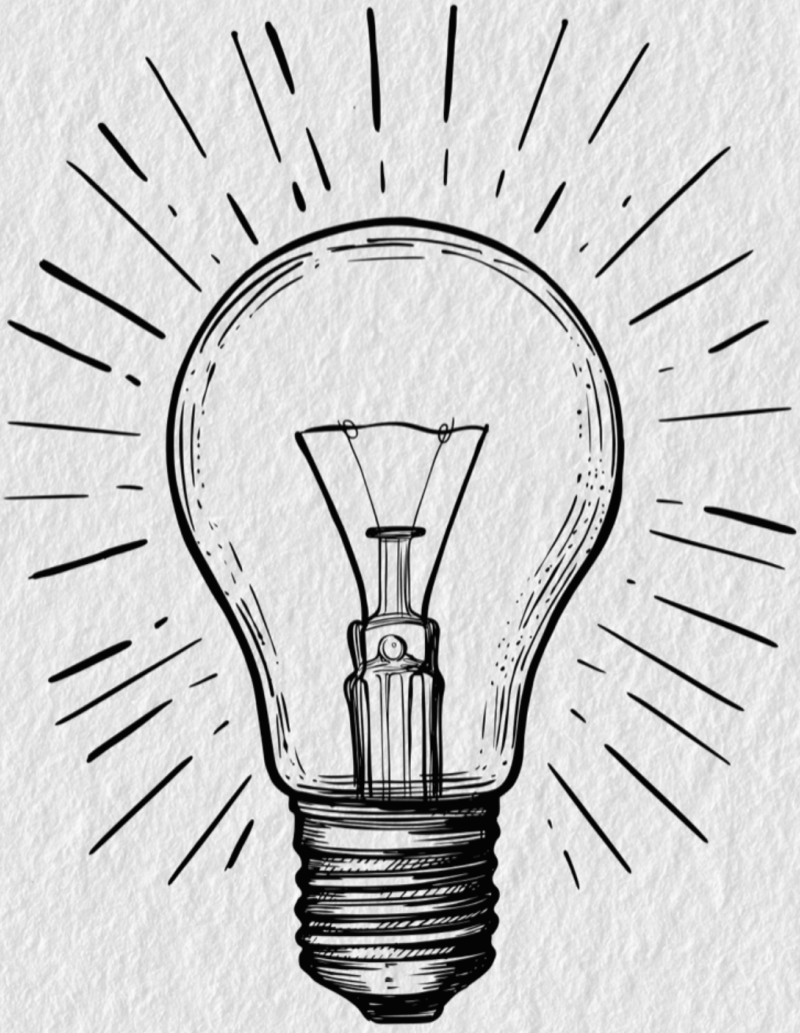
NOTEBOOK



NOTE IN

Creative notes

By reading we enrich the mind;
by writing we polish it.

Shiva

1/2/25

Bubble sort → Stable

Intitution push the maximum element to the last by adjacent swapping.

```java
public static void bubbleSort(int a[], int n) {
    // Iterate over the entire array
    for (int i = 0; i < n; i++) {
        boolean swap = true;  // Initially assume no swaps have occurred
        // Compare adjacent elements up to the unsorted portion of the array
        for (int j = 0; j < n - i - 1; j++) {
            if (a[j] > a[j + 1]) {
                // Swap a[j] and a[j+1]
                int temp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = temp;
                swap = false;  // A swap occurred, so mark the flag accordingly
            }
        }
        // If no swaps occurred during the inner loop, the array is sorted
        if (swap) {
            break;
        }
    }
}
```

Here $i \Rightarrow 0$ to $n-1$ is enough

→ swap technique

→ Best case

Time complexity Derivation => For Example $n = 5$

$i=0 \rightarrow j=0$ to $n-1$ (4)

$i=1 \rightarrow j=0$ to $n-2$ (3)

$i=2 \rightarrow j=0$ to $n-3$ (2)

$i=3 \rightarrow j=0$ to $n-4$ (1)

$i=4 \rightarrow j=0$ to $n-5$ (0) X (Not running j loop)

From Above, for $n=5$

$$4+3+2+1 = 10$$

$$\frac{5 \times 4}{2} = 10$$

So, $\Rightarrow \frac{n(n-1)}{2} \Rightarrow \frac{n^2-n}{2} \Rightarrow \frac{n^2}{2} - \frac{n}{2} \Rightarrow O(N^2)$

⊗ Time complexity is $O(N^2)$

Time complexity ⇒ $O(N^2)$ worst case, Average case

$O(N)$ Best case.

Space complexity ⇒ $O(1)$

$n=6$ [13, 46, 24, 52, 20, 9]

24 46 20 9 52

$i=0 \rightarrow$ [13, 24, 46, 20, 9, 52]

pushing the maximum to last

Note :-

$j=0$ to $n-i-1$?

→ Every large element pushed to end for every i^{th} iteration.

⊗ sorting from right side to left side.

⇒ Best case :

[2, 3, 4, 5, 6, 7]

Swap = true;

if any swap occurs,

Swap = false;

⊗ Selection sort → unstable

⇒ Select minimum number and push to first end.

```java
public static void insertionSort(int a[], int n){   // SelectionSort
    for(int i = 0; i<n; i++){                       // → n-1 is enough.
        int minIdx = i;                             // → Intial min Idx
        for(int j = i+1; j<n; j++){
            if(a[minIdx]>a[j]){                     // → if any min value found
                minIdx = j;                         //   update minIdx.
            }
        }
        int temp = a[minIdx];                       // } swap that min value with
        a[minIdx] = a[i];                           //   current ith value.
        a[i] = temp;
    }
}
```

Time complexity Derivation: ⇒ For example N=5

- i=0 ⇒ j = 1 to n (4)
- i=1 ⇒ j = 2 to n (3)
- i=2 ⇒ j = 3 to n (2)
- i=3 ⇒ j = 4 to n (1)
- i=4 ⇒ j = 5 to n (x) Not running jth loop

So i = 0 to n-1 is enough

⇒ for n=5, 4+3+2+1 = 10

$$\frac{5(5-1)}{2} = 10$$

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

Time complexity ⇒ $O(N^2)$

Space complexity ⇒ $O(1)$

Time complexity ⇒ $O(N^2)$

Space complexity ⇒ $O(1)$

Time complexity ⇒ $O(N^2)$ } Best case, Average, worst

Space complexity ⇒ $O(1)$

i=0

[13, 46, 24, 52, 20, 9]
0 1 2 3 4 5

minIdx = i; arr[minIdx] > arr[j] then minIdx = j;

At ⇒ i=0

minIdx = 0 5

swap ⇒
```
temp = arr[minIdx];
arr[minIdx] = arr[i];
arr[i] = temp;
```

→ why ⇒ j = i to n ?

→ pushing min element to first for every ith iteration.

⊗ sorting from left side to right.

Insertion Sort

⇒ Taking an element, place it in it's correct position.

```java
public static void insertionSort(int a[], int n){
    for(int i = 1; i<n; i++){      → 1 to n-1
        int j = i;                 → pick 1 element
        while(j>0 && a[j]<a[j-1]){
            int temp = a[j];       } swap
            a[j] = a[j-1];         → put that in it's correct position.
            a[j-1] = temp;
            j--;  → decrement
        }
    }
}
```

Run | Debug

Time complexity ⇒ $O(N^2)$ Average, Worst

⇒ Best case ⇒ $O(N)$

Space complexity ⇒ $O(1)$

[14, 9, 15, 12, 6, 8, 13] (0 1 2 3 4 5 6)

i=1, j=1 → 0 : 9 14

[9, 14, 15, 12, 6, 8, 13]

i=2, j=2 → 1 → 0 : 12 14 15

[9, 12, 14, 15, 6, 8, 13]

i=3 → j=3 ⇒ Here (j < j-1) → false ; j → 15 is in it's position

[9, 12, 14, 15, 6, 8, 13]

i=4 → j=4 → 3 → 2 → 1 → 0 : 6 9 12 14 15

[6, 9, 12, 14, 15, 8, 13]

i=5 → j=5 → 4 → 3 → 2 → 1 : 8 9 12 14 15

[6, 8, 9, 12, 14, 15, 13] ← final sorted array.

i=6 < n-1 ✗ loop terminates.

Time complexity Derivation ⇒ For ex: n=5

i=1 ⇒ j=1 to 1 (1)

i=2 ⇒ j=2 to 1 (2)

i=3 ⇒ j=3 to 1 (3)

i=4 ⇒ j=4 to 1 (4)

By analysing above, for 5 ⇒ 4+3+2+1 ⇒ 10

$$\frac{5 \times 4}{2} = 10$$

$$\frac{n(n-1)}{2} \Rightarrow \frac{n^2}{2} - \frac{n}{2} \Rightarrow O(N^2)$$

Time complexity is $O(N^2)$

Space complexity is $O(1)$

Best case ⇒ $O(N)$

Worst, Average ⇒ $O(N^2)$

⊛ merge sort Algorithm ⇒ Divide and merge

4/2/25

```java
// Recursive method to perform merge sort on a subarray from index 'start' to 'end'
public static int[] mergeSort(int arr[], int start, int end) {
    // Base case: if the subarray has one element or is empty, return it as a single-element array
    if (start >= end) {
        return new int[]{arr[start]};
    }

    // Calculate the middle index to divide the array into two halves
    int mid = (start + end) / 2;

    // Recursively sort the left half (from start to mid)
    int left[] = mergeSort(arr, start, mid);
    // Recursively sort the right half (from mid+1 to end)
    int right[] = mergeSort(arr, mid + 1, end);

    // Merge the two sorted halves and return the result
    return merge(left, right);
}
```

→ Divide the array recursively

→ merge the array before returning

⊛ Time complexity derivation

→ why N*logN.

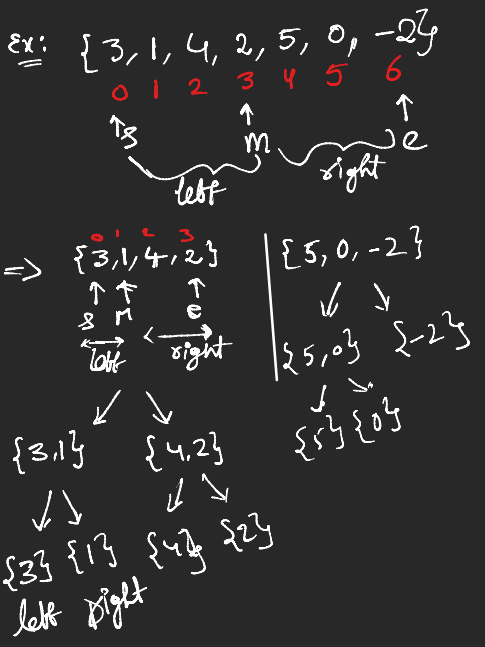
① For dividing log N

For every subpart of array, we call merge().

Merge() fun takes O(N)

⇒ Overall time complexity is O(N*log N)

```java
// Helper method to merge two sorted arrays into one sorted array
private static int[] merge(int left[], int right[]) {
    // Create a new array to hold the merged result
    int ans[] = new int[left.length + right.length];
    int i = 0; // Index for the left array
    int j = 0; // Index for the right array
    int k = 0; // Index for the merged array

    // Merge elements from left and right arrays into ans in sorted order
    while (i < left.length && j < right.length) {
        // Compare current elements of left and right
        // Add the smaller element to the merged array
        if (left[i] < right[j]) {
            ans[k] = left[i];
            i++;
        } else {
            ans[k] = right[j];
            j++;
        }
        k++;
    }

    // Add any remaining elements from the left array (if any)
    while (i < left.length) {
        ans[k] = left[i];
        i++;
        k++;
    }

    // Add any remaining elements from the right array (if any)
    while (j < right.length) {
        ans[k] = right[j];
        j++;
        k++;
    }

    // Return the merged sorted array
    return ans;
}
```



⇒ Base condition (single element)

(start >= end)

↓

return new int[] {arr[start]}

left[] = divide( );
right[] = divide( );
return merge (left, right);

Time complexity : O(N * log N)

Space complexity ; O(log N) ⇒ stack space

O(N) ⇒ for merging

⊛ Quick Sort (pivot & partition)

4/2/25

Approach

→① Take pivot (Any element) (Here I took mid)

→② partition (place pivot element in it's position)

⊛ Partition logic

- → as [start] < pivot, then move start++
- → as [end] > pivot, then move end--
- → swap (start, end)
- → Do, Above three until (start < end)

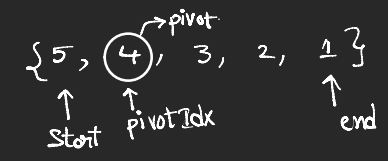
```java
public static void quickSort(int a[],int low, int high){
    if(low < high){
        int pivotIdx = partition(a,low,high);
        quickSort(a, low,pivotIdx-1); // left part after placing pivot at it's correct
        quickSort(a, pivotIdx+1, high); // right part after placing pivot at it's corre
    }
}

private static int partition(int a[],int low, int high){
    int start_Idx = low;
    int end_Idx = high;
    int mid = (start_Idx + end_Idx)/2; // Here you can take any element as pivot. In th
    int pivotIdx = mid; // Mid is the pivot index.
    int pivot = a[mid];
    while(start_Idx < end_Idx){
        while ( start_Idx< high && a[start_Idx]<=pivot) { // see these cases carefully,
            start_Idx++;
        }
        while (end_Idx > low && a[end_Idx]>pivot){
            end_Idx--;
        }
        if(start_Idx < end_Idx){ // swap
            int temp = a[start_Idx];
            a[start_Idx] = a[end_Idx];
            a[end_Idx] = temp;
        }
    }
    // place pivot at it's position. so swap a[pivotIdx] with a[end_idx]
    int temp = a[end_Idx];
    a[end_Idx] = a[pivotIdx]; // which is pivot; pivot ;
    a[pivotIdx] = temp;

    return end_Idx; // return pivot postion.
}
```

→ this is the reason, why we return endIdx

Time complexity ⇒ O(N * log N)

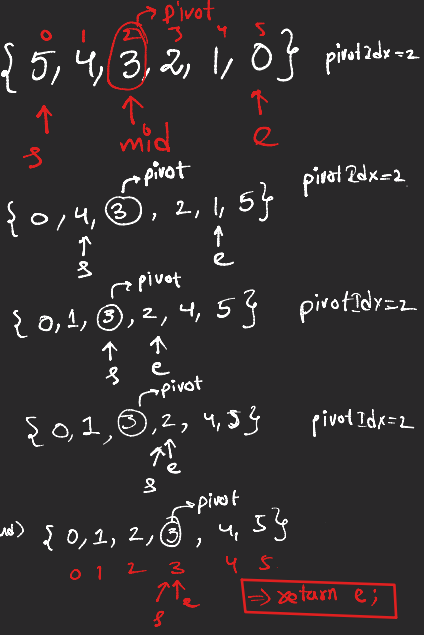
Space complexity ⇒ O(1)

⊛ How to put pivot At correct pos?



Aim: put larger elements to right of pivot
put smaller element to left of pivot

Ex: {5, 4, 3, 2, 1, 0} pivotIdx=2



Recursively for

left {0, 1, 2} and right {4, 5}

⊛ Time complexity ⇒ Worst case ⇒ O(N²)
Average case ⇒ O(N*log N)

⊛ space complexity ⇒ O(1) (constant space)

Why should we return end ? (Important)

- → Reason is we check a[startIdx] <= pivot
- → So, All lesser than equal elements go left.
- → If we see, a[endIdx] >= pivot,
  then swap (a[startIdx], pivot)