Day 16
24/02/2024

Sorting

* Sorting is the process of arranging items in a certain sequence or order.

* This arrangement can be done based on various criteria, depending on the context, such as numerical order, alphabetical order, or even based on the size OR length of items.

* The main goal of sorting is to organize data in a way that it makes easier to understand, analyze or find specific items.

in computing and data processing, sorting plays a crucial role in managing and organizing data efficiently.

Different algorithms have been developed to perform sorting each with its strengths and weaknesses, depending on the nature and size of the dataset.

These are some common sorting algorithms.

1. Selection Sort
2. Bubble Sort
3. insertion Sort
4. merge sort
5. Quick Sort

1. Selection Sort:

* Selection sort is a straightforward sorting Technique That organizes items by repeatedly finding the smallest element from an unsorted section and moving it to the begining of that section

How it works:

→ Start the whole array as unsorted
→ find the smallest element in the unsorted section of the array
→ Swap this smallest element with the element at the begining of the unsorted section

→ consider the begining of the array as sorted and the rest as unsorted. move the boundary between sorted and unsorted sections one element to the right

→ Repeat steps 2-4 until whole array is sorted.

***

it doesen't require extra memory for another array, making it memory-efficient." However, it's slower with large datasets because it's goes through entire array ~~making it~~ to find the smallest element for each position in the sorted section".

<u>Example</u>

$$\text{arr} = \{\overset{0}{29}, \overset{1}{\underline{10}}, \overset{2}{14}, \overset{3}{37}, \overset{4}{13}\}$$

Step 1: finding minimum element and put it in 0 index

$$\{10, \underbrace{29, 14, 37, 13}\}$$

Step 2: Begining is sorted and repeat it for remaining elements

$$\{10, 14, 29, 37, 13\}$$

step3: $\{10, 14, 13, 29, 37\}$ now array is sorted

~~step4: {10, 14, 13, 29, 37}~~

code

```
int n = arr.length;
for (int i = 0; i<n-1, i++) {
    int min_idx = i;
        for(int J = i+1; J<n; J++)
            if (arr[J] < arr[min_idx])
                min idx = J;
~~//Swap the found element~~
// Swap the found minimum element with the first
int temp = arr[min_idx]
arr[min_idx] = arr[i]
arr[i] = Temp
```

How it works: first outer loop iteration ($i=0$)

initial state: [29, 10, 14, 37, 13] → entire array is unsorted at this point, min idx = 0;

inner loop iterations: iteration 1

for (int J = i+1; J<n; J++)

J = 0+1 J<n J++
J = 1 1<5

Compare arr[J] = 10 with arr[min_idx] = 29
arr[1] = 10 arr[0] = 29

if arr[J] < arr[min_idx]
10 < 29
Then min idx = J
min_idx = 10

10 is smaller, so min_idx updated to 10

inner loop iteration 2 [29, 10, 14, 37, 13]

int J = i+2 J<n J++

J = 0+2 2<4

J = 2 2<5 2++

compare arr[J] = 14 with arr[min-idx] = 10

arr[2] = 14 arr[1] = 10

14 < 10 false

so no change in min idx

inner loop iteration 3

int J = i+3 J<n J++

J = 3 3<5 3++

compare arr[J] = 37 with arr[min-idx] = 10

arr[3] = 37 arr[1] = 10

37 < 10 false

so no change in min idx

inner loop iteration 4

int J = i+4 J<n J++

J = 4 4<5 4++

compare arr[J] = 13 with arr[min-idx] = 10

arr[4] = 13 arr[1] = 10

13 < 10 false

so no change in min idx

~~second outer loop (i=1)~~

after completing first outer loop we need to swap them to first element in array

29, 10, 14, 37, 13

int Temp = arr[min_idx] → Temp = 10

arr[min_idx] = arr[i] → arr[1] = arr[0]

arr[i] = Temp → arr[0] = Temp

~~Temp = 29~~ 10 ~~arr[i] = 10~~

~~arr[min_idx] = 29~~ ~~arr[0] = 29~~

arr[1] =

after swapping our array looks like this

[10, 29, 14, 37, 13] unsorted

sorted — now the second outer loop starts for the ~~array~~ unsorted array

(i = 1) {29, 14, 37, 13}

initial state [10, 29, 14, 37, 13] min_idx = 1

inner loop iterations:

inner iteration 1 (J=2)

for (J=i+1; J<n, J++)

J = 1+1 2<5 J++

compare arr[J] with arr[min_idx]

arr[2] = 14 arr[1] = 29

14 < 29

so min_idx updated to arr[2] = 14

inner iteration 3

for (J=1+2)

J=3

37 < 14 false

so min_idx = 14

inner iteration 4

J = 1+3

J = 4

arr[4] = 13 arr[min] = 14

13 < 14 true

min_idx updated to 13

arr[4] = 13

next outer loop starts i=2 ..... continue

so this iterations continue till whole array is sorted

2. BubbleSort:

Bubble Sort is a Simple comparison-Based Sorting algorithm. it works repeatedly stepping Through the list comparing adjacent elements, and Swapping Them if They are in wrong order.

The Pass Through the list is repeated until the list is Sorted.

the algorithm gets it's name because Smaller elements "bubble" to the Top of the list with each iteration.

Algorithm Steps: How it works

1. Compare adjacent Elements: Begin with the first element. compare it with the next element.

2. Swap if necessary: if the current element is greater than the next, Swap them.

3. Continue Through the List: move to the next element, repeating the compare-and-Swap process until the end of array.

4. Repeat: perform the entire process for each element, progressively reducing the array's Size by excluding already Sorted elements.

5. Termination:

The algorithm ends when a complete pass results in no swaps, indicating the list is fully sorted.

*** The time complexity is $O(N^2)$ → worst complexity
But it can be improved averge. if array is already sorted Then no swaps will be needed and the time complexity can become $O(N)$

Example:

~~{13, 46, 24, 52, 20, 9}~~

{13, 25, 22, 9, 5}

Pass 1

compare 13 and 25, no swap needed because 13 < 25

compare 25 and 22, swap because 25 > 22

• after swap {13, 22, 25, 9, 5}

compare 25 and 9, swap because 25 > 9

• after swap {13, 22, 9, 25, 5}

compare 25 and 5, swap because 25 > 5

• after swap {13, 22, 9, 5, 25}

after first pass 25 is the large element in array

Pass 2:

compare 13 and 22, no swap need because 13 < 22

compare 22 and 9, swap because 22 > 9

after swap {13, 9, 22, 5, 25}

Compare 22 and 5, swap because 22 > 5

after swap {13, 9, 5, 22, 25}

after Second pass Now 22 is in correct possition.

we don't need to compare 25 for future passes.

Pass 3

compare 13 and 9, swap because 13>9

- after swap: {9, 13, 5, 22, 25}

compare 13 and, swap because 13>5

- after swap {9, 5, 13, 22, 25}

13 is in now correct position

Pass 4

compare 9 and 5, swap because 9>5

- after swap {5, 9, 13, 22, 25}

now all the elements in correct position and the array is sorted.

final sorted array:

{5, 9, 13, 22, 25}

code implementation for BubbleSort

```
int n = arr.length
for (int i=0; i<n-1; i++){
    for(int j=0; j<n-i-1; j++)
    if (arr[j] > arr[j+1]){
        //swap arr[j] and arr[j+1]
        int temp = arr[j];
        arr[j] = arr[j+1];
        arr[j+1] = temp;
```

3. Insertion Sort :

* insertion sort is a straightforward sorting algorithm that sequentially builds a final sorted array by inserting one item at a time in it's correct position
* while not as efficient for large datasets as algorithms like quicksort, heapsort or mergesort,
* it's simplicity and minimal number of swaps make it ideal for small datasets or as an element within more complex algorithms.

How it works:

1. Starting from the second element:
   the process begins with the second element, treating the first as already sorted.

2. Compare and insert:
   each current element is compared with those to it's left. if it is smaller, it's shifted left untill it reaches it's correct position.

3. Repeat until sorted:
   this procedure is repeated for every element untill the entire array is sorted.

Example

```
int n = arr.length;
for (int i=1; i<n; i++){
    int key = arr[i]
    int j = i-1;
    while (j>=0 && arr[j]>key) {
        arr[j+1]=arr[j];
        j=j-1
    }
    arr[j+1] = key
```

4. MergeSort:

* mergeSort is a popular and efficient Sorting algorithm that follows the Divide and Conquer paradigm.
* it divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves.
* The merge() function is used for merging two halves.
* the merge (arr, l, m, r) is a key process that assumes that arr[l...m] and arr[m+1....r] are sorted and merges the two sorted sub-arrays into one.

Steps of MergeSort:

1. Divide: find the middlepoint to divide the array into two halves.

$$\text{middle } M = \frac{l+h}{2}$$

2. Conquer: Recursively Sort the two halves:

- mergeSort (arr, l, m)
- mergeSort (arr, m+1, r)

3. Combine: merge the two Sorted halves.

pseudocode:

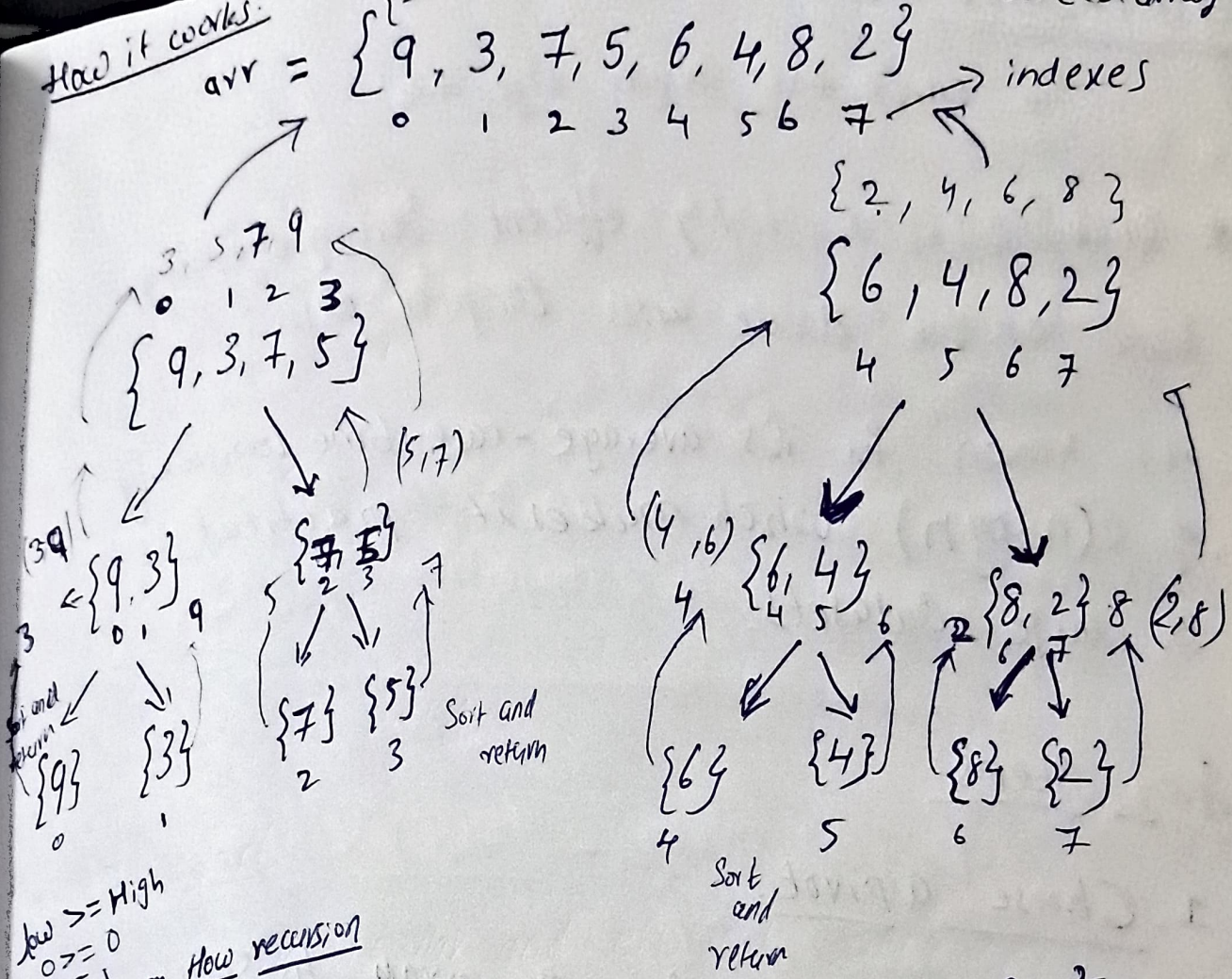
```
mergeSort (arr, l, r){
    if (l<r)
    {
        mid = (l+h)/2;
        mergeSort (l, mid);
        mergeSort (mid+1, r);
        merge (arr, l, mid, r);
    }
}
```

l = left
r = Right
mid = middle

How it works:

arr = {9, 3, 7, 5, 6, 4, 8, 2} → indexes

0 1 2 3 4 5 6 7



low >= High

0 >= 0

1 >= 1

Example on How recursion works

arr = {4, 2, 3, 5, 1}

```
mergeSort (arr, left, right){
    if (low >= right)
        return;
    mid = (left + right)/2
    mergesort (arr, left, mid)
    mergesort (arr, mid+1, right)
    ~~merge (arr, low, mid, high)~~
    mergSort (arr, left, mid, right)
```

```
mergesort (arr, left, right)
if (low >= right)
    return
mid = (left+right)/2
mergesort (arr, left, mid)
mergesort (arr, mid+1, right)
merg (arr, left, mid, right)
```

5. QuickSort

The divide and Conquer algorithm

* QuickSort is a highly efficient Sorting algorithm based on the divide and Conquer principle.

it's known for it's average-case time complexity of $O(n \log n)$ which makes it practical for large datasets.

Key steps:

1. Choose a pivot:

→ Select an element from the array to serve as the pivot.

→ common strategies include choosing the first, last, middle element, or a random element.

2. ~~Random~~ partition:

→ Re arrange the array so that all elements smaller than the pivot are placed to it's left and all elements greater than the pivot are placed to it's right.

→ The pivot ends up in it's final Sorted position.

3. Recursion:

→ Recursively apply steps 1 and 2 to the Subarrays to the left and right of the pivot.

→ Base case: if a sub array has only one element or is empty, it is already Sorted.

*** important Considerations:

- Pivot Selection:

→ The choice of pivot greatly influences QuickSort's performance.

→ in the worst case scenario, if the pivot is always the smallest or largest element, the algorithm degrades to $O(n^2)$ complexity.

→ Randomizing the pivot selection helps mitigate this.

- Stability:

→ QuickSort is not a stable sorting algorithm. This means that the relative order of elements with equal values may not be preserved after Sorting.

- Space Complexity:

→ although quick sort is generally done-in place it's recursive nature can lead to increased space usage on the call stack in the worst case scenario.

5

Benifits of QuickSort:

- **Efficiency:**
  QuickSort average-case time complexity is excellent, making it a popular choice for sorting large data sets.

- **inplace:** QuickSort can often be performed in-place, conserving memory.