ALGORITHMS AND DATA STRUCTURES

Lecture 1 SORTING

Problem defenition :

⇒ Input : Sequence of elements $x_1, x_2, \cdots x_n$

→ Ordering relation ($\leq$) on elements

⇒ Output : Sorted sequence of 'n' elements according to given order.

Eg: [15 , 3 , 27 , 49 , 23 , 18 , 6 , 1 , 31]

[1 , 3 , 6 , 15 , 18 , 23 , 27 , 31 , 49]

Algorithm

Selection Sort ()

1) Find smallest element → swap to 1st position
2) Find next smallest → swap to 2nd position
3) Iterate till array is sorted.

```
def SelectionSort (array):          n = length of array
    for i in range (0, n-1):
        minIndex = i ;
        for j in range (i+1, n):
            if array[j] < array[minIndex]:
                minIndex = j ;
    // Swap elements to i
    array[i], array[minIndex] = array[minIndex], array[i]

    return array
```

Write a python program and run doctest

Pseudo code [No formal rules]

```
SelectionSort [A]:

1> for i = 0 to n-2 do
2>     // find min in A[i ... n-1]
3>  minIdx = i
4>  for j = i+1 to n-1 do
5>      if A[j] < A[minIdx] then
6>          minIdx = j
7>  // Swap A[i] with min of A[i ... n-1]
8>   tmp = A[i]
9>   A[i] = A[minIdx]
10>  A[minIdx] = tmp
```

Analysis of an Algorithm :

(1) Show that an algorithm is correct
   → Algorithm computes correct output for every possible input
   → Algorithm terminates (total correctness)

(2) Analyse the running time and other properties
   → Efficiency of the algorithm
   → Performance and speed

Correctness of Selection Sort Algorithm

Properties after iteration (i) :

* Array contains same values as in beginning
* $A[0 \cdots i]$ is sorted correctly.
* values in $A[0 \cdots i] \leq$ values in $A[i+1, \cdots n-1]$

Proof by induction

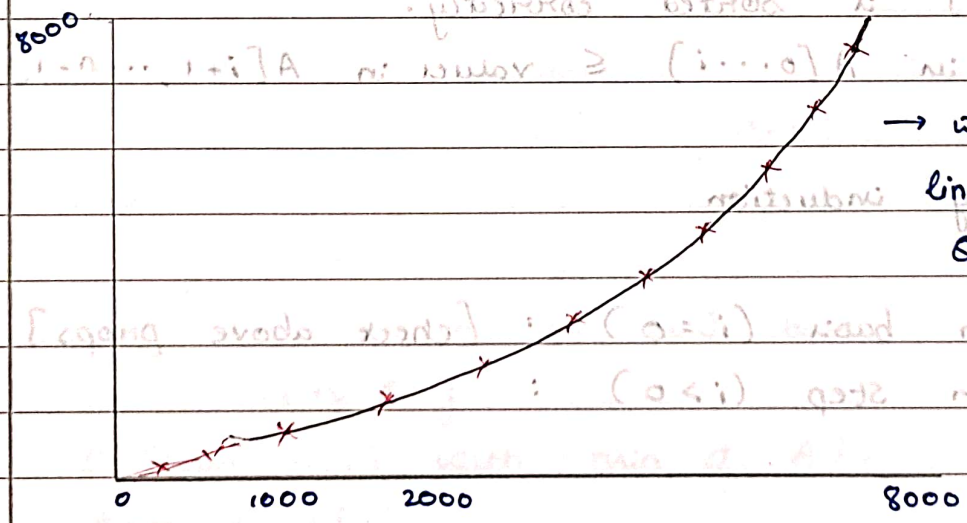
- Induction basis ($i = 0$) : [check above props]
- Induction step ($i > 0$) :

To check how fast the algorithm is

(a) Measuring Time : (python)

```
import time                                  milliseconds
Start_time = time.time();
        // Algorithm
run_time = (time.time() - start_time) * 1000;
```

```
def selection_sort_performance():
    for n in range(100, 8001, 100):
        array = [random.randint(0, 2*n) for
                 k in range(0, n)];
        StartiTime = time.time()
        selection_sort(array)
        run_time = (time.time() - Starti.time)% * 1000
        print("%d\t%.1f" % (n, run_time))
```




```
gnuplot
plot data.txt
```

Observation

→ Become slower over-proportionally when size of array increases

→ Time grew roughly quadratically with size of the array.

- Array 2× size ⟹ 4× to sort
- Array 3× size ⟹ 9× to sort

→ Running time of selection sort algorithm is indeed quadratic.

Algorithm

Insertion Sort

→ Beginning (prefix) of array is sorted
↳ one element per iteration

→ Step by Step each element will be sorted.

15	3	27	49	23	18	6	1	31

1) Start with 15, then 3 as 3 < 15 swap them
2) Move to 27, 0 to i+1 is sorted
3) Move to 49, 0 to i+1 is sorted
4) Move to 23, 23 < 27 < 49 Swap twice — 1 with 49, — 2 with 27
5) Continue in some way

Pseudocode

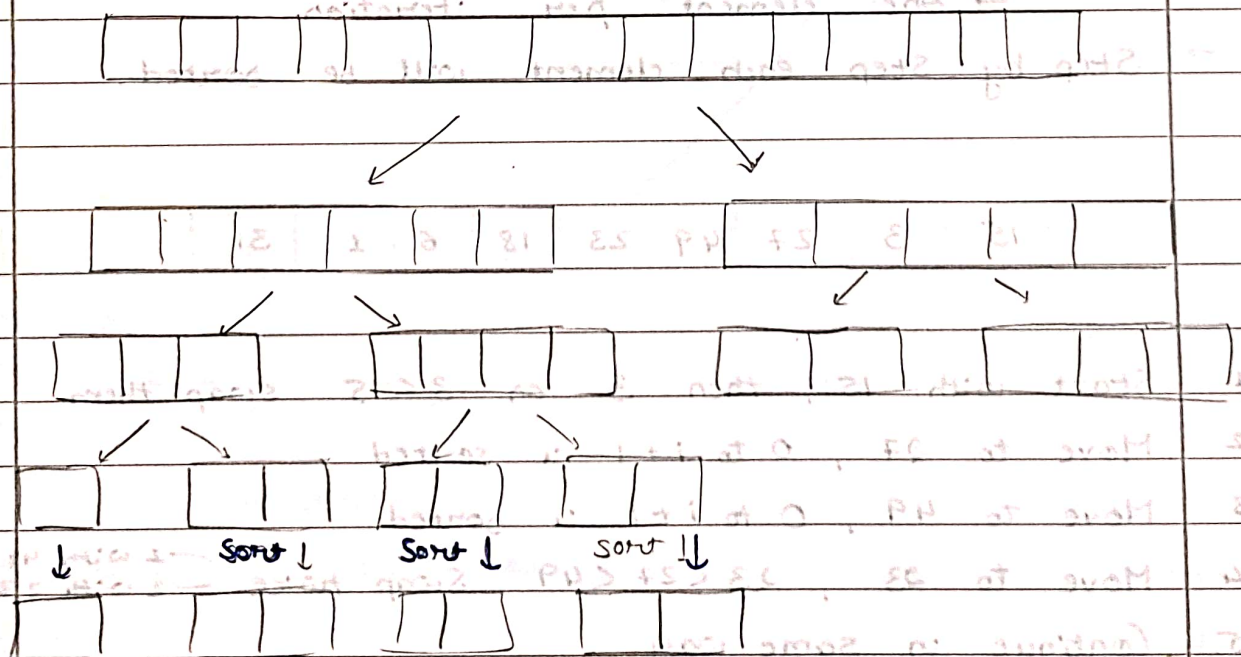
```
InsertionSort (A) :                    → at the end (0, i) will be already sorted.
1     for i = 0 to n - 2 do
2         // prefix A[0...i] is already sorted
3         pos = i + 1
4         while (pos > 0) and (A[pos] < A[pos - 1]) do
5             swap (A[pos], A[pos - 1])
6             pos = pos - 1
```

Algorithm

Quick Sort (Divide & Sort)

→ Divide array into two parts

* left part small elements – right large

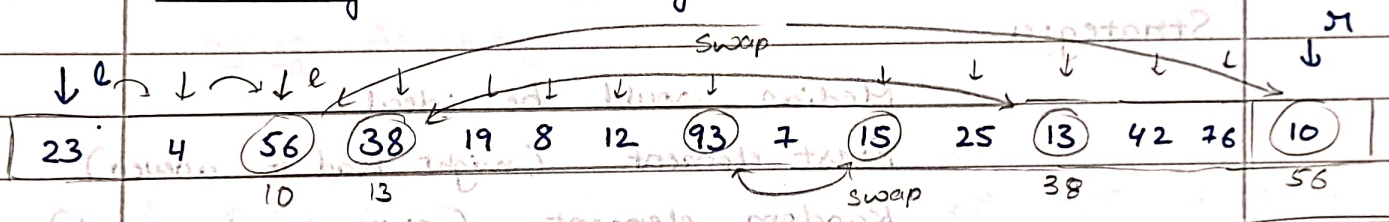


1> Divide array into left and right part
elements left ≤ element right

2.> (a) Sort element in left recursively
(b) Sort element in right recursively

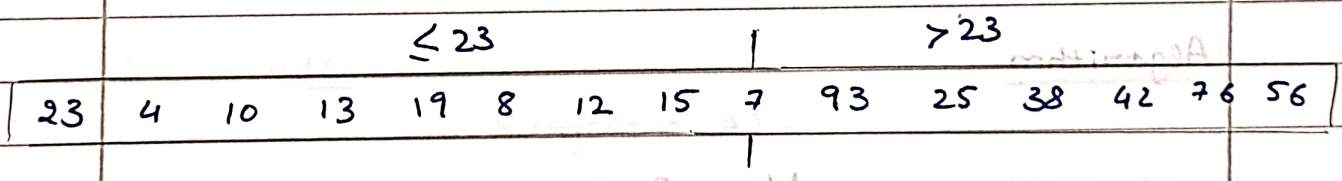
3> As soon as the sub problems become small enough such that sorting becomes Trivial recursion ends.

Partitioning the Array



pivot = 23

1. Increment l as long as A[l] ≤ pivot
2. Decrement r as long as A[r] > pivot
3. Swap when both are struck



Procedure

- Two variables l, r to iterate over the array from left and the right

→ Increment l until $A[l] > x$ (element goes to right)
→ Decrement r until $A[r] \leq x$ (elem goes to left)
→ Swap $A[l] \leftrightarrow A[r]$, increment l decrement r
→ Divide is done as soon as l and r meet

Choice of Pivot

This determines how large the two parts become when array is divided. The algorithm works best if size of two parts are almost equal.

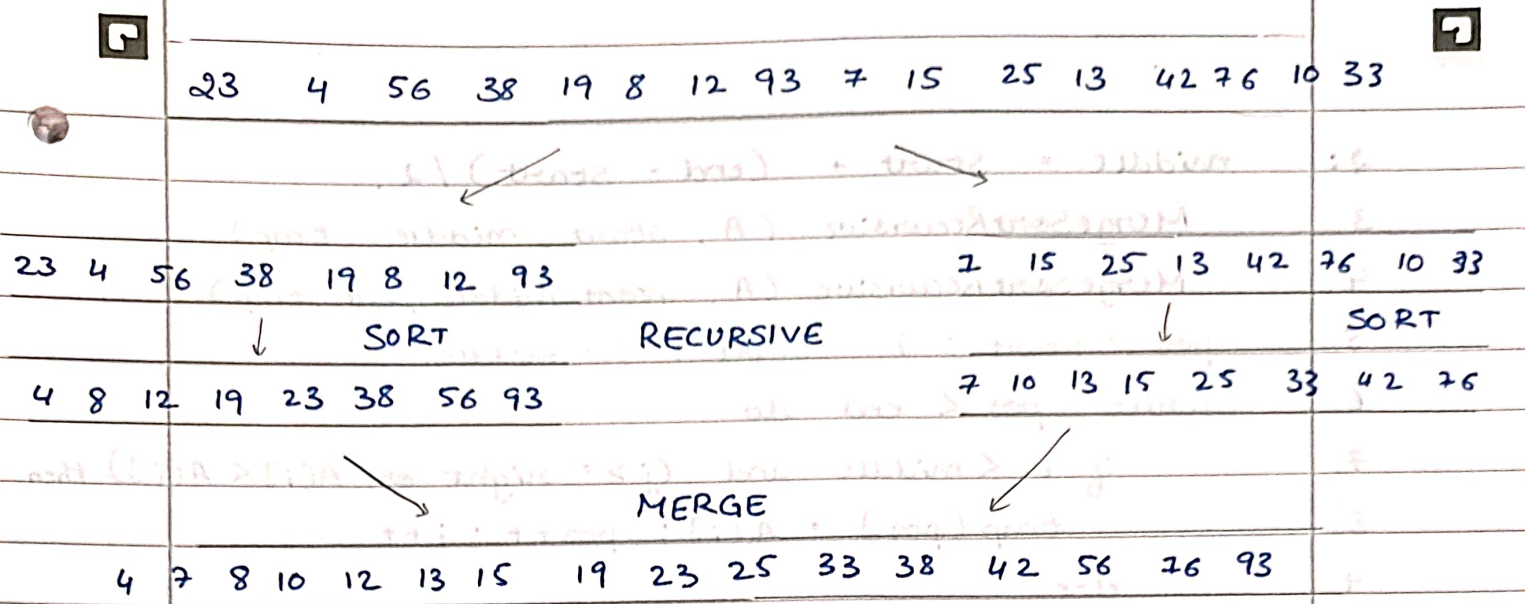
Strategies

→ Median would be ideal
→ First element (might lead to uneven)
→ Random element (gives roughly equal)
→ Median of ≥3 random elements.

Algorithm

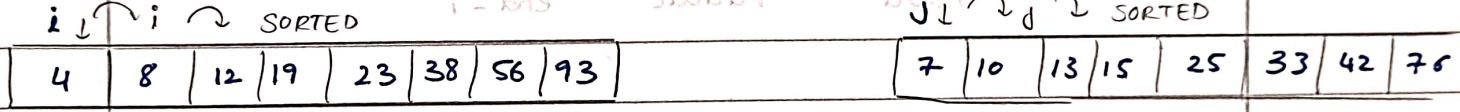
MergeSort

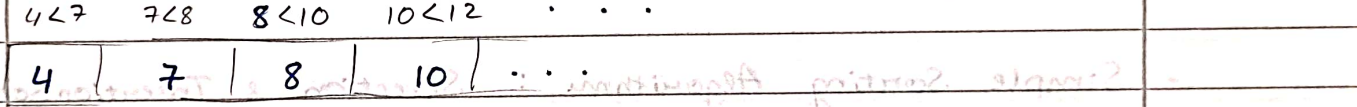
MergeSort is also based on divide & conquer.



* Divide trivial for MergeSort
* Merge requires more work

Merging Algorithm





Pseudo Code

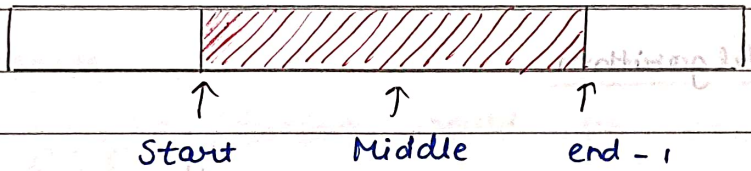
MergeSort (A) :

1. allocate array tmp to store intermediate results
2. MergeSort Recursive (A, 0, n, tmp)

MergeSortRecursive (A, start, end, tmp)

1. if end - start > 1 then

```
2) middle = start + (end - start) / 2.
3) MergeSortRecursive (A, start, middle, tmp)
4) MergeSortRecursive (A, middle, end, tmp)
5) pos = start ; i = start ; j = middle
6) while pos < end do
7)     if i < middle and (j >= right or A[i] < A[j]) then
8)         tmp[pos] = A[i]; pos++ ; i++
9)     else
10)        tmp[pos] = A[j] ; pos++ ; j++
11) for i = start to end - 1 do A[i] = tmp[i]
```



Summary :

* Simple Sorting Algorithm : Selection & Insertion Sort
* Recursive Sorting Algorithm : Quick & Merge Sort