DnC - 2 Mar 19 / 2023

Quick Sort →

i/p → array →

8	3	4	1	20	50	30

Quick Sort

① → 1 no. into it's right place
→ baki recursion.

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

② ~~8 at~~ right place of 8 → 4th position
⇒ 3rd index.

1	3	4	8	20	50	30

now 8 is in it's right place.

③ place all element < pivot into left and > pivot into right side of the array.

Partition

→ element → pivot
→ pivot → right position
→ pivot
- left side: all elements < pivot
- right side: all elements > pivot.

1	3	4	8	20	50	30

left → Recursion
right → Recursion.

QS →

→ Partition algo/logic
→ Recursion logic.

Partition logic has basically two works.

1→ Place the pivot element into it's correct position.

2→ The element lesser than pivot element should be in left and the elements greater than pivot element should be in right side of pivot.

arr

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

Partition algo →

Steps

- (A) pivot = 8
- (B) pivot → place in it's right place
  count small than pivot
  ⇒ count = 3

  → if (arr[i] < pivotElement) count ++;

  count = ~~0~~ ~~1~~ ~~2~~ 3

  ⇒ swap(arr[pivotIndex], arr[s+count])

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

- (C) Place all elements less than pivot in left and all elements greater than pivot in right side of array.

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

- (4) Recursive Calls → left

0	1	2
4	1	3

s, e

→ right

4	5	6
20	50	30

s, e

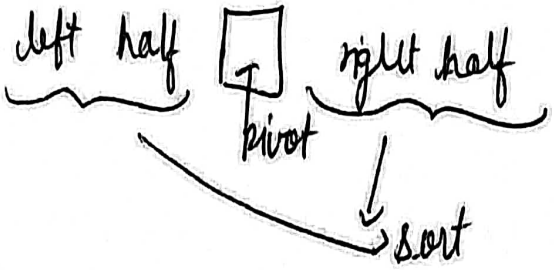
```
int main (){
    int arr[] = {8,1,3,4,20,50,30};
    int n = 7;
    int s = 0;
    int e = n-1;
    quicksort (arr, s, e);
    return 0;
}
void quicksort (int* arr, int s, int e){
    if(s>=e)
        return;
int p = partition (arr, s, e);
    // recursive calls
    // left
    quicksort (arr, s, p-1);
    //right ~~arr~~.
    quicksort (arr, p+1, e);
}
```

```
int partition(int* arr, int s, int e){
    int pivotIndex = s;          <------ Step ① -> choose pivot element
    int pivotElement = arr[s];
    // Step ② -> Find right position for pivot element and
    place it there.
    int count = 0;
    for (int i = s+1; i<=e; i++){
        if(arr[i] <= pivotElement){
            count ++;
        }
    }
    // Now when I came out of loop, I have right index
    of pivot element.
    int rightIndex = s + count;
    swap(arr[pivotIndex], arr[rightIndex]);
    ~~// Step 3~~
    pivotIndex = rightIndex;
    // Step 3 -> left m chote, right m ~~et~~ bade elements.
    int i = s;
    int j = e;

    while (i< pivotIndex && j> pivotIndex){
        while (arr[i] <= ~~pivotIndex~~ pivotElement){
            i++;
        }
        while (arr[j] > pivotElement){
            j--;
        }
        // We can ~~to~~ encounter 2 cases ->
            ① You found the elements to swap
            ② No need to swap.
        if (i< pivotIndex && j> pivotIndex)
            swap (arr[i], arr[j]);
        }
    }
    return pivotIndex;
}
```

Quick Sort →

- Partioning logic
  - responsible for placing at it's right position
  - and
- Recursive logic



- **Partioning Logic →**
  1. choose Pivot
  2. Place pivot into it's right position
  3.

     left → all elements will be smaller or equal than pivot.

     right → all elements will be greater than pivot.
  4. return pivot Index.

Dry Run →

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

→ pivotIndex = 0

rightplace → count = 4

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

i = 0, j = 7

6 < 8 ↳ i++

1 < 8 ↳ i++

20 < 8 → F, i = 2

swap (arr[2], arr[7])

5 > 8 → F, i = 7

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

5 < 8 → i++ i = 3

30 < 8 → F i = 3

swap(arr[3], arr[5])

20 > 8 , j-- → j = 6

60 > 8 , j--, j = 5

5 > 8 → F, j = 5

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

5 < 8 → i++, i = 4

30 > 8 → T → j--, j = 4

i = j = pivotIndex → call for left recursion
call for right recursion.

→

0	1	2	3
6	1	5	5

s e

PI = 0

count = 3

0	1	2	3
5	1	5	6

i j pivotIndex

j = pivotIndex → call for left recursion
→ call for right recursion

→

0	1	2
5	1	5

s e

→ PI = 0

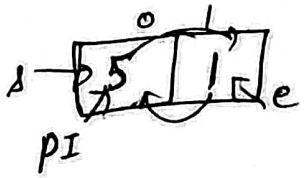
→ count = 2

0	1	2
5	1	5

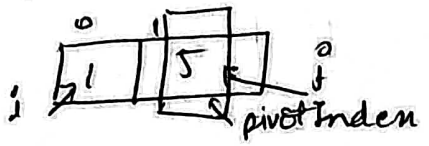
pivot Index = 2

i j

j = pivotIndex
↳ call for left recursion
↳ call for right recursion

→ 

PI = 0
Count = 1

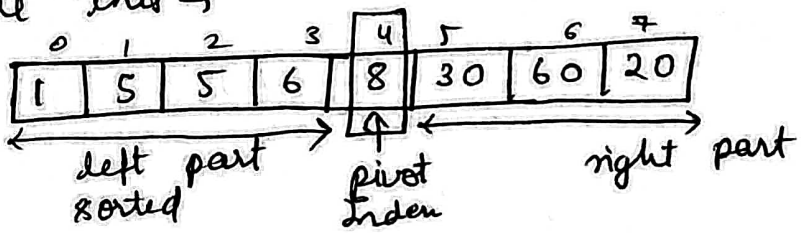


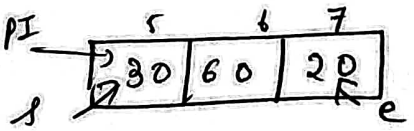
j = pivotIndex → call for left recursion
→ call for right recursion

→ 

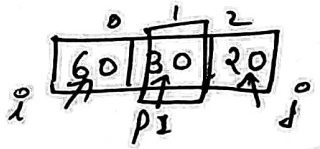
s == e return.

now our left tree is completed and ~~tree~~ array looks like this →



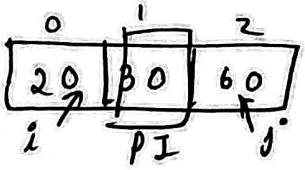
→ 

PI = 5
count = 1



60 < 30 → F, i = 0

swap(ar[0], ar[2])
20 > 30 → F, j = 2

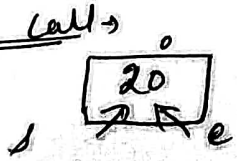


20 < 30 → i++, i = 1

60 > 30 → j--, j = 1

i = j = pivot Index → left call
↳ right call

<u>left call</u> →



<u>s == e</u> return

right call

$\boxed{60}$ s e s == e return

now right side of the array is also sorted, so now array looks like this →

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

And our array is sorted.

⊙ T.C → $O(n \log n)$ ← In avg and best case.

$O(n^2)$ ← In worst case

worst case is when array is reversly sorted, and best case is when array is already sorted.

⇒ Partition logic →

Array is passed two times :→

① To place pivot element into it's correct position.
⇒ $O(n)$

② To place ~~left &~~ smaller elements on left side and greater elements on right side of pivot.
⇒ $O(n)$

T.C ⇒ $O(2n)$ = $O(n)$ ← linear Time

⇒ Recursive calls -

In avg. case (when ~~or~~ recursive calls are being held for n/2 size array), then in recursive tree there will be a levels.

$a = \log n$

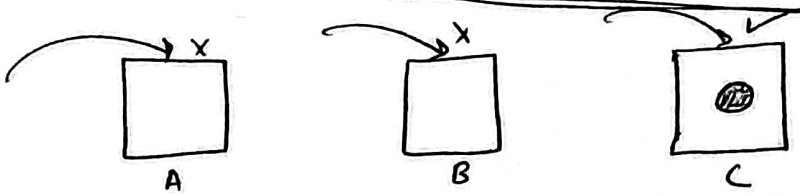
So T.C = $O(\log n)$

Total T.C of Quick Sort = $O(\underbrace{n}_{\text{partitioning logic}} \underbrace{\log n}_{\text{Recursive calls}})$.

Basically we have to perform partition logic $\log n$ times.

In worst case we perform partitioning logic n times, so that's why the T.C is $O(n^2)$.

BACKTRACKING → (specific form of recursion).



(explore all possible solutions) and do not check again a solution if you discarded it.

A problem solving technique, entirely based on recursion.

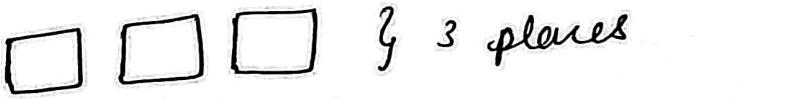
Permutations of a String →

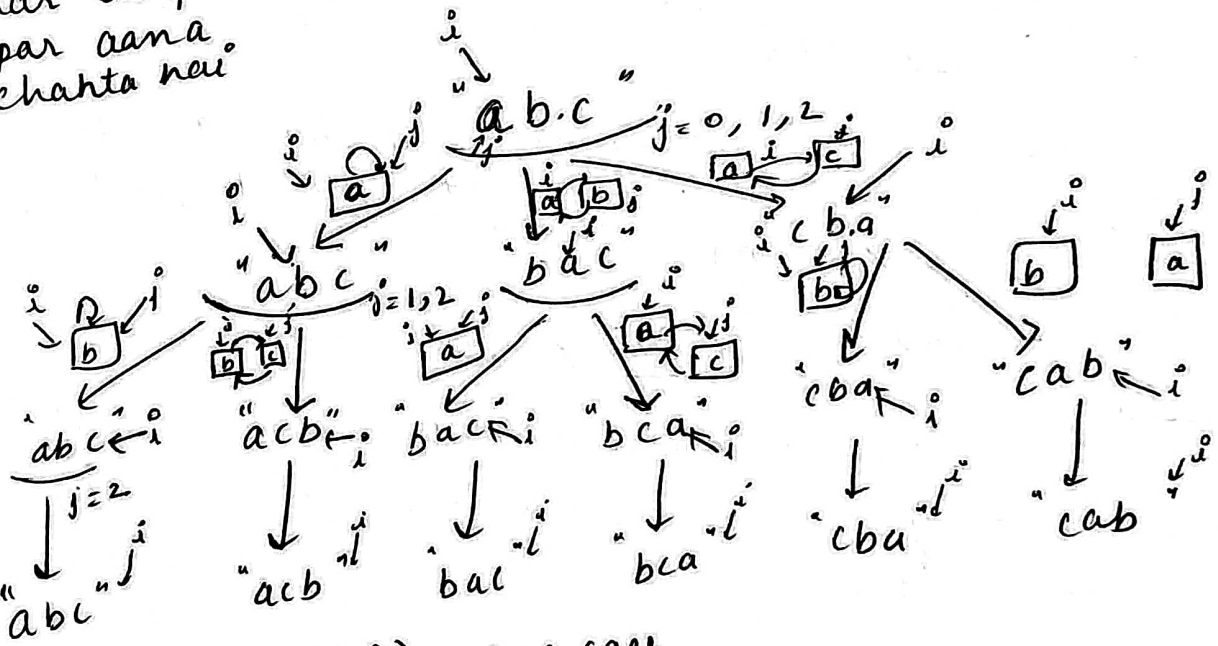
i/p → string str = "abc"
o/p → all permutations → abc, bac, bca, cab, cba, acb.

string → "xy"
o/p → xy, yx

string → 'a'
o/p → a

Har ek charater har ek place par aana chahta hai



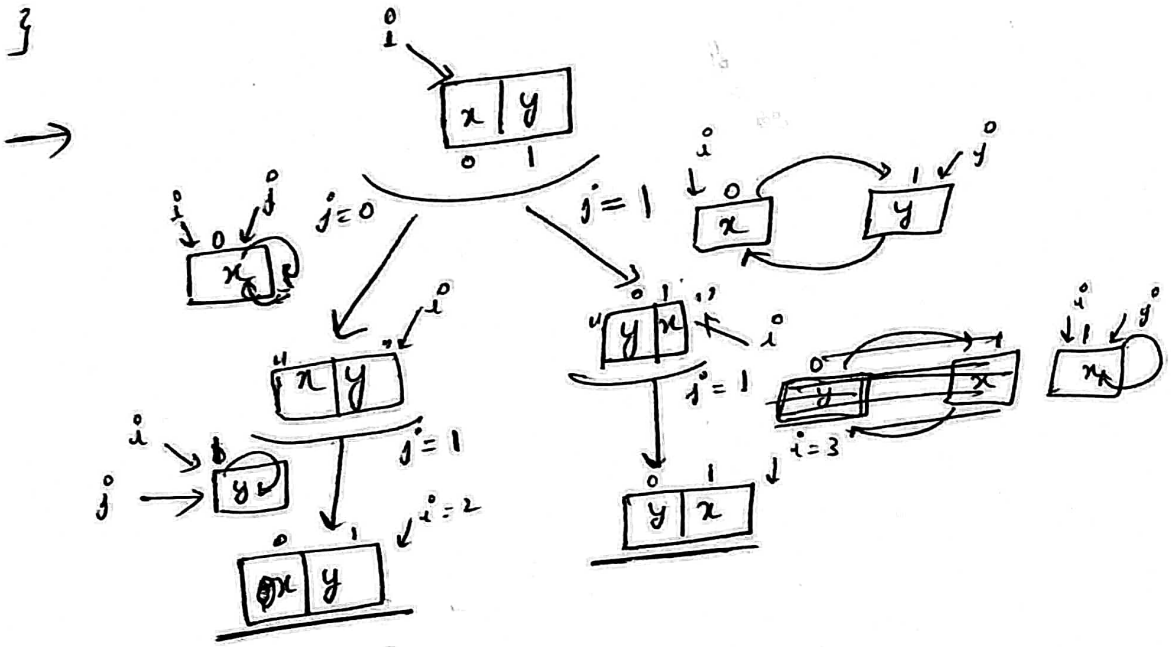


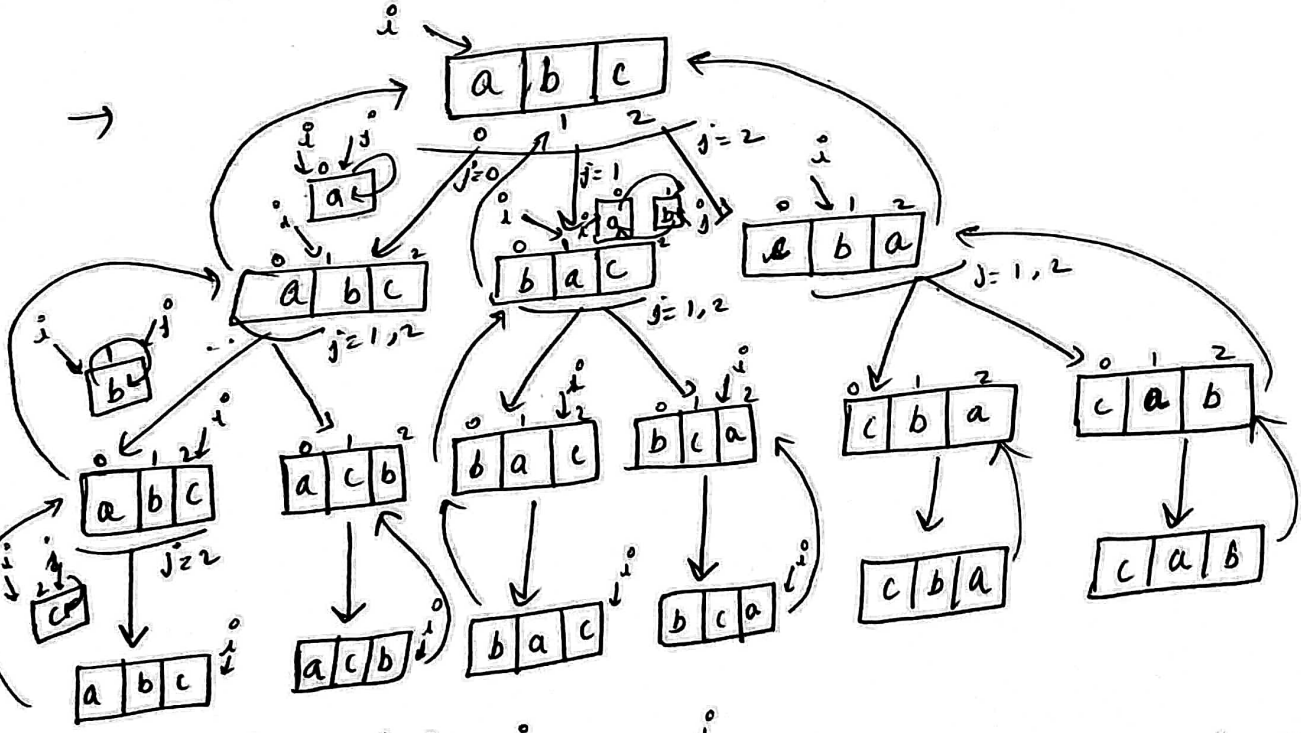
i == str.length() → Base case
→ print

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

```
//swapping
for(int j= i ; j< str.length(); j++){
    //swap
    swap(str[i], str[j]);
    //rec call
    printPermutations(str, i+1);
    //backtracking -> why? to recreate the original string.
    swap(str[i], str[j]);

}
}
```





Why backtrack? so we have a | b | c and we swapped ~~str~~ str[i] and str[j], so now our string is like this (because it is passed by reference) c | b | a

so when we return from the func call (whenever base cases matches) this altered string is passed to next recursive call (not the original one), so that's why we are backtracking (basically <u>undo</u> the changes we made) here.

→ so after backtrack the string

	0	1	2
	c	b	a

swap(str[i], str[j])

0	1	2
a	b	c

, we got the original string.

⊙ <u>T.C</u>→

for 1st → 3 recursive calls
for 2nd → 2 recursive calls
for 3rd → 1 recursive calls.

} ?!

$\boxed{\text{so } T.C = O(n!)}$, find out why?

• <u>Dry Run one more examples.</u>