Today's Content

- → Quick Sort - (Heavy)
- → Count Sort

Qn: Given N array elements. Rearrange array st

(i) ar[0] should go to its sorted posn.
(ii) All elements <= ar[0] should go to its left
(iii) All elements > ar[0] should go to its right.

	0	1	2	3	4	5	6	7	8	9	10
Eg: a[11]=	10	3	8	15	6	12	2	18	7	15	14
						10					

idea 1: Sort the array.

a[] = 2 3 6 7 8 | 10 | 12 14 15 15 18

TC: O(nlogn) Overkill?

idea 2: Count the no. of elements < a[0] & then swap a[0] to its correct place. Take another loop to swap elements.

TC: O(n+n) = O(n), SC: ??

idea 3: Use a single loop to do swapping

	0	1	2	3	4	5	6	7	8	9	10
a[11]=	10	3	8	15	6	12	2	18	7	15	14

	0	1	2	3	4	5	6	7	8	9	10
temp[11]=	3	8	6	2	7	10	14	15	18	12	15
						↑ p1	↑ p2				

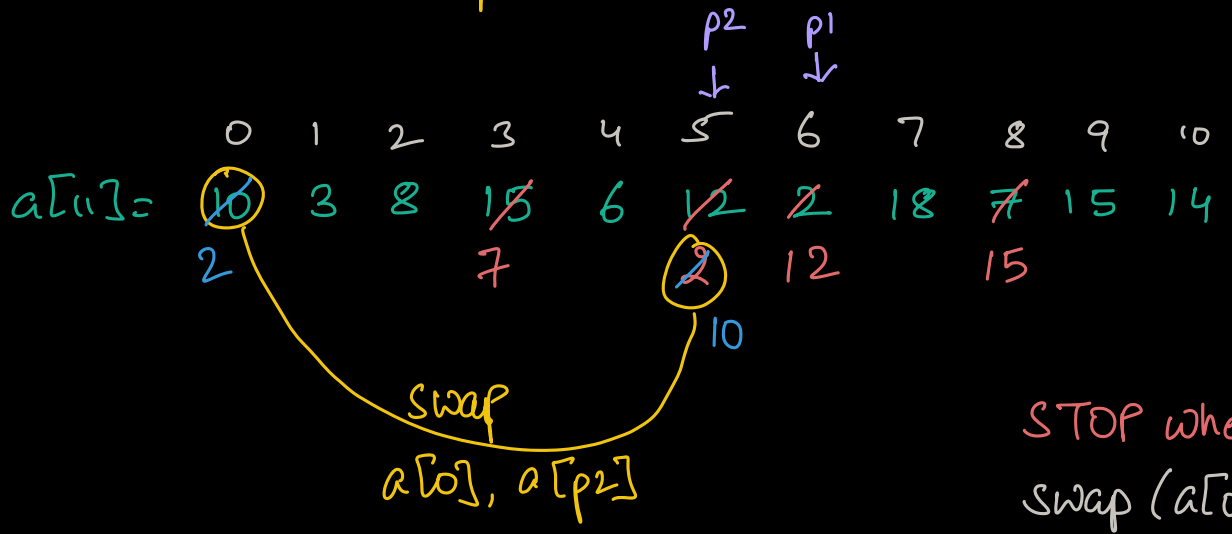
// Pseudo code :

```
void rearrange (int a[]) {
    temp[n]
    p1 = 0 , p2 = n-1
    for (i = 1; i < n; i++) {
        if ( a[i] <= a[0]) {
            temp[p1] = a[i]
            p1++
        }
        else {
            temp[p2] = a[i]
            p2--
        }
    }
    temp[p1] = a[0]
}
```

TC: O(n)

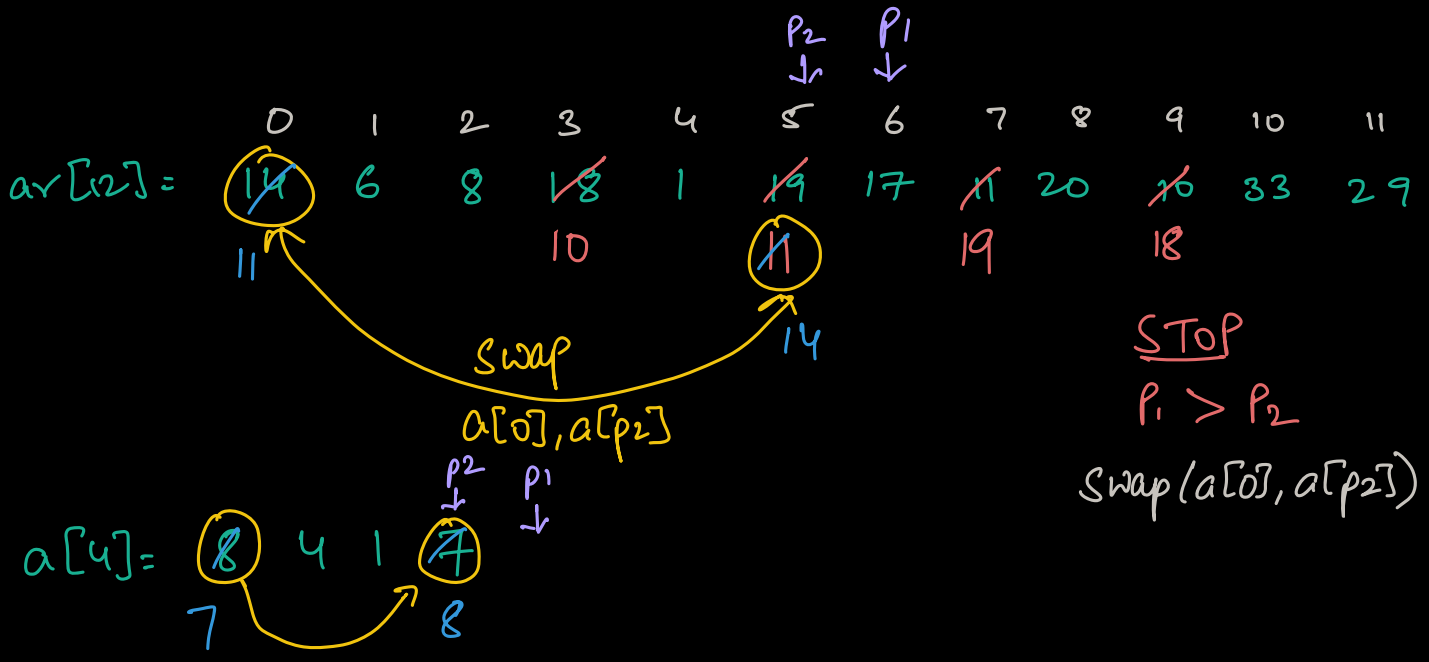
SC: O(n)

Do it in const space



STOP when $p_1 > p_2$

swap (a[0], a[p2])




```
void rearrange(int a[]) {
    p1 = 1, p2 = n-1
    while (p1 <= p2) {
        if (a[0] >= a[p1]) { p1++ }
        else if (a[0] < a[p2]) { p2-- }
        else {
            swap(a[p1], a[p2])
            p1++, p2--
        }
    }
    swap(a[0], a[p2])
}
```

TC: O(n)

SC: O(1)

Qn: Given a[N] & subarray[s e].
Rearrange subarray [s e] st. a[s] should be in correct posn of subarray, & return the idx of correct posn.

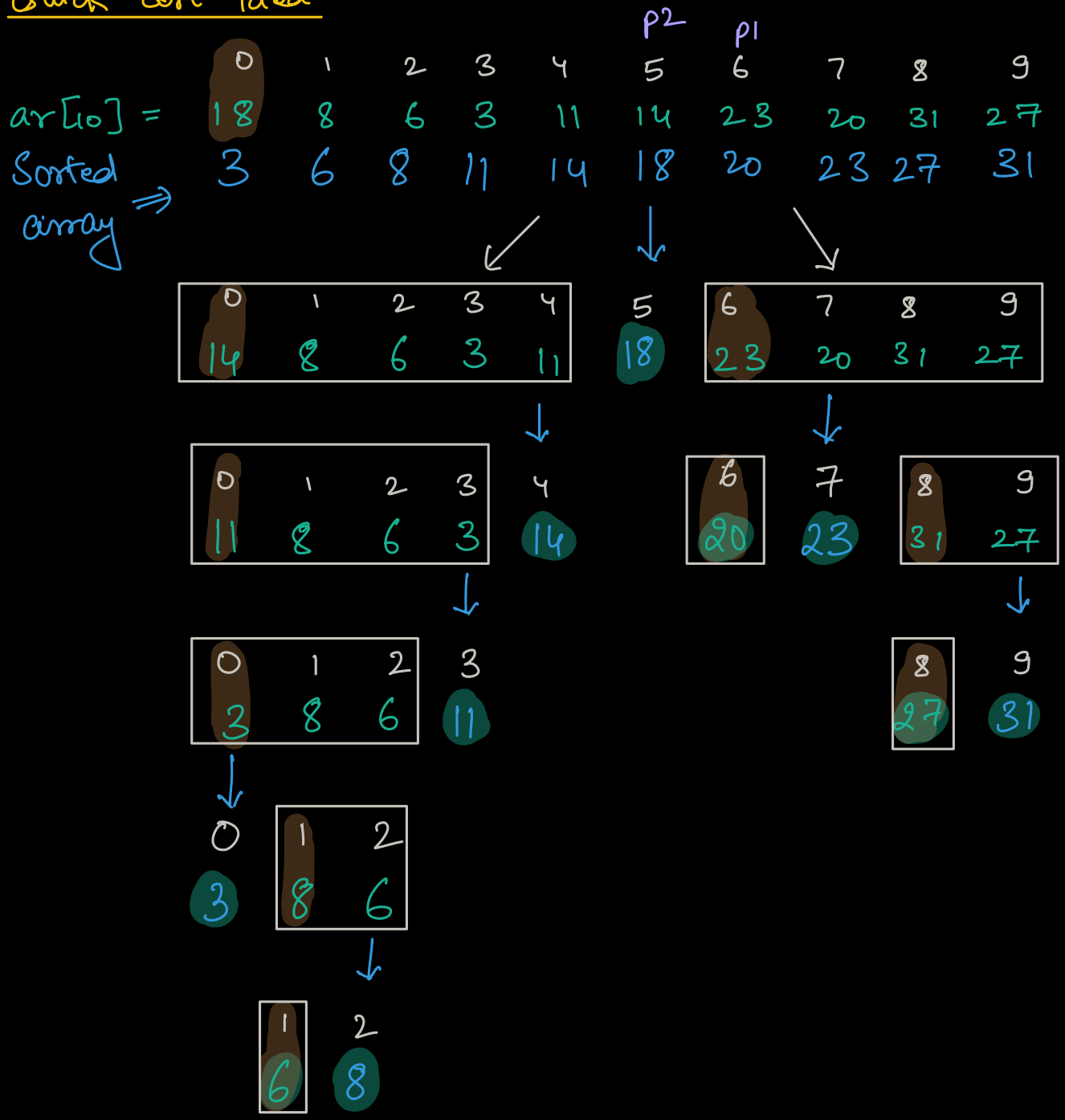
	0	1	[2	3	4	5	6	7]	8	9
Eg:-	10	3	[8	6	14	7	4	12]	7	1

[s e] → [2 7]

```
int rearrange(int a[], int s, int e){
    p1 = s+1 , p2 = e
    while(p1 <= p2){
        if(a[s] >= a[p1]){ p1++ }
        else if(a[s] < a[p2]){ p2-- }
        else {
            swap(a[p1], a[p2])
            p1++, p2--
        }
    }
    swap(a[s], a[p2])
    return p2
}
```

TC = O(n)
SC: O(1)

Quick Sort Idea




```
void quickSort(int a[], int s, int e) {
    if (s >= e) { return }
    p = rearrange(a, s, e)
    quickSort(a, s, p-1)
    quickSort(a, p+1, e)
}
```

s=1 e=2

1 2

8 6

2

qs(a, 1, 1)

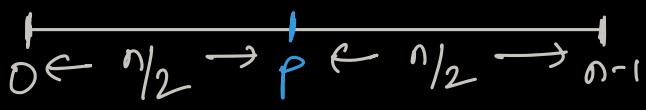
qs(a, 3, 2)

s>e ??

Interesting Observation :

Quick Sort	v/s	Merge Sort
① Do the job		① Do recursion
② Do recursion		② Do the job
job: pre order		job: post order

Time Complexity part :

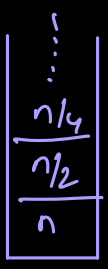


Best case: The chosen/pivot comes in middle.

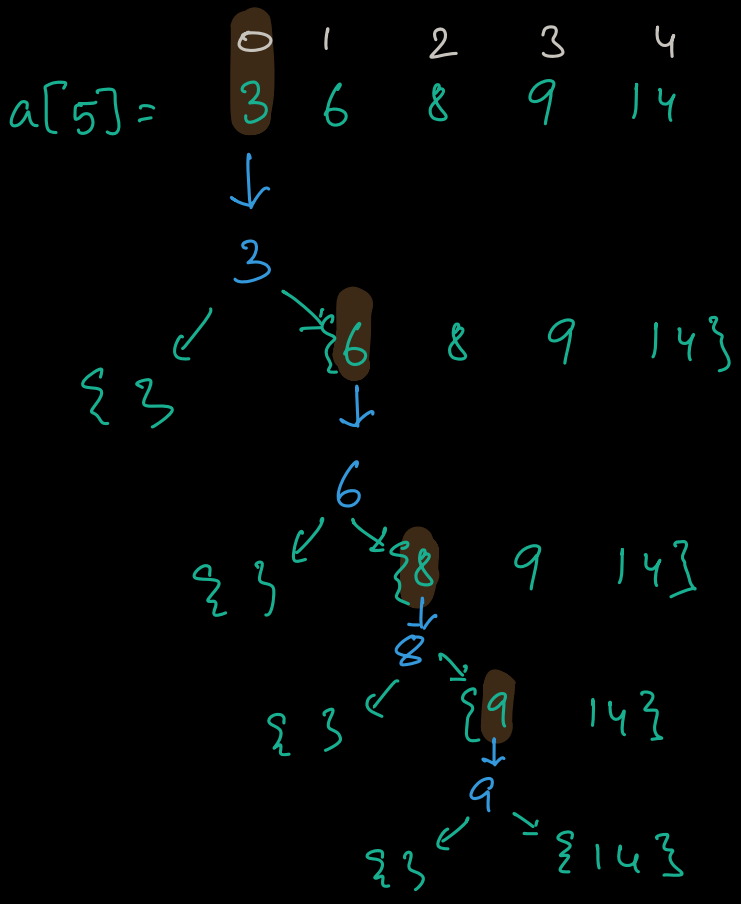
$$T(n) = n + T(n/2) + T(n/2)$$

$$T(n) = 2T(n/2) + n$$

TC: $O(n \log n)$, SC: $O(\log n)$

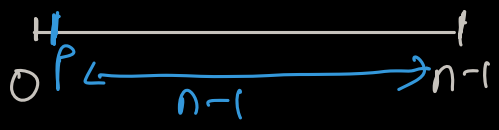


Worst case:



In worst case, the chosen element is smallest / largest element & ends up at extreme ends.

$$T(n) = T(n-1) + n$$



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

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

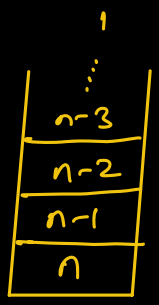
$$T(n-2) = T(n-3) + n-2$$

$$T(n) = T(n-3) + n + n-1 + n-2 \qquad T(1) = 1$$

$$\Downarrow$$

$$T(n) = T(1) + n + n-1 + n-2 + n-3 + \dots + 2$$

$$T(n) = O(n^2) \qquad SC: O(n)$$



a[]: 15 10 8 6 4 2

15

10

Worst case
TC: $O(n^2)$

a[]: 3 15 10 6 8

3

15

10

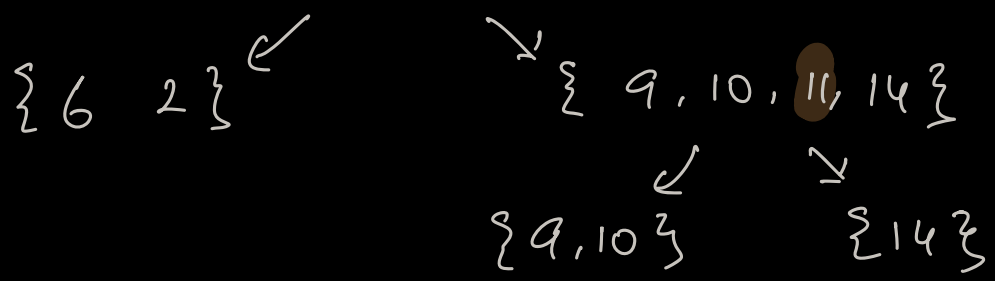
6

ref: min - max element

At every rearrange we pick min or max as reference. ⇒ WORST CASE.

∴ Instead of choosing a fixed reference point, make a random selection. to avoid WORST CASE SCENARIO

Eg: a[]: 9 6 8 2 10 11 14



Randomized QuickSort ⟶ TC: avg case: O(nlogn)
Worst case: O(n²)

```
int rearrange(int a[], int s, int e){
    int r = rand(s,e)
    swap(a[s], a[r])
  p1 = s+1 , p2 = e
  while(p1 <= p2){
    if(a[s] >= a[p1]){ p1++ }
    else if(a[s] < a[p2]){ p2-- }
    else {
        swap(a[p1], a[p2])        TC: O(n)
        p1++, p2--                SC: O(1)
    }
  }
  swap(a[s], a[p2])
  return p2
}
```

Count Sort

Qn: Given a[N]. Every element is in range [1-4].
↳ {1,2,3,4}
Sort the array.

	0	1	2	3	4	5	6	7	8	9
ar[10] = {	3	1	4	4	2	1	3	3	2	1 }

idea: Arrays.sort → TC: O(n log n)

idea2: Use a Hashmap <element, freq>
<1:3> <2:2>
<3:3> <4:2>

```
void countSort(int a[]) {
    // Store the frequency in a Hashmap (hm) — n
    K=0
    for(i=1; i<=4; i++) {                    } → n
        int c = hm[i]                        }
        for(j=1; j<=c; j++) {                }
            a[k] = i
            K++
        }
    }
}
```

TC: O(n)
SC: O(4) → O(1)

```
for(i=0; i<n; i++) {
    for(j=0; j<n; j++) {
        print(i+j)
    }
}
```

This effort by i should be considered.

i	j	iter
0	[0 n-1]	n
1	[0 n-1]	n
2	[0 n-1]	n
⋮	⋮	⋮
n-1	[0 n-1]	n
		n^2

$\Rightarrow n + n^2 = O(n^2)$

Qn: Given A[N]. Every element is in range [a-b]. Sort the array.

$a = \min(A)$

$b = \max(A)$

```
void countSort(int a[]) {
    // Store the frequency in a Hashmap (hm)
    K=0
    for(i=a; i<=b; i++) {
        int c = hm[i]
        for(j=1; j<=c; j++) {
            a[k] = i
            k++
        }
    }
}
```

i	j	iter
a	—	—
a+1	—	—
a+2	—	—
⋮	—	—
b	—	—
		n

$TC: O(n + n + (b-a+1))$

$= O(n + R)$

TC: $O(n+R)$

if $R <= n$	$n < R <= n\log n$	$R > n\log n$
TC: $O(n+n)$	TC: $O(n+n\log n)$	$R = n^2 / n^3 / 2^n / n!$ etc.
TC: $O(n)$	TC: $O(n\log n)$	Count Sort ✗
Use count sort	Use count sort or any sorting algo of $(n\log n)$	Use any sorting algo of $O(n\log n)$

$R = [1 \quad 100000]$

$a = \{1, 2, 3, 4, 5, 6\}$

Doubts

1. Selection – min
2. Bubble – compare adj. elements
3. Insertion – insert into sorted data
4. Merge – divide + merge
5. Quick – partition + divide
6. Count – frequencies in a range.