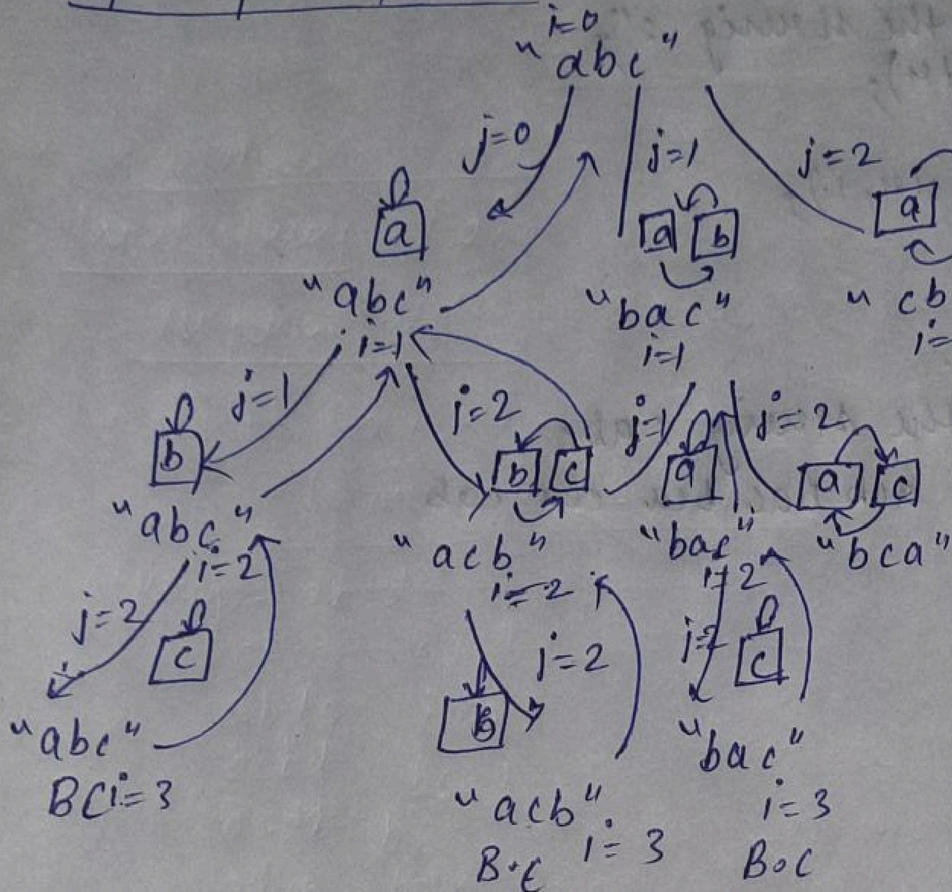


we can see that bac & bac print two times so we will start j = i.
to not to create duplicate.

* We need to pass the vector by reference:

If we pass by value

(j = i; j < str.length(); j++;)



In pass by value you will find that their will create the duplicate string's

```
void permutations (string &str, int i){
    //base case.
    if(i >= str.length()){
        cout<< str << " ";
    }   return;
}

//processing single case
for(int j = i; j<str.length; j++){
    ~~// swap~~
    swap(str[i], str[j]);
    permutation(str, i+1).
    swap(str[i], str[j]); // backtracking
  }
}
```

```
int main (){
    string str;
    cout <<"Enter the string :";
    getline (cin, str);
    int i=0;
    permutation (str, i);
}.
```

Output

Enter the string : abc
abc acb bac bca cba cab.