Bubble Sort

Selection Sort

Insertion Sort

$\left.\right\}$ $\underline{\underline{O(N^2)}}$ → any Range $-\infty$ to ∞

$\underline{N \to 10^4}$

$N^2 \to 10^8$ itr $= \boxed{1s}$

$N = 10^5$

$N^2 = 10^{10}$ itr

Time ↗ $\boxed{100s}$

↑

TLE

Counting Sort $\}$ $O(N + \text{Range})$

$N = 5$

$\underrightarrow{1,\ 2,\ \ 8,\ 6,\ 25}$

$N=5$

Range $= N^2$

$O(N + N^2)$ → Not useful

$-\infty$. - - - - - - ∞ → Not practical for such cases.

Agenda

- Merge Sort O(N log N)
- Quick sort O(N log N) , O(N^2)
  Avg case Worst Case

(Randomized Quicksort → O(N log N)

Arrays.sort(----)
↓
N log N

MERGE SORT

↳ 'Break & Make'

↳ Divide & Conqre Algorithm

↳ 3 Simple Steps

arr

3	6	2	1	7	5

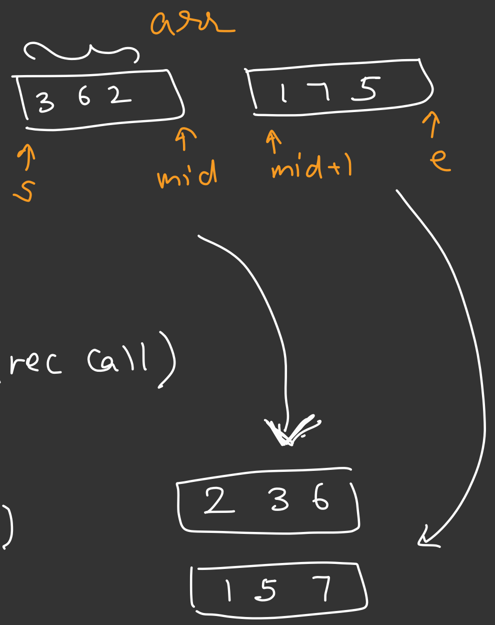
in ↑ e

1	2	3	5	6	7

↑
Output

MergeSort(arr, s, e)

① Divide the array into 2 parts.



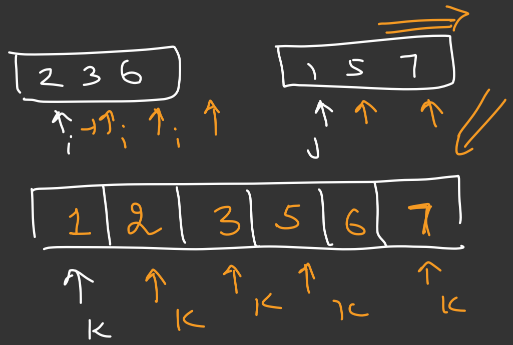
Assuming Merge Sort works
↓

② Run MergeSort on Smaller Arrays (rec call)

```
MergeSort(arr, s, mid)
MergeSort(arr, mid+1, e)
```

③ 'Merge' the two parts

(two pointer)



Code

```
MergeSort (A, s, e) {
      // Base Case
      if ( s >= e ) {
            return,
      }
      // Rec Case
(1)   mid = (s+e)/2;
(2)   mergeSort(A, s, mid);
      mergeSort(A, mid+1, e);
(3)   merge(A, s, mid, e);
}
```

write as a Separate method

s

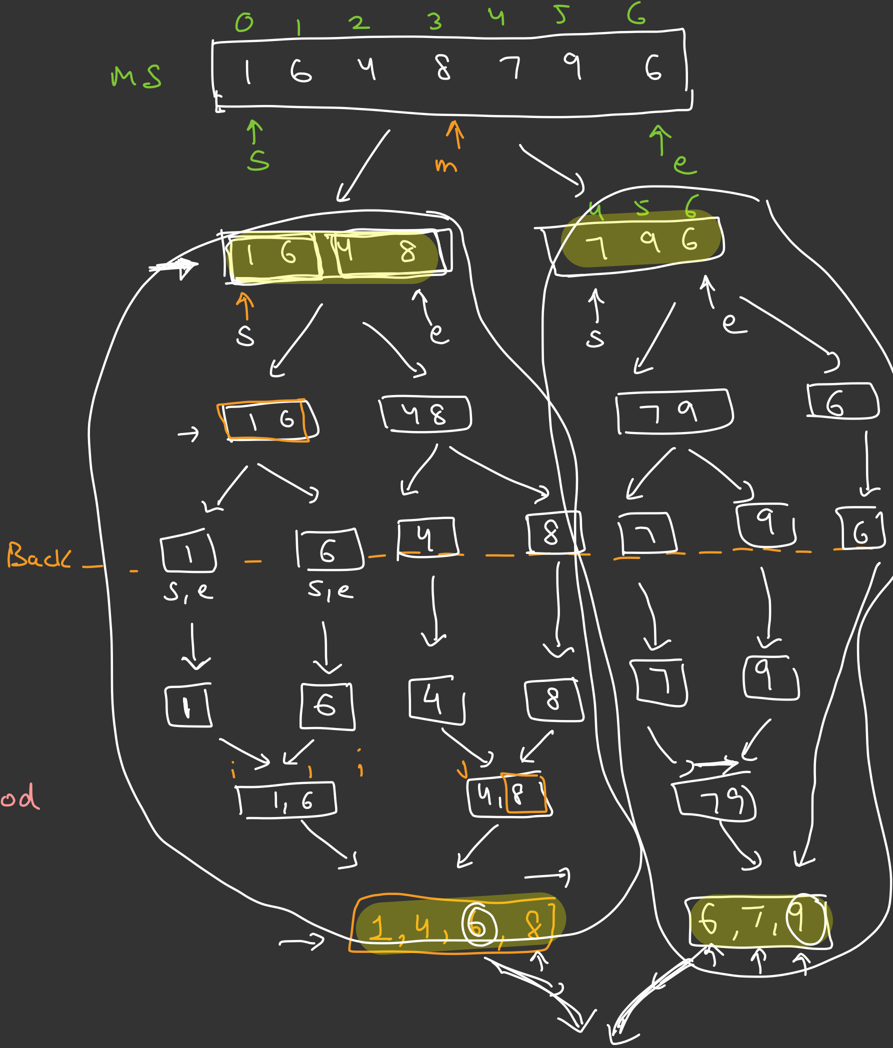
e

s == e   s > e

1 6 4 8 7 9 3

```
Merge Sort (A, s, e) {
        // Base Case
        if (s >= e) {

              return;

        }

        // Rec Case
  ①     mid = (s+e)/2;
  ②     mergeSort(A, s, mid);
        mergeSort(A, mid+1, e);

  ③     merge(A, s, mid, e);
}
```

write as a Separate method



Code ⇒ merge Fn

Time Complexity ?

1, 4, 6, 6, 7, 8, 9

final output

```
Merge Sort (A, s, e) {
        // Base Case
  L0    if ( s >= e ) {
              return;
        }
        // Rec Case
① L1    mid = (s+e)/2;
② L2    mergeSort(A, s, mid);
  L3    mergeSort(A, mid+1, e);
③ L4    merge(A, s, mid, e);
}
```

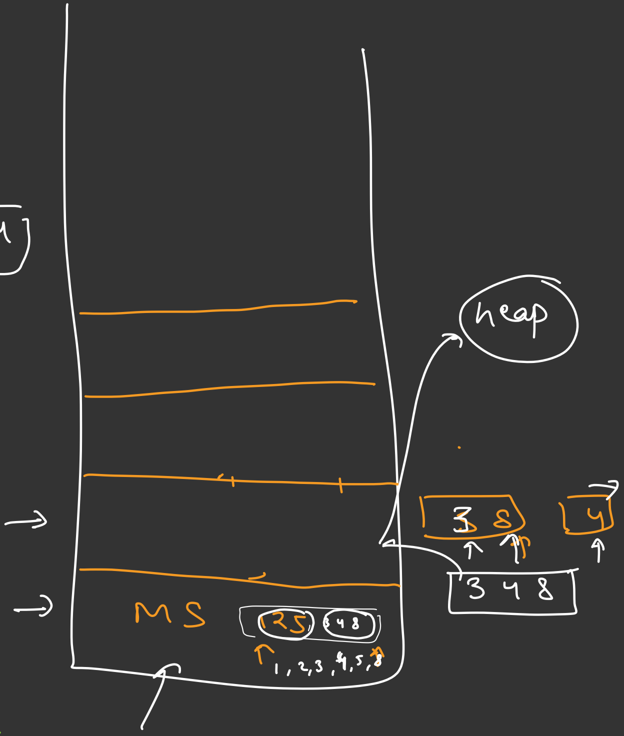
Calls are over

Not a Rec over

only 1 fn in call Stack

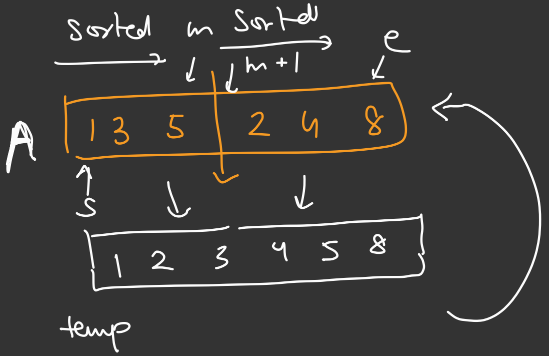
write as a Separate method

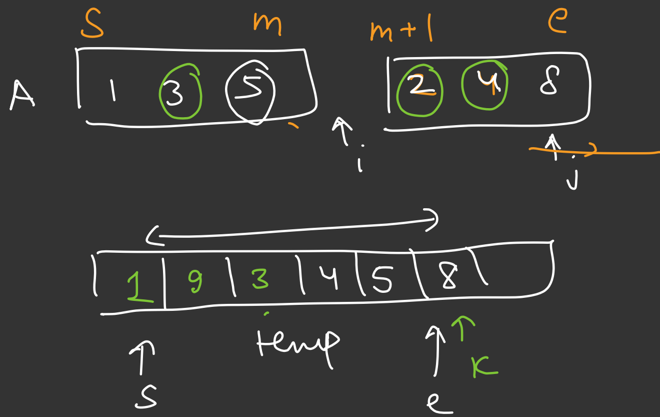
2, 1, 5, 8, 3, 4



```
merge (A, s, m , e) {

    temp[ Arr.length ];

    i=s
    j = m+1
    k = s

    while ( i<=m && j<=e ) {
        if (a(i) <= a(j)) {
            temp[k] = arr[i];
            i++;     k++
        }
        else{
            temp [k] = arr[j]
            j++
```

Stable [✓ yes]





```
            k++
        }
    }
```

✓

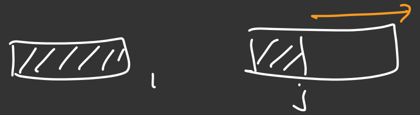
```
// Copy Remaing elements
while(j<=e){
    temp[k] = arr[j]
    j++ , k++
}
```

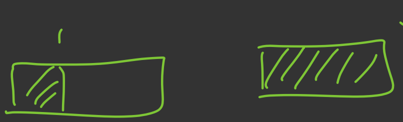
```
// Copy Remaing
while(i<=m){
    temp(k) = arr(i),
    i++ , k++
}
```

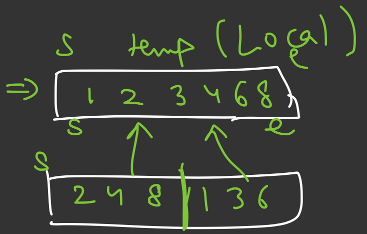
One of these will execute

```
for(int i=s. i<= e; i++){
    Arr[i] = temp[i],
}
```

Copy to original

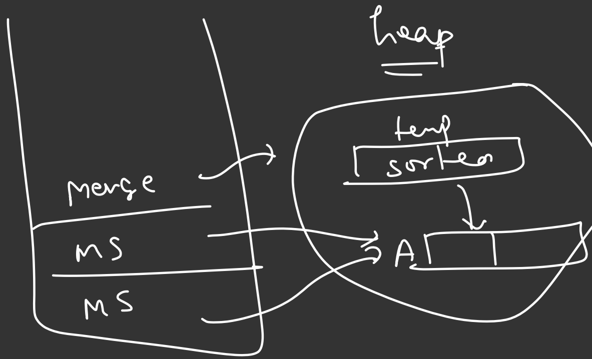






}

}

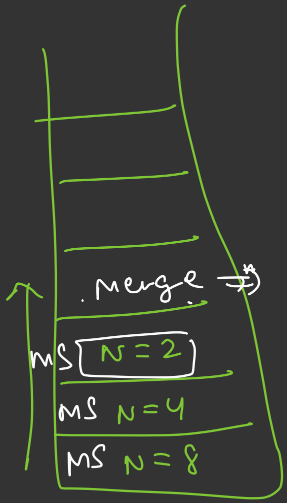


Time & Space

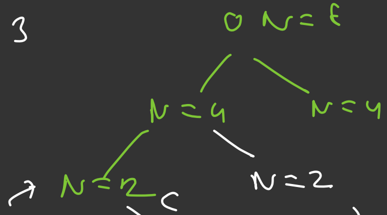
Space

$O(\log N + N)$

$= O(N)$



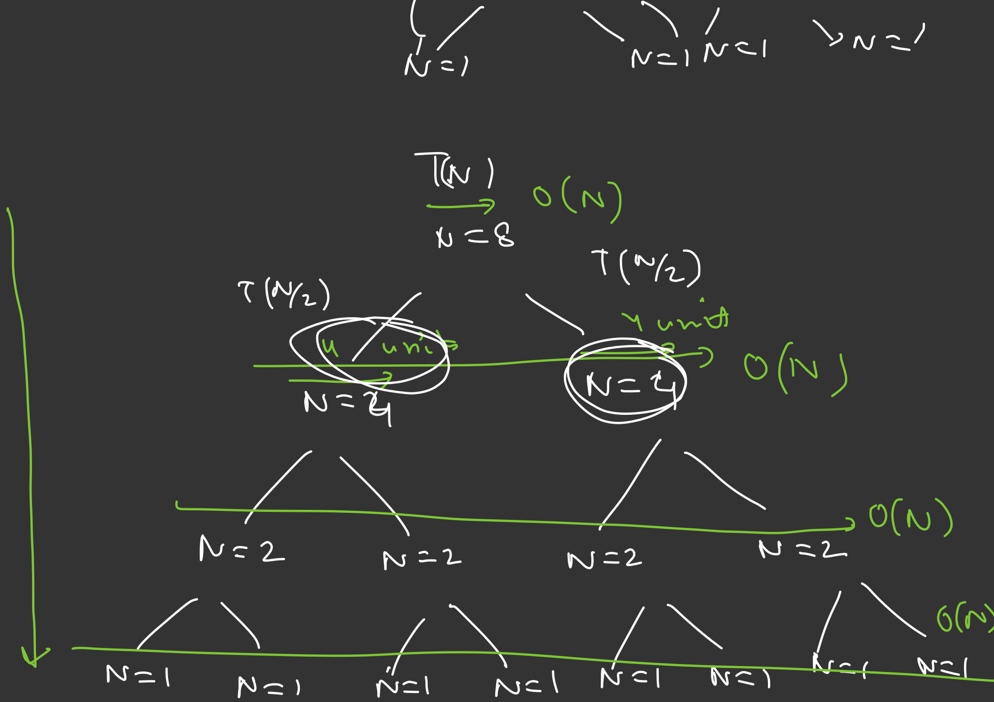
```
main() {
    arr=[ ]
    mergesort(arr, 0, n-1);
    print(arr);
}
```



Time

Visual way

$$O(N \underset{\text{Levels}}{\boxed{\log N}})$$



Recurrence way

$$T(N) = \overset{\text{Mid}}{k} + \overbrace{T\left(\frac{N}{2}\right) + T\left(\frac{N}{2}\right)} + \overbrace{kn}^{\text{merge}}$$

$$T(N) = \underset{\nearrow}{\overset{\text{Mid}}{k}} + \underbrace{\overbrace{T\left(\frac{N}{2}\right) + T\left(\frac{N}{2}\right)}} + \overset{\text{merge}}{\overbrace{kn}}$$

Break

10:40

$$\begin{array}{llllll}
N & \boxed{T(N)} & = & \cancel{2T\left(\frac{N}{2}\right)} & + kn & \\
\downarrow\ \frac{N}{2} & \cancel{2T\left(\frac{N}{2}\right)} & = & \cancel{4T\left(\frac{N}{4}\right)} & + \cancel{2}\frac{kn}{\cancel{2}} & \times 2 \\
\downarrow\ \frac{N}{4} & \cancel{4T\left(\frac{N}{4}\right)} & = & \cancel{8T\left(\frac{N}{8}\right)} & + \cancel{4}\frac{kn}{\cancel{4}} & \times 4 \\
\vdots & \vdots & & \vdots & \vdots & \\
T(1) & \cancel{\ } & & & &
\end{array}$$

$$T(N) = kn + kn + kn + \cdots\cdots$$

$$= \sum^{\log N} kn = kn \log n = O(n \log n)$$

10.40

QUICK SORT

→ Divide & Conquer

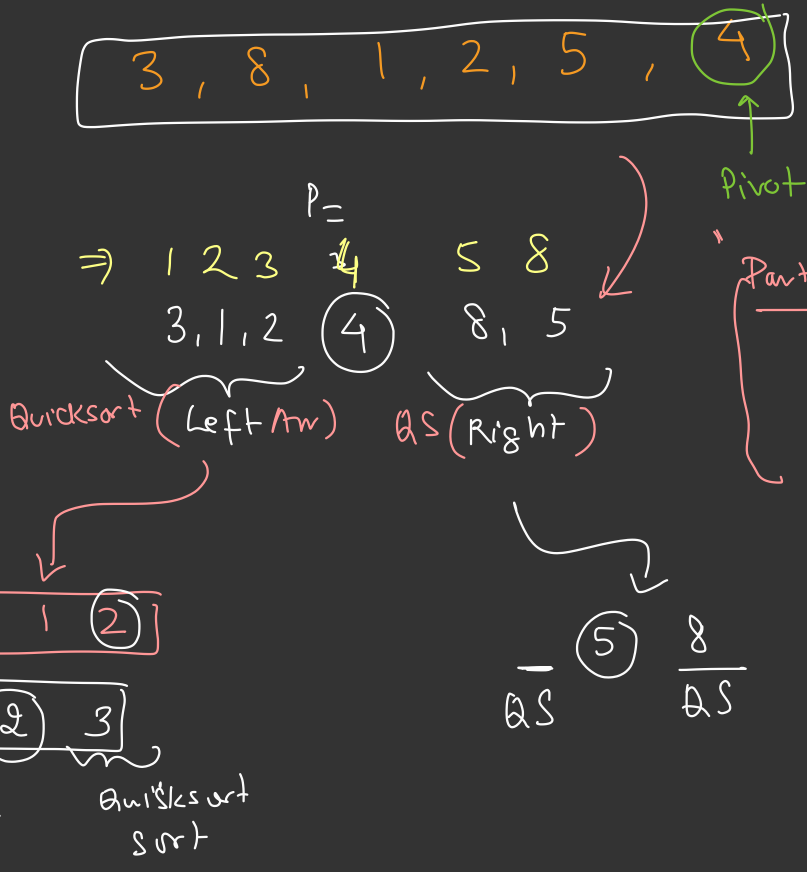
→ Tries to partition the array around a pivot element & recursively Sort the two parts.

↳ last elemnt OR

↳ first element OR

↳ random element

high level idea

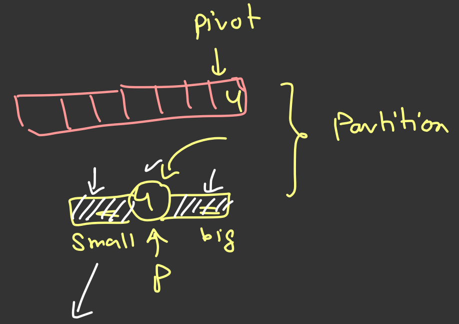


"Partitioned" st that [pivot is at correct location.]

Code

Done

```
void quicksort( arr[] , int s, int e ) {

    // Base Case

    if( s >= e) {

        return,

    }

    // Rec Case

 →  p = partition (arr, s, e);

    quicksort ( arr, s, p-1),
    quicksort (arr, p+1, e);

}
```



⇒ s to p-1 (p)

⇒ p+1 to e

CLRS (Book) way

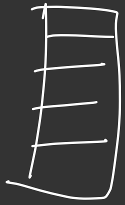
```
int partition(int arr[], int s, int e){

    pivot =arr[e]

    int i= s-1

    for(j=s; j<=e-1; j++){

        if(arr[j] <= pivot){
            i++
            swap(a[i],a[j]);

        }

    }

Swap(arr[e],arr[i+1]);

return i+1;
```

Pivot's positioning

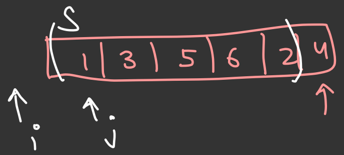
DRY Run

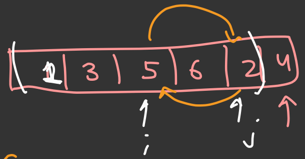
$O(N)$

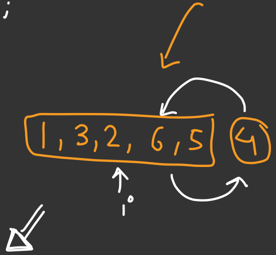
4



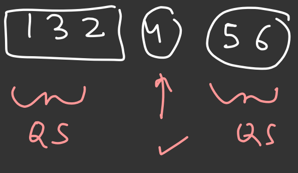
3



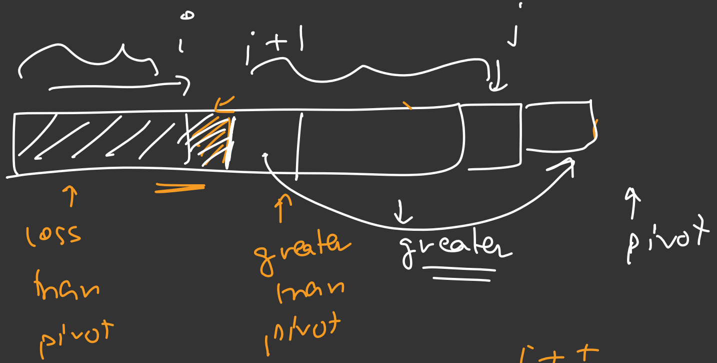




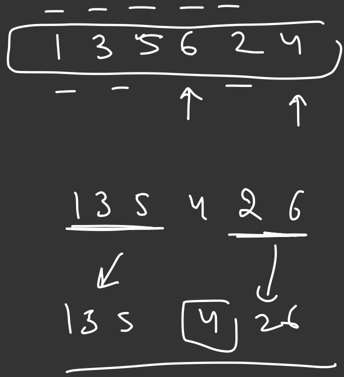
1<4	i++
3<4	i++
5<4	×
6<4	×
2<4	✓



Intutionely



```
i++
swap(a(i), a(j))
```



Time

Partitioning

$$T(n) = KN + T(p-1) + T(N-p)$$

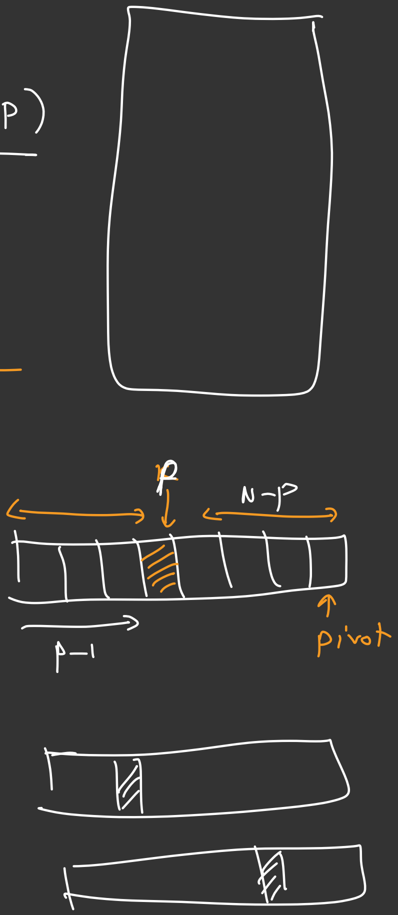
p → non-determined

Appx ⇒ on avg we assume p → n/2

$$= kn + T\left(\frac{n}{2}\right) + T(n/2)$$

$$T(n) = kn + 2T\left(\frac{n}{2}\right)$$

$$= O(n\log n)$$ in average case.



Worst Case

$N-1 + N-2 + N-3 + ---$

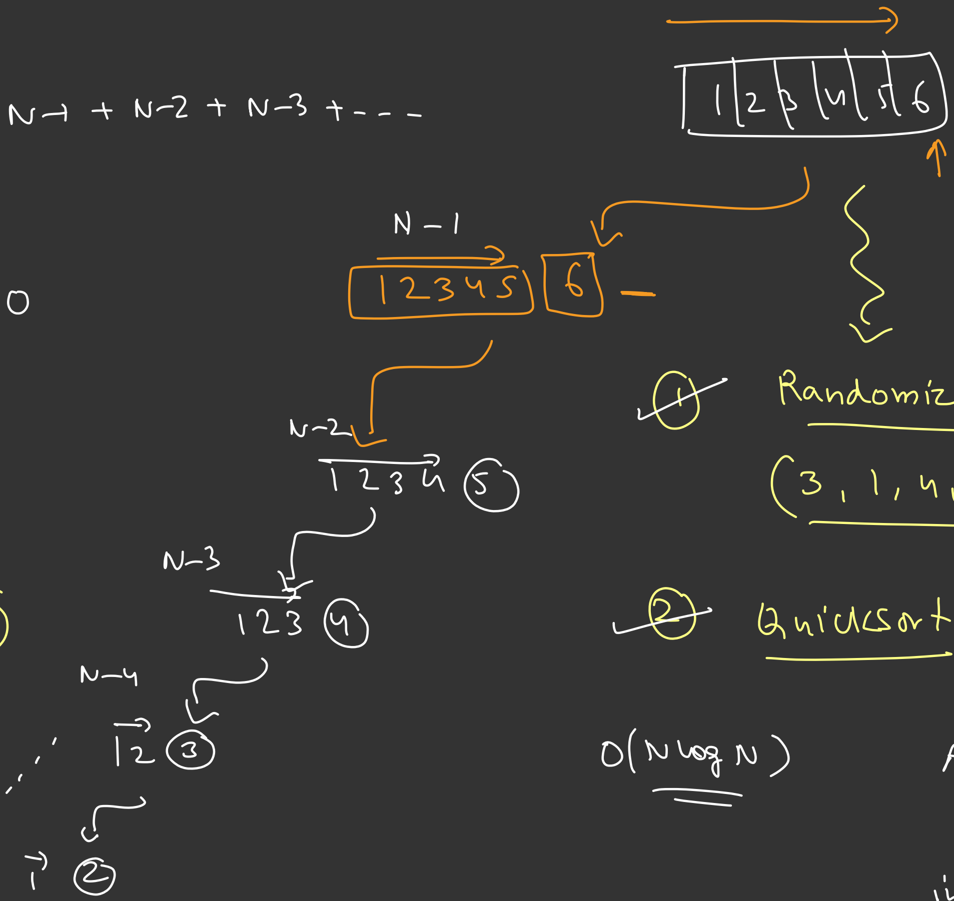
$= O(N^2)$

$\boxed{\frac{1}{N!}} \longrightarrow$ close to 0

N! arrangement

1, 2, 3
3, 2, 1
3, 1, 2
1, 3, 2
2, 1, 3
2, 3, 1

$\Big\}$ $3! = \boxed{6}$



~~①~~ **Randomize Array**

(3, 1, 4, 6, 5, 2)

N

~~②~~ **Quicksort**

O(N log N)

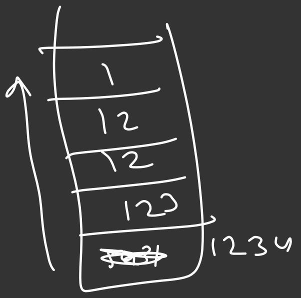
Avoid the worst case in practical Sce

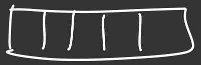
Space

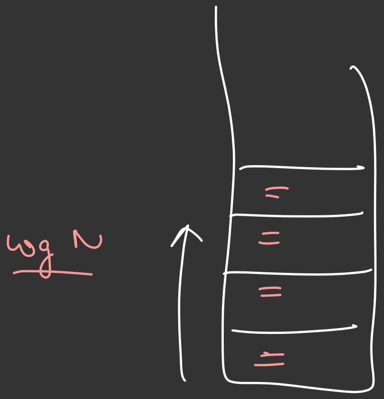
Extra space

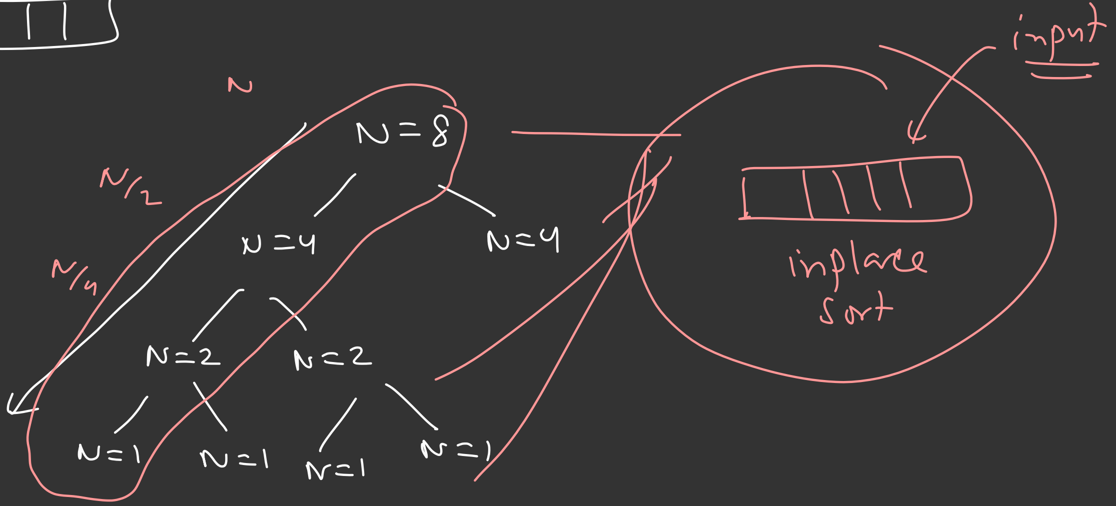
Avg Case O(log N)

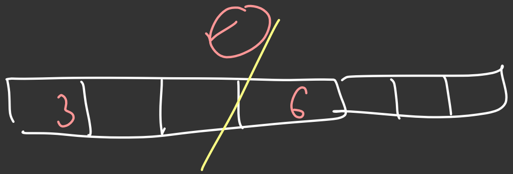
worst Case O(N)











Log $\frac{N}{2}$

Log

Arrays.sort in Java

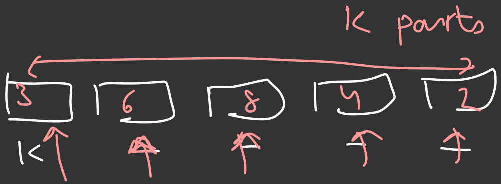
↓

"Quicksort"

↓

Less overhead of copying

faster



merging ↓ slower