Bubble Sort Algorithm

* Sorting

  eg- given - 5 4 2 1

  1 2 4 5 Increasing order

  5 4 2 1 Decreasing order

* Applications -
  * contact list (Alphabetically sorted)
  * Shopping website (If we want to see prices from low to high we have a filter i.e Low to high) same for high to low.

* Bubble Sort Algorithm

  → we do a certain no. of passes & in every pass, we compare adjacent elements & swap them if they are not in correct order.

eg- arr[] = {5, 7, 4, 3, 1}; n = 5

o/p - 1 3 4 5 7

(For increasing order the current element must be less than the next adjacent element.)

5 7 4 3 1

Iteration 1 → 5 4 3 1 7

Iteration 2 → 4 3 1 5 7

Iteration 3 → 3 1 4 5 7

Iteration 4 → 1 3 4 5 7

(After 1st Iteration the largest element will come at last similarly for 2nd Iteration the 2nd largest element will come..)

* We sorted the Array in (n-1) total passes

→ In every Iteration, the largest number in part of array to be processed gets its correct position.

IH

* we are iterating outer loop for no. of passes | iterations. i.e from 0 to n-1. (n-1 iterations means 1 element less than the total no of elements. If we sort the n-1 elements the last element will already be sorted in required order.

* we are iterating the inner loop from 0 to n-i-1. (because last i elements are already sorted at correct positions so need to check them.)

```
* for(int i=0; i<n-1; i++){
      for(int j=0; j<n-i-1; j++){
          if(a[j] > a[j+1]){
              int temp = a[j];
              a[j] = a[j+1];
              a[j+1] = temp;
          }
      }
  }
```

n = 5

a

5	4	3	2	1
0	1	2	3	4

Iteration No					
i=0	4	3	2	1	5	5-0-1=4 (n-i-1)
i=1	3	2	1	4	5	5-1-1=3
i=2	2	1	3	4	5	5-2-1=2
i=3	1	2	3	4	5	5-3-1=1

Time and Space Complexity

* Space complexity → O(1) (we are changing the elements with in the Array only so no space is used) only constant variables are used.

* Time complexity

for i times how many times the j loop runs

i → 0 1 2 . . . n-2

j → n-1 n-2 n-3 . . . 1

→ n-(n-2)-1 = n-n+2-1 = 1

(n-1) + (n-2) + (n-3) + . . 1 → n-(n-1)

n(n-1) - (1 + 2 + . . n-1)

$n^2 - n - \left(\frac{(n-1)n}{2}\right)$

$\frac{2n^2 - 2n - n^2 + n}{2} = \frac{n^2 - n}{2} = \frac{n^2}{2} - \frac{n}{2} = \frac{n^2}{2} \simeq n^2$

T.C → $O(n^2)$ worst case, best case, Average case

* Maximum no. of swaps in worst case in Bubble sort

worst case → 7 6 5 4 3

No. of swaps are for i=0, n-1
i=1, n-2
i=2, n-3
i=n-2, 1

= (n-1) + (n-2) + (n-3) + . . 1

= $\frac{n(n-1)}{2} \simeq n^2$

* How to optimize the bubble sort in the case of nearly sorted Arrays

eg - 2 1 3 4 5

	1	2	3	4	5
i=0					
no swaps after i=0					
i=1	1	2	3	4	5
i=2	1	2	3	4	5
i=3	1	2	3	4	5

* If the array is sorted we dont do any swaps. If we dont do any swaps means the arr is sorted.
* check code in Vscode (boolean flag = false → If swaps has happened flag will change to true → Atleast we check the flag value if it is false we will return we will not do more iterations)
* Time complexity for optimized approach is
  worst case → $O(n^2)$
  best case → $O(n)$ (we will iterate the j loop for n times but for i loop we will not iterate) so $O(n)$

* stable & unstable Sort (for every Sorting Algorithm we can say it is stable/unstable sort) i.e
  Stable - order of appearence of duplicate elements is same in the sorted Array as given in original Array
  unstable - is different eg. 5 4 3 2 3*
  Stable - 2 3 3* 4 5
  Unstable - 2 3* 3 4 5.

* Is bubble sort stable?
  Yes, because we dont consider equality comparisions i.e if (a[j] > a[j+1])
  If it is strictly greater than only we swap or else we dont swap so order of duplicate elements will not change

* Is bubble sort In-place Algorithm?
  In-place Algorithm means changes in same Array rather taking extra space. So, Yes it is Inplace Algorithm.

Bubble Sort -	Best case	worst case	Average case
	$O(n)$	$O(n^2)$	$O(n^2)$

Selection Sort Algorithm

* Selects an element and puts it at its correct position

eg - Not in place (using extra space i.e ans)

arr	3	5	4	1	3
	0	1	2	3	4

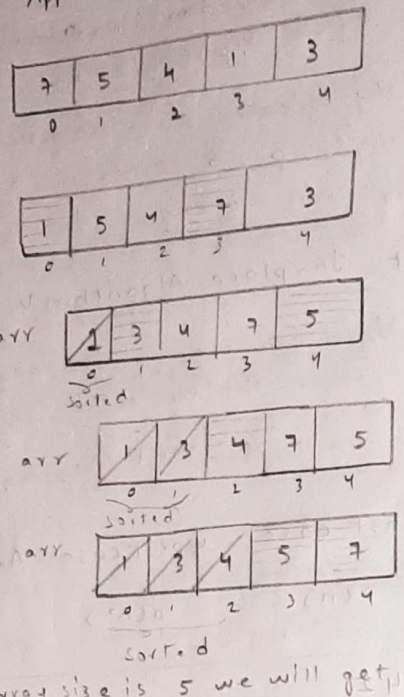
o/p - 1 3 4 5 7

→

ans	1	3	4	5	7
	0	1	2	3	4

* In the given array find the minimum element and put that minimum element at index 0 in ans
* The minimum element that is found replace that with max value Integer.MAX_VALUE and again find the minimum element in that array and put that element in ans one by one. and so on.

* In place Approach (In same Array)



* If the array size is 5 we will get the sorted Array in n-2 Iterations which is 3 because If you put n-2 elements In correct position the remaining element will also be in correct position. (Indexing is from 0 so n-2)

```
* static void SelectionSort(int[] arr) {
    int n = arr.length;
    for(int i=0; i<n-1; i++) {  // i represent the current Index
        // Find the minimum element
        int minIndex = i;
        for(int j=i+1; j<n; j++) {
            if(arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        if(minIndex != i) {
            int temp = arr[i];
            arr[i] = arr[minIndex];
            arr[minIndex] = temp;
        }
    }
}
```

* Find the minimum element In the Array and replace It with current Index

* Selection Sort Time and Space Complexity

→ Space complexity → O(1) (no extra array used to store the elements)

→ Time complexity → $O(n^2)$ (worst, best, Average cases)

$$(n-1)c1 + (n-1)c3 + \frac{n(n-1)}{2}c2 \approx n^2$$

* Selection sort is not stable

eg - 4 10 4* 2
0 1 2 3

If i=0 - 2 10 4* 4 (first minimum element is considered)

If 1 - 2 4* 10 4

If 2 - 2 4* 4 10

* Selection sort is In-place Algorithm because It is not using extra space

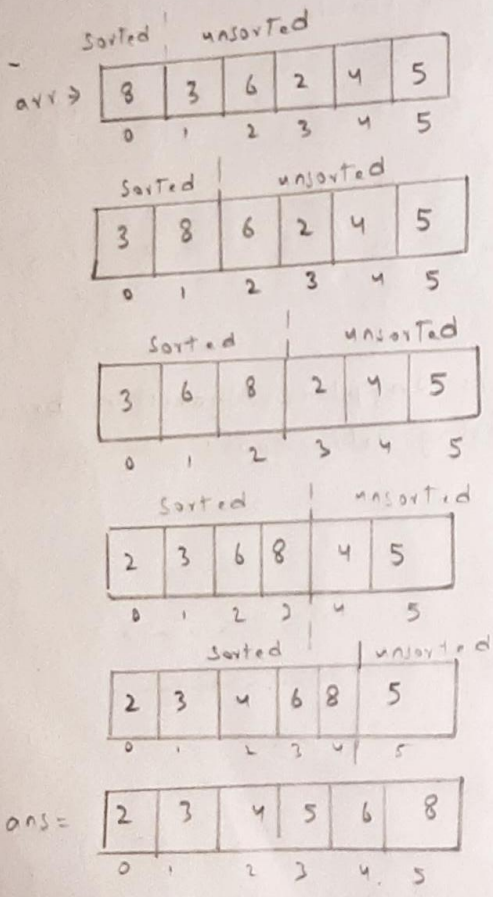
43

Insertion Sort Algorithm

* The array is virtually split into a sorted and an unsorted part.
* values from the unsorted part are picked and placed at the correct position in the sorted part

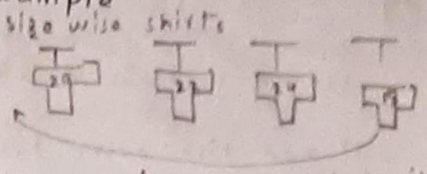
(Take one element from the unsorted part, iterate through the sorted part & find the correct position of this element)

eg -

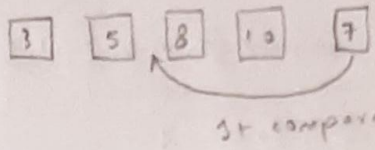


* Real life example

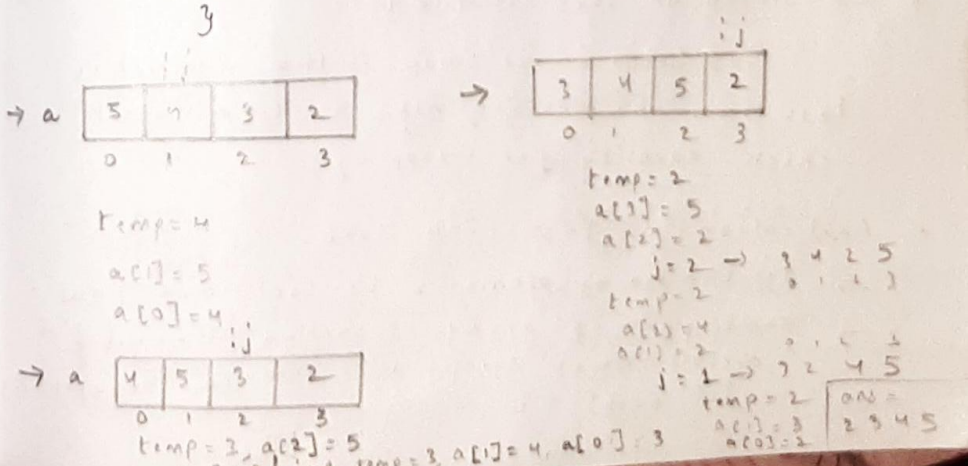
→ shirts



→ cards (while playing cards with sort the cards in our hand at correct position)




```
* static void insertionSort(int a[]) {
      int n = a.length;
      for (int i=1; i<n; i++) {
          int j = i;
          while (j > 0 && a[j] < a[j-1]) {
              int temp = a[j];
              a[j] = a[j-1];
              a[j-1] = temp;
              j--;
          }
      }
```



* Time Complexity

→ Best case - arr = {1, 2, 3, 4, 5}

we will run the outer loop n-1 times
& n. we will not run the inside loop. i.e if it is sorted we will not goin while loop. So,

T.C → O(n) (For Best case)

→ worst case - arr = {5, 4, 3, 2, 1}

i=1, 1 swap
i=2, 2 swaps
i=3, 3 swaps
:
i=n-1, n-1 swaps

$= 1+2+3 \ldots (n-1)$

$= \frac{n(n-1)}{2} \approx n^2$

T.C → $O(n^2)$ (For worst case)

& Average case

* Space Complexity → O(1) (because we are not using any extra space).

* Is Insertion sort stable?

Yes, because we swap if the number is less than the previous number. (we doesn't check equality for swapping)

* Applications of Insertion sort

→ If the no. of elements to sort are few
→ some part of array is sorted Already overall no. of swaps will reduce benefitting In O(n) T.C

Problems on Bubble, Selection, Insertion Sorting Algorithms

Problem -

Given an Integer array arr, move all 0's to the end of it while maintaining the relative order of the non zero elements

Note that you must do this in-place without making a copy of the array

Ip - 0 5 0 3 42

Op - 5 3 42 0 0

* we can solve this problem using bubble sort. because In bubble sort Algorithm after one Iteration the required maximum or minimum element will move to the last Index, Same likely we can move all zeros to the end using bubble sort.

→ Compare current Index and current+1 Index i.e if (arr[j] == 0 && arr[j+1] != 0) swap them.

arr = | 0 | 5 | 0 | 3 | 42 | (0 1 2 3 4) i = 5

i = 0 | 5 | 0 | 3 | 42 | 0 | (0 1 2 3 4) j < 5-0-1

i = 1 | 5 | 3 | 42 | 0 | 0 | (0 1 2 3 4) j < 5-1-1

i = 2 | 5 | 3 | 42 | 0 | 0 | (0 1 2 3 4)

i = 3 | 5 | 3 | 42 | 0 | 0 | (0 1 2 3 4)

use flag variable to reduce the ith Iteration i.e if they are not swapped don't Iterate ith Variable because if they not swapped means Array is already

Problem - Given an Array of names of the fruits, you are supposed to sort it in lexicographical order using selection sort.

i/p - ["papaya", "lime", "watermelon", "apple", "mango", "kiwi"]

o/p - ["apple", "kiwi", "lime", "mango", "papaya", "watermelon"]

* lexicographical order means dictionary order means a b c d . . z.

eg. If the i/p is ["college", "cot", "cottage"] all the first elements are same so we will move to 2nd element and so on and sort them correctly.

eg - lexicographical order — 1, 10, 100, 2, 20

sorted order — 1, 2, 10, 20, 100

* In strings we compare the elements using compareTo method eg. s1.compareTo(s2); returns

s1 < s2 → < 0 (neg number)

s1 > s2 → > 0 (pos number)

s1 == s2 → s1.equals(s2) (zero)

* use selection sort code to solve this problem.

** In lexicographical order sorting is done in type String (for string of numbers also).

Merge Sort Algorithm

* It uses divide and conquer Approach

Bigger problem is divided into to smaller problems, and after solving smaller problems bigger problem answer is made.

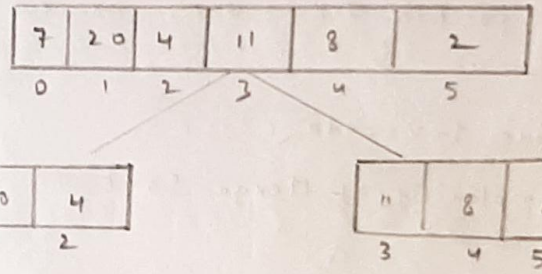
* we mostly & generally Implement divide & conquer using Recursion

Algorithm -

→ Divide the given array into to 2 equal halves

→ sort the 2 subarrays separately using Recursion } subproblem

→ Merge the 2 sorted subArrays to create an overall sorted array. Self work.

eg - arr - 

→ How to merge 2 sorted subarrays to create an overall sorted array

* we have bigger problem i.e mergeSort(arr, l, r)

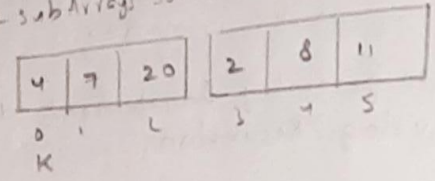
this will sort arr from l to r i.e mergeSort(arr, 0, n-1)

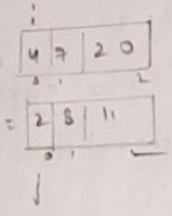
mid = l+r/2

mergesort(arr, l, r) = { mergeSort(arr, l, mid)
mergeSort(arr, mid+1, r)
merge(arr, l, mid, r)

* How to use merge function i.e merge(arr, l, mid, r)

from previous page example we got the array as (2 subArrays sorted)





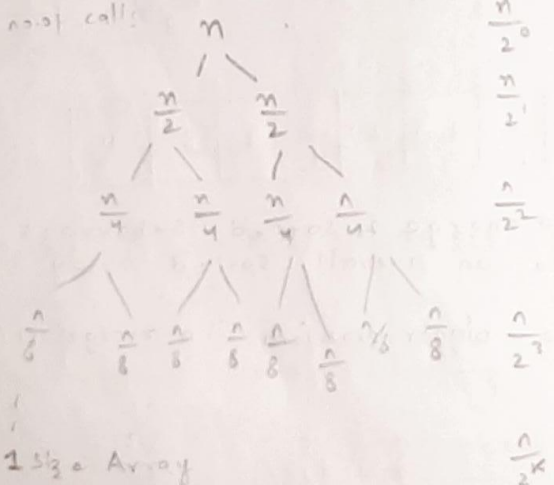
* compare i & j pointers and keep in the arr(ick) by overridding them.

* merge function Tc is O(n) because we traverse until n times (ie K pointer) (0 to 5)

* Refer code in VS code

* Time complexity of Merge Sort

Total no of calls



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

$n = 2^k$

$\log_2 n = \log_2 2^k$

$k = \log n$

Time taken → cn — time taken to merge 2 sorted Arrays i.e constant

eg $\frac{n}{2} + \frac{n}{2} = n$

$\frac{n}{4} + \frac{n}{4} = n$ so on

TC → Kcn

TC → nc log n

TC = O(n log n)

* Time complexity using substitution method

size of Array

$$T(n) \rightarrow \begin{cases} c, & n = 1 \\ T(\frac{n}{2}) + T(\frac{n}{2}) + n = 2T(\frac{n}{2}) + n, & n > 1 \end{cases} \quad - (1)$$

$T(\frac{n}{2})$ mergesort(arr, l, mid) — $T(\frac{n}{2})$ mergesort(arr, mid+1, r) — n merge(arr, l, mid, r)

substitute $n = \frac{n}{2}$ in eq (1)

$$T(\frac{n}{2}) = 2T(\frac{n}{4}) + \frac{n}{2} \quad - (2)$$

put $T(\frac{n}{2})$ from eq (2) to eq (1)

$$T(n) \rightarrow 2\left[2T(\frac{n}{4}) + \frac{n}{2}\right] + n$$

$$= 2^2 T(\frac{n}{2^2}) + 2n = 4T(\frac{n}{4}) + 2n \quad - (3)$$

put $n = n/4$ in eq (1)

$$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + \frac{n}{4} \quad -(4)$$

substitute (4) in (3)

$$T(n) = 4\left[2T\left(\frac{n}{8}\right) + \frac{n}{4}\right] + 2n$$

$$= 8T\frac{n}{8} + 3n$$

$$= 2^3 T\left(\frac{n}{2^3}\right) + 3n$$

i.e

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

we want to eliminate this T term w.k.t

$$T(1) = c$$

$$T(n) = 2^k T(1) + kn$$

$$= 2^k c + nk \qquad \frac{n}{2^k} = 1$$

$$= 2^{\log n} c + n\log n \qquad k = \log n, \quad 2^{\log n} = n$$

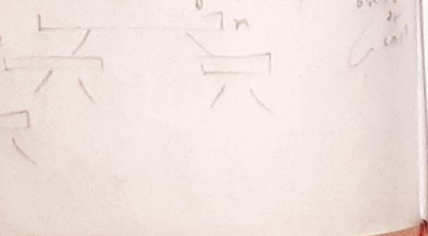
$$= cn + n\log n \cong n\log n$$

T.C → O(nlogn)

* Space complexity -

At any given point of time how many stack frames are present in stack

logn no of stack frames are present at any given point of time



$\frac{n}{2^k} < 1$ k < logn

* No. of stack frames are logn because after completing some call it is removed from the stack frame and next call is called

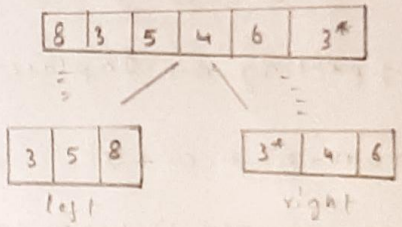
* at any point we use n space w.r.t total stack frames are logn so

$$n + \log n \cong n$$

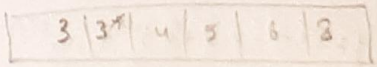
* Space complexity → O(n)

→ Is Merge sort stable?

Yes



we use left[i] ≤ right[j] so 3 comes first



→ Applications of Merge Sort

* works on large data sets
* using merge sort we can sort linked list sorting very easily

→ Drawbacks of merge sort

Best case - O(nlogn) worst, Average also same

→ In place !

NO, Space complexity is O(n)

Substitution method for calculating Time complexity

* Fibonacci Series

refer 32 lecture +imetaken

$$T(n) = \begin{cases} T(n-1) + T(n-2) + c & n \geq 2 \\ c & n \leq 1 \end{cases}$$

$T(n-1) \approx T(n-2)$

$T(n) = 2T(n-1) + c \quad -①$

→ Find $T(n-1)$ by putting $n-1$ inplace of n in eq①

$T(n-1) = 2T(n-2) + c \quad -②$

→ put the value of $T(n-1)$ from eq② in eq①

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

$\boxed{T(n) = 4T(n-2) + 3c}$

Find above 2 steps

→ $T(n-2) = 2T(n-3) + c$

$T(n) = 4[2T(n-3)+c] + 3c$

$\boxed{T(n) = 8T(n-3) + 7c}$

so on

→ $T(n) = 16T(n-4) + 15c$

$\boxed{T(n) = 2^k T(n-k) + (2^k - 1)c}$ $(k=4)$

we have to eliminate the T term

W.K.T $T(0) = c$

$T(1) = c$

$T(n-k) = T(0) = c$

$n-k = 0$

$k = n$

$T(n) = 2^n T(0) + (2^n - 1)c$

$= 2^n c + 2^n - c$

$= 2 \cdot 2^n c - c \approx 2^{n+1} c - c \approx 2^n$

$\boxed{T(n) = 2^n}$

$T \cdot C = 2^n$

* $T(n) = 2T(\frac{n}{2}) + n$ → merge sort Ans - $n \log n$

* $T(n) = 2T(n-1) + c$ → Fibonacci $- 2^n$

Follow up questions

$T(n) = T(\frac{n}{2}) + c$

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

Quick Sort Algorithm

* quick sort Time complexity In worst case is O(n²), even though it is most used but still Sortings Algorithms in practical scenarios Infact many languages library functions that use .sort (Inbuilt methods) they are used under the hood quicksort

* quick sort has an randomized version in which worst case with a good probability is avoided and we go to average case which has T.C nlogn

* Quicksort is an Inplace Algorithm (constant space)

* It is also an Divide and conquer Algorithm

* It picks an element as a pivot and partitions the given array around the picked pivot.

* we can pick the pivot element as first, last, random and median element.

* The key process in quick sort is a partition().

* The target of partition is given an array and an element x of an array as a pivot. put x at its correct position in a Array
put all smaller elements (smaller than x) before x
greater elements (greater than x) after x.

i.e

Partition 1	pivot	partition 2

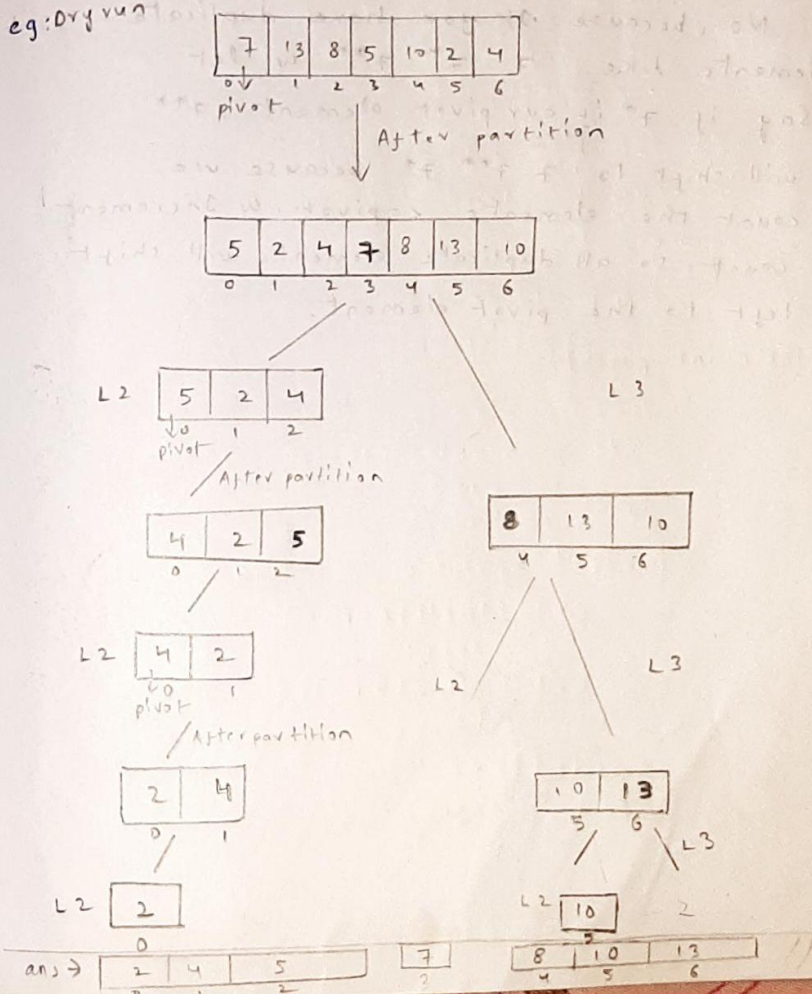
values<pivot values>pivot

* quickSort(arr, st, end) {
  if(st >= end) return;
  L1 int pi = partition(arr, st, end) → n
  L2 quickSort(arr, st, pi-1) n/2
  L3 quickSort(arr, pi+1, end) n/2
  }

pivot index

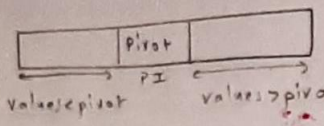
for best case i.e after partition our pivot index is at middle

eg: Dry run



* Partition Algorithm -

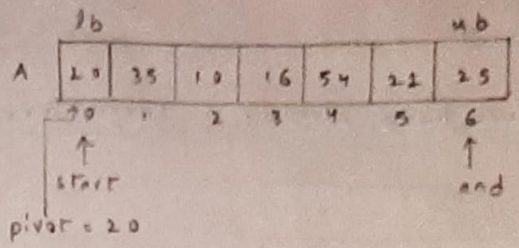
1. choose pivot & we put it at its right position that is named as pivotIndex (PI)
2. we rearrange elements before and after pivotIndex such that



* static int partition(int[] arr, int start, int end){

```
int pivot = arr[start];
int countElementsLessThanEqualToPivot = 0;
for(int i=start+1; i<=end; i++){
    if(arr[i]<=pivot){
        count++;
    }
}
int pivotIndex = start + count;
Swap(arr, start, pivotIndex);

// on left side of pivot, values should be less than pivot & on right side of pivot, values should be greater than pivot
int lb=start, ub=end;
while(lb<pivotIndex && ub>pivotIndex){
    while(arr[lb]<=pivot)
        lb++;
    while(arr[ub]>pivot)
        ub--;
    if(lb<pivotIndex && ub>pivotIndex){
        Swap(arr, lb, ub);
        lb++;
        ub--;
    }
}
return pivotIndex;
}
```



* IS quick sort stable ? No

because, let say you have duplicate elements i.e 7 7^* 7^{**} and let say 7^* is pivot element, the values on left side should be less than and equal to pivot element so 7^{**} will be shift left i.e 7 7^{**} 7^* so quick sort is not stable

* Quick sort Time Complexity

→ Average case / Best case → $o(n \log n)$

→ worst case → $o(n^2)$
↳ can mostly be avoided by using randomized quicksort

* partition function time complexity is
$c + n + c + cn \Rightarrow 2n \cong n$

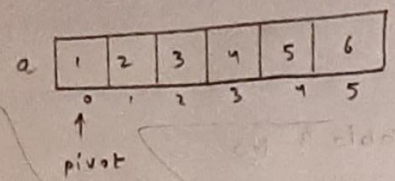
* Best case - (completely balanced partition)

$T(n) = 2T\left(\frac{n}{2}\right) + n$ $\quad T(1) = c$

$T(n) \rightarrow o(n \log n)$ (refer previous lecture)

* Worst case - (completely unbalanced partition)

let say we have already a sorted array



i.e

$T(n) = T(0) + T(n-1) + n$ → partition function

($T(0) = c$)

$\boxed{T(n) = T(n-1) + n}$ - (1) $\boxed{T(1) = c}$

Substitute $n = n-1$ in above eq

→ $T(n-1) = T(n-2) + n-1$

Substitute $T(n-1)$ in eq (1)

$\boxed{T(n) = (T(n-2) + n-1) + n}$ - (2)

Substitute $n = n-2$ in eq (1)

→ $T(n-2) = T(n-3) + n-2$

Substitute in eq (2)

$\boxed{T(n) = (T(n-3) + (n-2) + n-1) + n}$ - (3)

→ Similarly $n = n-3$ in eq (1)

$T(n-3) = T(n-4) + n-3$

Substitute in eq (3)

$T(n) = T(n-4) + (n-3) + (n-2) + (n-1) + n$

$\boxed{T(n) = T(n-4) + 4n - (1+2+3)}$

→ For K^{th} step

$\boxed{T(n) = T(n-K) + Kn - (1+2+..K-1)}$

$T(n) = T(n-K) + Kn - (1+2+3..K-1)$

$T(n) = T(n-K) + Kn - \frac{K(K-1)}{2}$

$\frac{n(n+1)}{2}$ $\frac{K(K-2+1)}{2}$ $\frac{K(K-1)}{2}$

we have to eliminate the T term by K

w.k.t $T(1) = c$

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

$\boxed{n-K = 1 \quad K = n-1 \simeq n}$

$T(n) = T(1) + n(n) - \frac{n(n-1)}{2}$

$= c + n^2 - \frac{n^2}{2} \simeq n^2 \simeq n^2$

$\boxed{T(n) = O(n^2)}$

* Average case -

quicksort (arr, st, pi-1) → size $T(pi-1)$

quicksort (arr, pi+1, end) → size $T(n-pi)$

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

pi can be anything starting from 0 to n-1

i.e

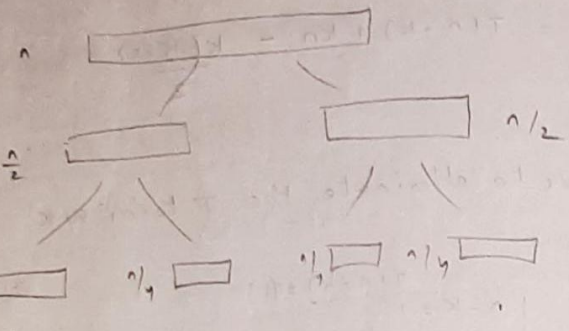
$T(n) = \frac{1}{n} \sum_{pi=0}^{pi=n-1} [T(pi-1) + T(n-pi)] + n$

Avg of all possible partitions

After solving

T.C → $O(n \log n)$

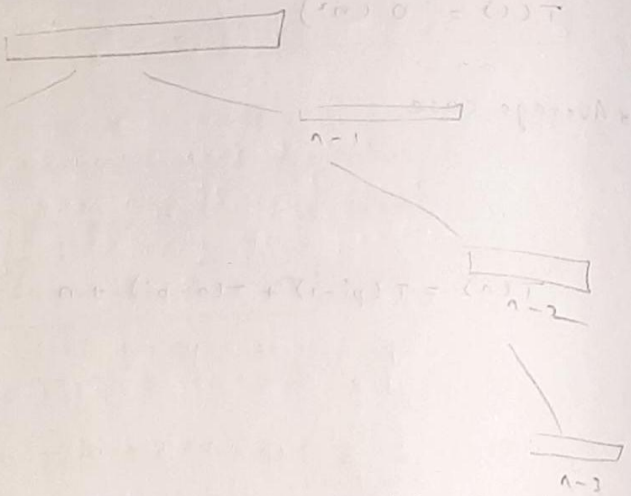
Best case - (Recursive tree)



$\frac{n}{2^k} = 2$ (no. of levels) $k = \log n$ (time in each level)

$k = (n \log n)$

worst case -



Total levels $k = n$

Time taken in one level = n

Total level * Time taken

T.C → $O(n^2)$

* we are getting the worst case if we are taking the pivot element at first index, let say our pivot element is at middle index (In sorted Array also) (T(n-1) component will not occur only if we choose our pivot at middle.

* So, while Implementing quicksort we should not take our pivot element as first index (everytime) we should take any random index as pivot element and that random index no. should be between start & end. and our pivot element should/will be arr[randomIndex] (partition function)

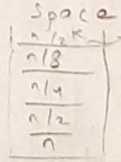
H.W Find random number b.w.t start and end (we have library function also for finding random no)

* After randomized approach we will reach the Averagecase/ Bestcase

* eventhough worst case T.C is $O(n^2)$ but always it can be avoided, (worst case) probability is very less thats why it is very good algorithm.

Space complexity

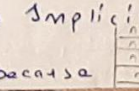
* We have not used any extra space (In code). but if we consider Implicit stack (Best/Avg case)
* our stack order will be (Quicksort recursive tree)
the maximum no. of stack frames at any given point of time is $\log n$



$n/2^k = 1$

$k = \log n$

* So Space complexity due to Implicit stack is $\log n$. (Best/Avg case)
* In worst case it will be linear because



n = k (no. of stack frames) so, $O(n)$

* It is an Inplace algorithm because S.C → logn, for all values of n, logn is Very Small No. So it is called as Inplace Algorithm because It is kind of using very small Space. & If we Ignore the Implicit stack (the recursion), then It is pure Inplace algorithm

* what is the need for new ways of partitioning & randomized quicksort Algorithm.

because we want to avoid the worst case so everytime we will not take pivot element as start Index so we use new ways of partitioning. (so to avoid worst case we use randomized quicksort Algorithm)

* Applications

→ where memory is concerned we use quicksort (because It is Inplace Algorithm) Infact java library sort function is also quicksort

* Mergesort vs/ quicksort

Mergesort	quicksort
→ stable	unstable
→ linear space	Inplace
→ best, Avg, worst are nlogn	worst case $O(n^2)$, best, avg → nlogn
→ not preferred than quick sort (but while Implementing linkedlist It is used more) because finding randomized no. is very difficult (not always possible to find random no. for partition) as pivot	preferred more due to less consumption of memory

Count Sort Bucket Sort Radix sort Algorithms

* Count Sort

→ It is non comparison based algorithm. (previous Algorithms are comparison based Algo's)

→ count sort is used when range of numbers are defined i.e range (1-100) are In Array (k) or If In question they have not given the range (traverse the Array and find maximum no. and make it as k i.e range of number)

→ we use extra memory (Not In place Algorithm)

→ under certain conditions/situations count sort can work better than previous sorting Algorithms which has T.C O(nlogn), which is best under worst conditions In previous Algo's.

* Implementation (Basic count Sort)

arr =

0	1	2	3	4	5	6	7
4	3	2	5	3	1	3	5

Find max in above array i.e 5, make length of size 0 to 5

frequency/count =

0	1	2	3	4	5
0	2	0	3	1	2

1, 2 times (same for rest)

arr =

0	1	2	3	4	5	6	7
2	2	3	3	3	4	5	5

* refer code in VS code
* In previous implementation/ Algorithm we have sacrificed the stability property entirely
* and using previous approach we cant sort objects also.

* why stability is important

let say we have flights destination & departure time

BLR , 2		BLR 2		BLR, 2
Mumbai, 5	Sorting Timewise	Mumbai 5	sort Alphabetically	BLR , 7
BLR , 13	→	BLR , 7	→	BLR , 13
Mumbai, 10		Mumbai, 10		Mumbai, 5
BLR , 7		BLR , 13		Mumbai , 10

If we do not have stability, while sorting alphabetically BLR 7 may come first or after BLR 13, the original order will basically not be maintained

* How we can make our count sort stable.

Implementation 2 - (Count Sort)

arr -

4	3	1	5	3^x	1^x	3^y	5^y
0	1	2	3	4	5	6	7

first 2 steps same as previous implementation

count =

0	2	0	3	1	2
0	1	2	3	4	5

change count to prefix sum Array i.e count[i] = count[i-1] (loop starts from 1)

count =

0	2	2	5	6	8

Traverse arr from back i.e n-1 find position at n-1 using count array & decrement the value of count array

o/p

1	1^x	3	3^y	3^x	4	5	5^x

* using prefix sum we can find the index of of each element of the original array in prefix sum Array and put it in the o/p array

i.e eg - arr value = 3^x

prefixsum[3] = 5 (i.e at position 5 the last value of 3 is present)

* refer code in VS code

Time complexity - (From code)

i e

worst case → $nc + max.c + n.c + n.c \approx (n + max)$

k

maximum no

Average case → $(n+k)$

Best case → (n) for range i.e k is 1

* we will think that it is linear it is best algorithm among all but eg

n =

1	2	10^5	4

$k = 10^5$

T.c depends on max No

i.e when range is defined and small the Best case works good.

Space Complexity - (From code)

S.C → $n+k$

output count arr arr

* Radix Sort

→ It is also non comparison based algorithm just like count sort.

→ we sort the array digit by digit and solve the full array, It is also called place value / position of digit in a number. (hundreds tens ones place value)

* starting from least significant bit to most significant bit we sort the array digit by digit.

eg-

a

170	45	75	90	802	2
0	1	2	3	4	5

n = 6 (because highest digit no. is 3 i.e 802_3)

→ Make all elements of size same : o 3^

170	045	075	090	802	002

(one's place sorting →)

170
090
802
002
045
075

(tens →)

802
002
045
170
075
090

(Hundreds →)

Sorted Array

002
045
075
090
170
802

* Make sure stability don't change (i.e order of no's)

* we sort digit by digit sorting using counting sort

* one's place counting sort

a

170	045	075	090	802	002
0	1	2	3	4	5

n = 6

S1- count[] → In count sort we take size of maximum frequency of no In Array, but in radix sort we make digit by digit sorting so maximum value is 9 i.e size is 10 digit

2	0	2	0	0	2	0	0	0	0
0	1	2	3	4	5	6	7	8	9

S2- prefixsum of count

2	2	4	4	4	6	6	6	6	6
0	1	2	3	4	5	6	7	8	9

S3- o/p Arr

170	090	802	002	045	075
0	1	2	3	4	5

* Tens place sorting.

arr =

170	090	802	002	045	075
0	1	2	3	4	5

count =

2	0	0	0	1	0	0	2	0	1
0	1	2	3	4	5	6	7	8	9

prefixsum count =

2	2	2	2	3	3	3	5	5	6
0	1	2	3	4	5	6	7	8	9

o/p Arr =

802	002	045	170	075	090
0	1	2	3	4	5

* same for Hundredi's place

Digit at place?

* x = 72456 (we have to find In the given number x what is the digit In given place?)

place = 10

ans- $\frac{72456}{10} = (7245) \% 10 = 5$

Division removes last digit
remainder gives last digit

* refer code In Vscode

* Time complexity (refer code)

$n + d(n+10) \cong dn$ (best, worst, Average)
$\cong dn$

* Space complexity

$(n+10) \cong n$

* It is not Inplace Algorithm (because countsort is used which consumes extra space).

* Bucket Sort

→ Bucket Sort is used generally when you are given a range of numbers and that no's are uniformly distributed

- what is uniformly distributed and why bucket sort is good Algo for uniformly distributed no's ?

Basic fundamental of Bucket Sort

1* put all elements in b no. of buckets

2* Sort each bucket Individually

3* Take out all elements & join/merge them.

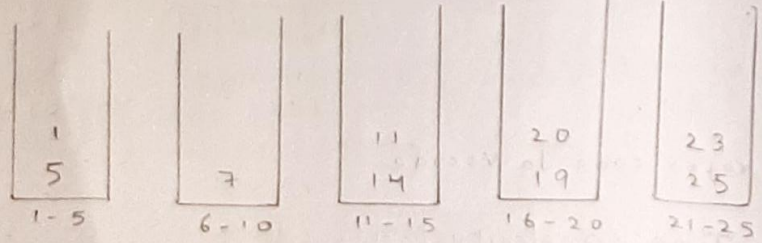
eg-

9	19	25	14	5	20	22	23	1
0	1	2	3	4	5	6	7	8

n = 9

Step1 (1)

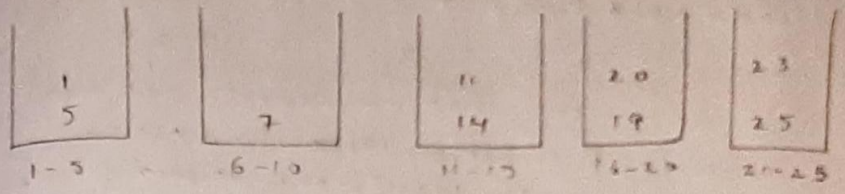
* The above arr elements are ranging from 1 to 25 because min no is 1 and max is 25.
* make certain no. of buckets you can choose that how many buckets you have to make. let say for now we have to make 5 buckets.



→ we have range of no's 1-25 & we have 5 buckets, So we can say that In each bucket take no's i.e 1-5, 6-10, 11-15, 16-20, 21-25. (pattern) on what basis we put In numbers in buckets .ie by range of numbers (this can be done using other pattern also)

Step2 (S2)

sort each bucket Individually



sort bucket 1 - 1, 5

2 - 7

3 - 11, 14

4 - 19, 20

5 - 23, 25

Step3 (S3)

Take all elements from bucket & join / merge them

1	5	7	11	14	19	20	23	25

* the use of this bucket sort is let say if we have done direct sorting then we will do for all n elements, but we put no. of elements In buckets, and no. of elements for which we have to sort Individually will decrease
* uniform distribution means that all elements In all buckets are divided uniformly. for eg- let say In each bucket we have 1 element. for 1 element to sort the time is very less, & if In one bucket we have all elements to sort this takes some T.c it will consume depending upon algorithm (if insertion $O(n^2)$), but if we put elements In buckets we for 1 element to sort the time is very less and finally we merge them. So we have best case is uniform distribution which has T.c constant

* while at the same time let say uniform distribution may occur all elements are at one place i.e

11 12 11 13 11 11 12 13 25

so we can say that it elements are not uniformly distributed there is no use of bucket sort. If elements are uniformly distributed we can use bucket sort

Question - sort an Array having numbers in range [0.0, 1.0] with uniform distribution find efficient Algo to sort

a =

0.42	0.32	0.25	0.52	0.23	0.47	0.51	0.92	0.68	0.95
0	1	2	3	4	5	6	7	8	9

n = 10

Buckets - 10 (let say)

	put elements using certain pattern	sort bucket	merge them
0			0.23 0.25 0.32 0.42
1			0.47 0.51 0.52 0.68
2	0.25, 0.23	0.23, 0.25	0.73 0.92
3	0.32	0.32	
4	0.42, 0.47	0.42, 0.47	
5	0.52, 0.51	0.51, 0.52	
6	0.68	0.68	
7	0.95	0.78	
8			
9	0.92	0.92	

→ finding a way to put elements in to the bucket? let say 0.42 x 10 = 4.2 ceil is 4, so put 0.42 in bucket 4.

let say we have numbers

25 27 39 41

put elements based on first digit or last digit

i.e arr[i]*/10

25*/10 = 9 in 9th bucket etc.

* refer code in Vs code
* Time complexity

Best case - Individual elements (uniform distribution)

$O(n+k)$

↓

buckets

Worst case - (non uniform distribution) $O(n^2)$

48

Problems on Advanced Sorting Algorithms

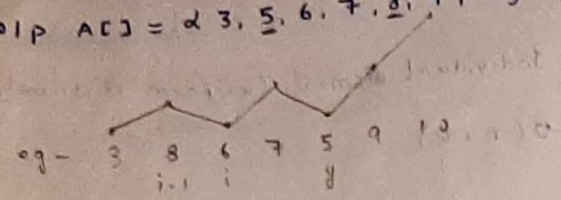
Algorithm	Worst case	Average case	Best case	Space complexity	Stable
Bubble	n^2	n^2	n	O(1)	Yes
selection	n^2	n^2	n^2	O(1)	Yes
Insertion	n^2	n^2	n	O(1)	Yes
Merge	nlogn	nlogn	nlogn	o(n)	Yes
Quick	n^2	nlogn	nlogn	O(nlogn)	No

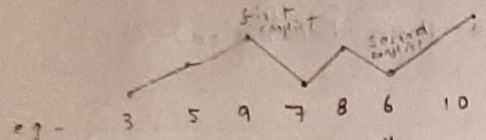
Problem -

Given an Array where all its elements are sorted in Increasing order except two swapped elements, sort it in linear time. Assume there are no duplicate elements in the Array.

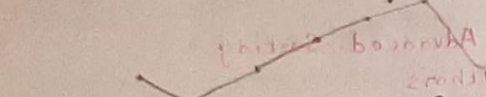
input A[] = {3, 8, 6, 7, 5, 9, 10}

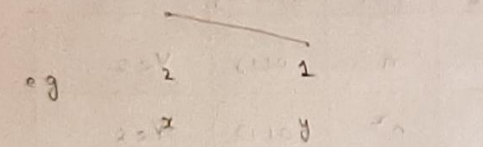
o/p A[] = {3, 5, 6, 7, 8, 9, 10}





How to know if it is first conflict or second conflict.
Initialize x with -1 so that if arr[i-1] > arr[i] x will change to i-1 and y remains i only for second conflict x is not equal to -1 so y value will be i.





* refer code in vscode
* T.C → o(n)

problem - given an array of +ve & -ve integers, segregate them in linear time and constant space. The o/p should print all negative numbers, followed by all positive numbers.

Input: {19, -20, 7, -4, -13, 11, -5, 3}

output: {-20, -4, -13, -5, 7 11 19 3}

* use partition method to solve this question (Quick sort)
* refer code in vscode
* T.C → o(n)

problem - Given an Array of size N containing only 0s, 1s, 2s; Sort the Array in Ascending order. (Dutch National flag problem)

i/p - N = 6

arr[] = {0 2 1 2 0 0}

o/p - 0 0 0 1 2 2

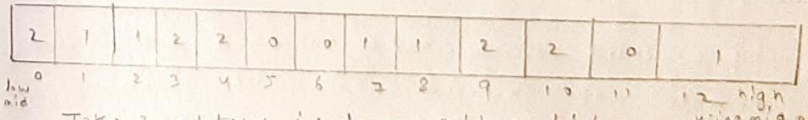
Approach -

① * Sort the Array using any of the sorting Algo & the sort the Array. if we use merge or quick sort T.C will be o(nlogn). If we use merge sort T.C will be o(n) & quick sort S.C will be o(logn).

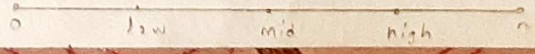
② * we can also use count sort if the range of numbers are already given in the question the range is already given. the question can be solved in linear time o(n) & constant space o(1)

this approach has an problem i.e we are use 2 passes for Array traversal i.e 1. to find the count & 2. to fill the Array so T.C will be n+n = 2n ≅ n but in interview they can ask solve this question in 1 pass. so we have to think a approach we takes linear time, 1 pass & constant space

③ * using 3rd approach we can solve this question in one pass eg-



Take 3 pointers i.e low=0, mid=0, high=n-1 & using mid pointer traverse the Array. Define 4 regions i.e



* [0, low-1] => 0
  [low, mid-1] => 1
  [mid, high] => unknown
  [high+1, n-1] => 2
  } -①

* use mid pointer to traverse the Array. we can have 3 conditions i.e
  arr[mid] == 0, arr[mid] == 1, arr[mid] == 2

→ c1 If arr[mid] == 0
  swap(mid, low)
  low++;
  mid++;

→ c2 If arr[mid] == 1
  mid++;

→ c3 If arr[mid] == 2
  swap(mid, high)
  high--;

→ [why ? In first case low++ & mid++
  why ? In second case only mid++
  why ? In third case only high--] reverse dryrun

* In every point (index) try to see that the ea ① regions satisfies or not

* So, the points which are in unknown region mark as X, (representing they are in unknown region).

Dry run — (Do dryrun as per regions)

* from example, we have arr[mid] == 2 so swap(mid, high) and decrement high. decrement only high because in old high index, the crct value is swapped already i.e 2 so dont consider the old high as unknown region & mark it as ✓.

* why in third case we have not done mid++ because we have not observed what is the value of arr[high] we are seeing only arr[mid] value So we are swapping if arr[mid] = 2 and decrementing high-- & we are not incrementing mid++ because arr[high] is unknown it can be either 1 so we are not incrementing. because if it is one it should be incremented, at starting index 0 values should be there but there will be 1 value. so we are not incrementing

2. now arr[mid] = 1 and we are only incrementing mid++ (we are not swapping with any index) because we want from range (low, mid-1) the value should be 1 if we increment mid for arr[mid] = 1, mid-1 will be one so we are incrementing mid. now previous arr[mid] = 1 will also be in correct region ✓.

3. If arr[mid] == 0 swap(mid, low) & low++ & mid++. mid++ because we know if we swap with low the value is already one so we are incrementing. i.e from range [low, mid-1) we know the values are ones. low++ because we want from range (0, low-1] the values should be zero.

* follow the entire process until (mid <= high) until the range [mid, high] they are unknown elements.

* refer code in vs code (Doubt we used if, else if, else why not if if if seperately)

* T.C → O(n)
  S.C → O(1)