19/03/2023

Quick Sort

We will be given an array & we need to sort in the increasing order.

0 1 2 3 4 5 6

i/p → {8, 3, 4, 1, 20, 50, 30}

s ↑ ↑ e

Quick sort says that just place one number at the right place & rest recursion will handle.

Let's pick 0th index element & place at the right position. This is basically known as the partition logic.

{1, 3, 4, 8, 20, 50, 30}

sorted by recursion ← ↑ → sorted by recursion

Placed at right position

8 is also known as pivot element. The elements to the left of pivot will be lesser than or equal to the pivot element & that to the right of the pivot will be greater elements.

Quick Sort = Partition logic + Recursive call.

Dry run of quick sort

1) Partition logic

→ Select the pivot element

pivot = 8 } pivotIndex = s

→ Now place pivot element at right place.

// Count elements smaller than 8.

{8, 3, 4, 1, 20, 50, 30}

count = 3 → {3, 1, 4}

✓ Now just replace the pivot element with the element present at index = count + s
Hence swap arr[pivotIndex], arr[s+count]
Now the array becomes

↑ pivot
{4, 1, 3, 8, 20, 50, 30}

→ Now we need left to pivot elements lesser than pivot. Also right to pivot elements need to be greater than pivot.

i j
{4, 1, 3, 8, 20, 50, 30}

Do i++ if arr[i] < pivot & do j-- if arr[j] > pivot. Here partition has been created.

2) Recursive call
→ Now recursive calls will go for arrays {4, 1, 3} and {20, 50, 30}

Code

```
// Partition function
int partition (int arr[], int s, int e) {

        //Select pivot
    int pivotIndex = s;
    int pivot = arr[s];
  // Find right place of pivot
    int count = 0;
    for (int i = s+1; i<=e; i++) {
            if (arr[i] <= pivot)
                count++;
    }
  // Index of right place is known now. So swap it
    int rightIndex = s + count;
```

```
swap (arr [pivotIndex], arr [rightIndex]);
pivotIndex = s+count ; //Update pivotIndex

//Create the partition now
int i= s ;
int j = e ;
while (i< pivotIndex && j> pivotIndex){
        //No need to swap
        while (arr [i] <=pivot)
                i++ ;
        while ( arr [j] > pivot)
                j-- ;
        //Now we need to swap

        if (i< pivotIndex && j>pivotIndex)
        {
        swap (arr [i], arr [j]) ;
         i++ ;
         j-- ;
        }
  }
  return pivotIndex ;
}
//Quick Sort function
void quickSort (int arr [], int s, int e){
        //s==e } Single element sorted
        //s >e } Invalid array
        if ( s>=e)
                return ;
        //Partition
        int p= partition (arr , s , e) ;
        //Recursion logic
```

```
// Call for left array
quickSort (arr, s, p-1);
// Call for right array
quickSort (arr, p+1, e);
}
```

Note → It is not necessary that pivot has to be 1st element. It can be last element also. Also it can be any random element & where pivot is a random element, it is known as randomized quick sort.

Time Complexity Analysis of Quick Sort

The time complexity of quick sort algorithm is $O(n \log n)$ in best and average case.
The time complexity in worst case is $O(n^2)$.
↳ Reverse sorted

Partition logic ⇒ $O(2n) \rightarrow O(n)$. Suppose that pivot will be placed at middle i.e it is a very balanced case

$$n \rightarrow \frac{n}{2} \rightarrow \frac{n}{4} \text{ ---} \rightarrow 1$$

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

$k = \log_2 n$ levels

Hence $O(n \times \log_2 n) = O(n \log n)$ is the time complexity.

→ Place at right place

{7, 6, 5, 4, 3, 2, 1} → n elements

Call will go for

{1, 6, 5, 4, 3, 2} → n-1 elements

Call will go for

{6, 5, 4, 3, 2} → n-2 elements

⋮

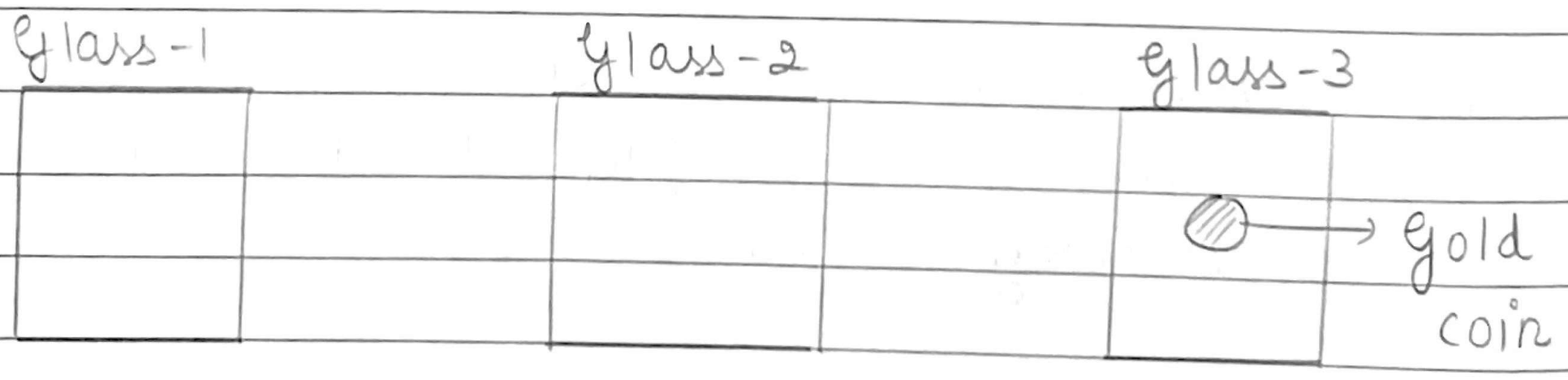
→ 1 element

Total n times partition will be done & in each partition, n time will be taken, so TC = O(n x n)

TC = O(n²)

Hence this is the worst case time complexity.

Backtracking



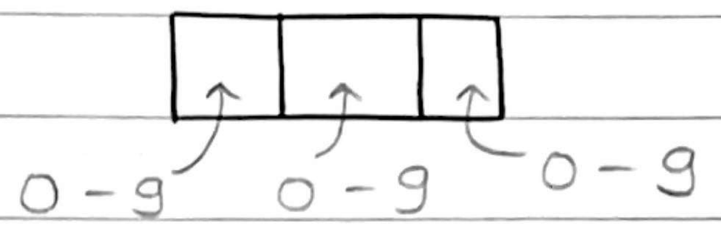
Pick glass-1 & we didn't find gold coin. Now pick glass-2 and then also we didn't find gold coin. Now pick glass-3 & we found gold & here we won't go back to glass1 & glass2 as we have discarded them.

Backtracking is special form of recursive solution. In backtracking all possible

solutions are explored & once we have explored, then we won't explore it again just like we did in glass & coins example.

Note → Backtracking is basically done by adding some lines of code in the recursive code.

Ex → Password block



All the permutations are done & once we find solution, then stop. All permutations will be 000 to 999.
The above concept is known as backtracking.
This is basically brute force approach & the time complexity will be very bad.

Note → There are problems such as N-Queen in which once we have found solution, then also we will further explore the other solutions.

Q → Permutations of string.

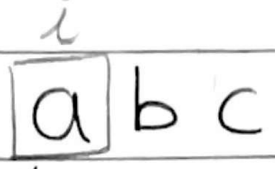
i/p → "abc"
o/p → abc, bac, bca, cab, cba, acb

abcd → abcd, abdc, acbd, acdb, adcb, adbc ---

Total permutations = n! where n is the no. of characters in the string.

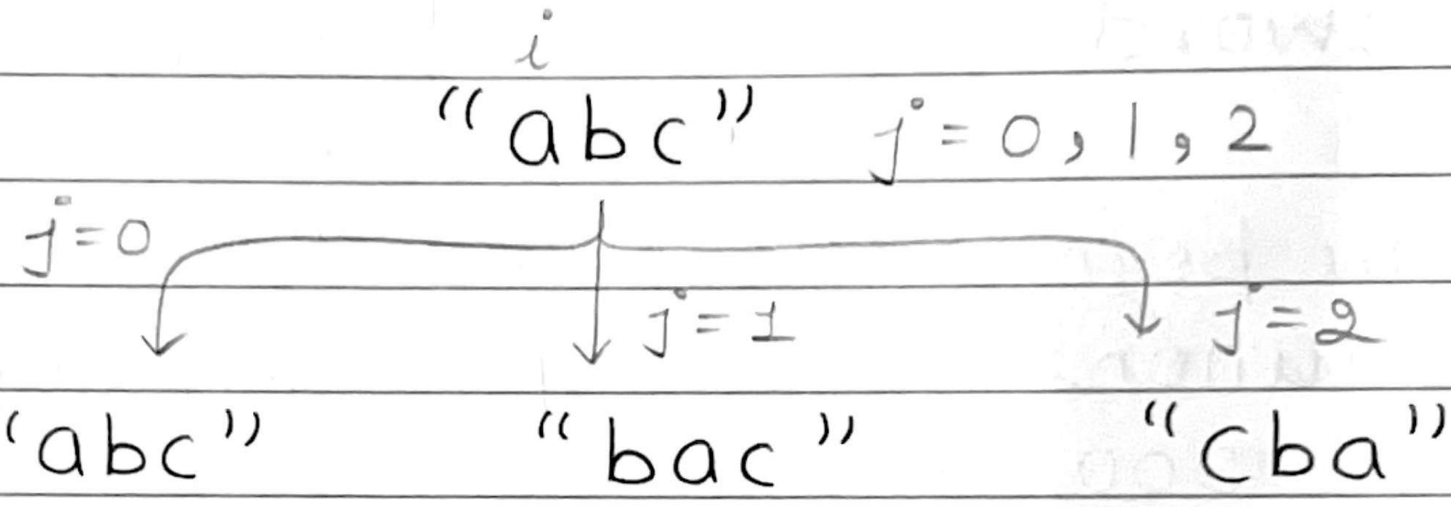
What can we observe here? We can observe

that each & every character wants to come at every position.

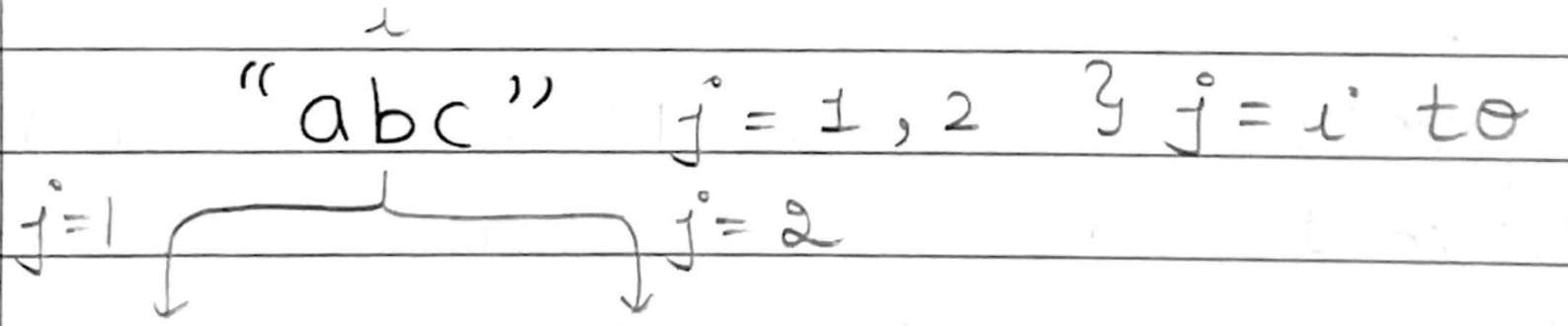


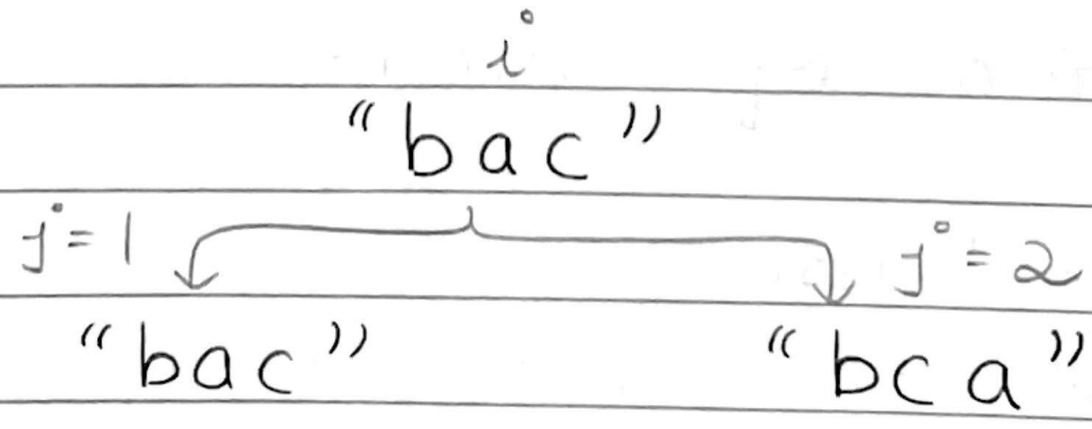
↳ At this block every character will come.

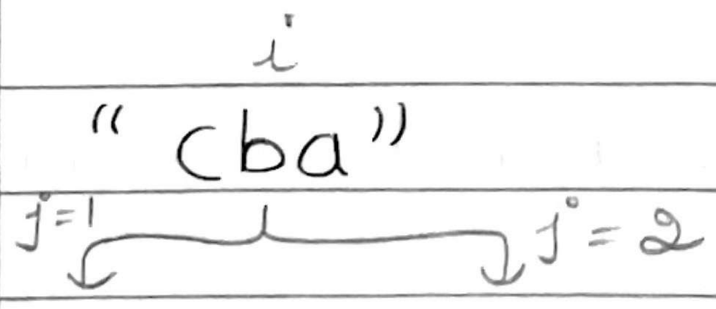
Dry run



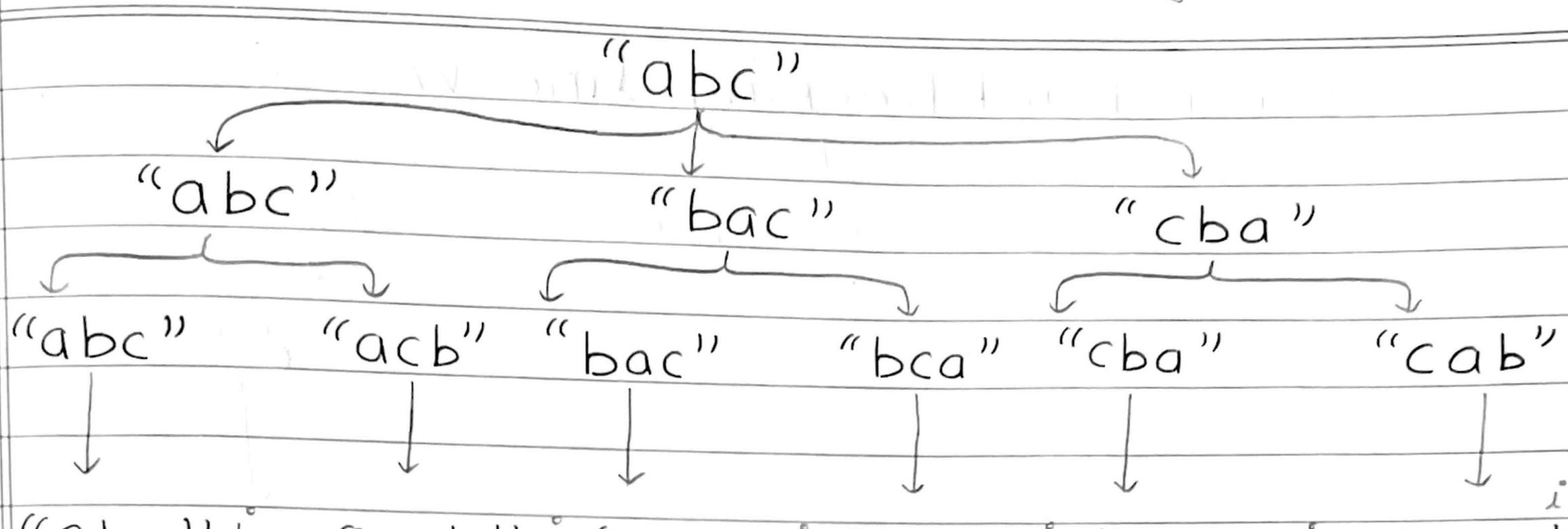
We have swapped arr[i] and arr[j].
Hence in total 3 calls will be gone here.







Here 2 calls have gone. We have to make sure that j moves from i to end else we will get repeated permutations.

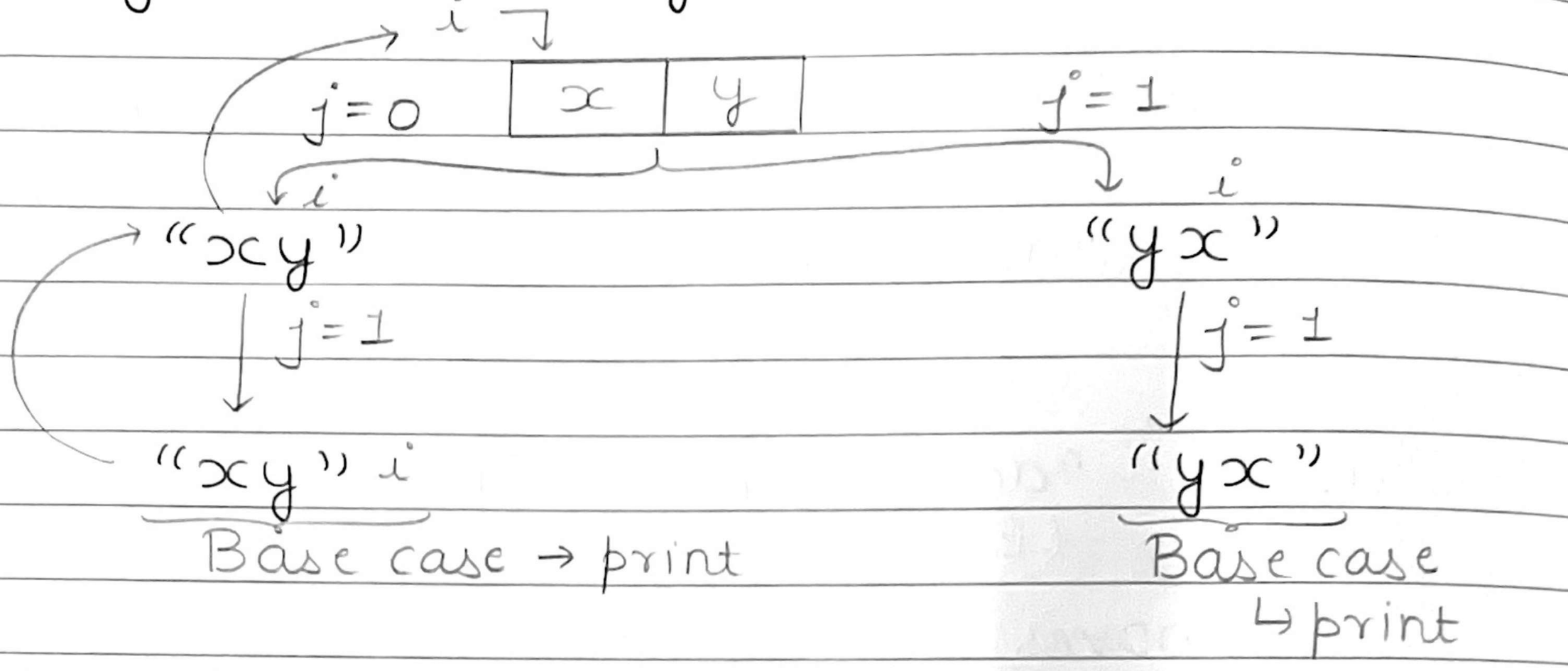


As i moves out & hence we see it as a base case. The above approach has an issue however. We will fix the issue with the help of backtracking code.

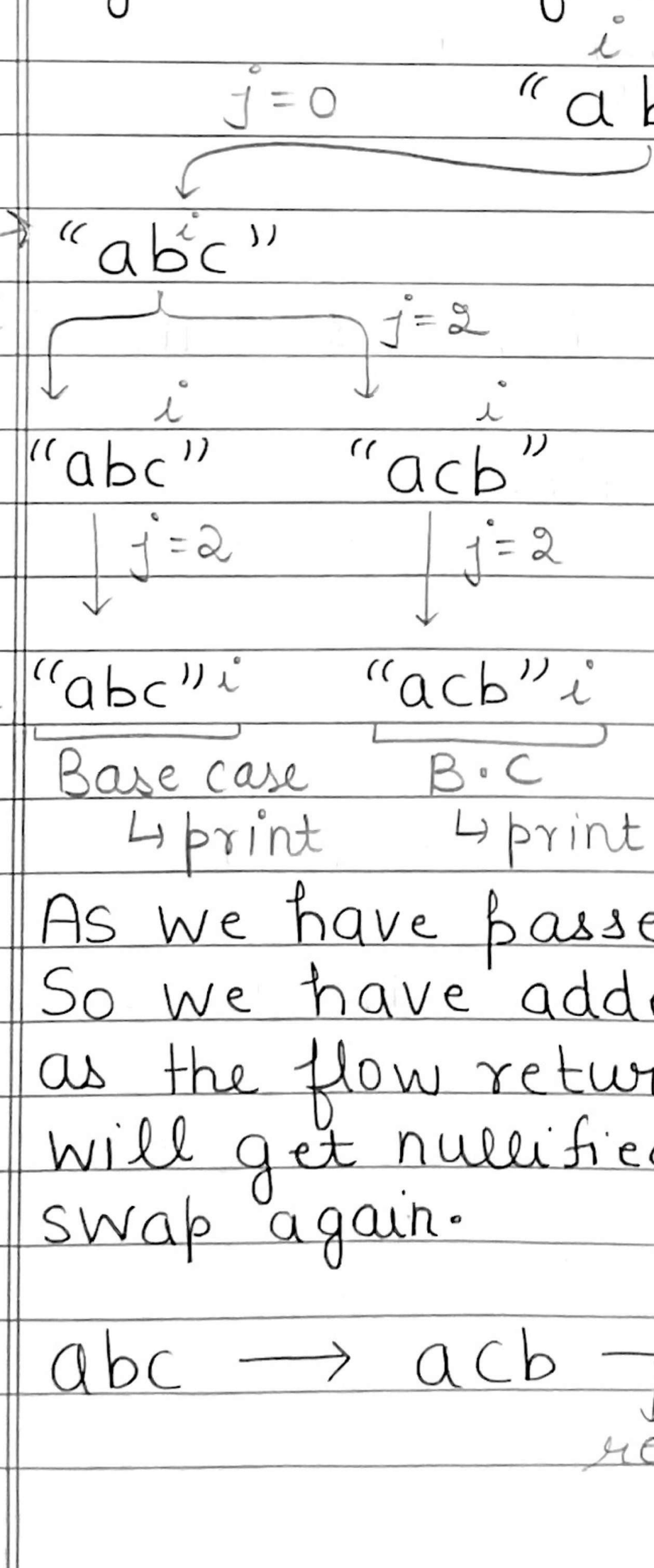
Code

```
void printPermutations (string &s, int i) {
    //Base Case
    if (i >= s.length()) {
        cout << s << " ";
        return;
    }
    //Swapping
    for (int j = i; j < s.length(); j++) {
        //Swap
        swap(s[i], s[j]);
        //Recursion
        printPermutations(s, i+1);
        //Backtracking
        swap(s[i], s[j]);
    }
}
```

Why backtracking line was added?



Here there was no issue. Let us understand by the case of 3 characters.



As we have passed the string by reference. So we have added backtracking line & as the flow returns, the modification will get nullified as we have done the swap again.

abc → acb → abc

↓ returning (Recreate original string)

If we didn't include backtracking line & string was passed by reference only, then some permutations will be repeated as the swap operation is being done on the modified strings.

Time complexity analysis of above question
Time complexity = O(n!)