Sorting

One of the most common tasks in data processing is sorting

For Ex: an array of employees often needs to be displayed in alphabetical order or sorted by salary.

→ Iteration based sorting algorithm (comparision based)
  → Selection Sort
  → Bubble Sort
  → Insertion Sort

→ Recursive sorting algorithm (comparision based)
  → Merge Sort
  → Quick Sort

→ Radix Sort (non-comparision based)

→ Properties of Sorting
  → Inplace & Outplace sort
  → Stable sort
  → Internal & External Sort

* Sorting is a technique to rearrange the elements of a list in ascending or descending order, which can be numerical, lexicographical or any user-defined order.

Internal and External Sort:-

In internal sort, the list of records is small enough to be maintained entirely in the physical memory for the duration of the sort.

In external sort, the list of records will not fit entirely into physical memory at once. In that case, the records are kept in disk files and only a selection of them are resident in physical memory at any given time. Eg. Merge Sort

Stable and Non-Stable Sorting:-

A sorting algorithm is stable if the relative order of elements with the same key value is preserved by algorithm.

Eg:- 5_a 2 5 1 5_b 6 5_c

1 5_a 5_b 5_c 6 2 5 — stable sorting

1 5_c 5_a 5_b 6 2 5 — unstable sorting

Inplace:- A sort algorithm is said to be an inplace sort if it requires only a constant amount (i.e. O(1)) of extra space during the sorting process.

Selection Sort :—

In selection sort, first element is compared with all remaining (n-1) elements. The smallest element is placed at the first location. Again the second element is compared with rest (n-2) elements and so on.

Demonstrate the selection sort results for each pass for following initial array of elements

A

21	6	3	57	13	9	14	10	2
0	1	2	3	4	5	6	7	8

in pass 1 (i = 0)

initially min_index = 0

find the index of minimum element in 1 to 8

min_index = 8

swap it with A[0] (A[0] ↔ A[8])

or

A[i] ↔ A[min_index]

0	1	2	3	4	5	6	7	8
2	6	3	57	13	9	14	10	21

pass 2 (i=1) search min index min_index = 2

swap (A[1] ↔ A[2])

0	1	2	3	4	5	6	7	8
2	3	6	57	13	9	14	10	21

sorted ← → unsorted

pass = 3 (i=2)

min_index = 2

swap (A[2] ↔ A[2])

0	1	2	3	4	5	6	7	8
2	3	6	57	13	9	14	10	21

pass 4 (i=3)

search min_index in (3 to 8)

min_index = 5

swap (A[i] ↔ A[min_index])

0	1	2	3	4	5	6	7	8
2	3	6	dg	13	57	14	10	21

Pass=5 (i=4)

0	1	2	3	4	5	6	7	8
2	3	6	9	13	57	14	10	21

pass = 6 (i=5)

min_index = 6

swap A[5] ↔ A[6]

0	1	2	3	4	5	6	7	8
2	3	6	9	13	14	57	10	21

pass = 7 (i=6)

min_index = 7

swap A[6] ↔ A[7]

0	1	2	3	4	5	6	7	8
2	3	6	9	13	14	10	57	21

pass = 8 (i=7)

min_index = 8

swap A[7] ↔ A[8]

2	3	6	9	13	14	10	21	57

function for Selection Sort!-

```
void SelectionSort(int A[], int n) {
    int j, min_index, temp;
    for (int i = 0; i < n-1; i++)
    {
        min_index = i;
        for (j = i+1; j < n; j++) {
            if (A[j] < A[min_index])
                min_index = j;
        }
    temp = A[i];
    A[i] = A[min_index];
    A[min_index] = temp;
} }
```

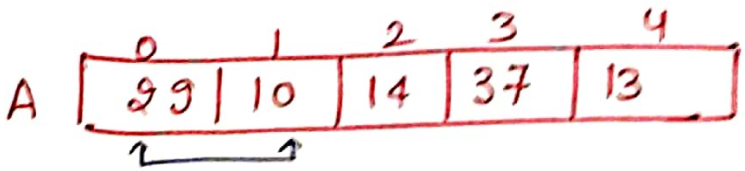
Bubble Sort!-

Large item is like "bubble" that floats to the end of the array.

function

```
void bubbleSort(int A[], int n) {
    int temp;
    for(int i=0; i<n-1; i++){
        for(int j=0; j<n-1-i; j++) {
            if (A[j] > A[j+1]) {
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
} } }
```

Illustration



Pass 1 (i=0)

$n-i-1 \Rightarrow 4$

- j=0: 10 29 14 37 13 — A[0] ↔ A[1]
- j=1: 10 14 29 37 13 — A[1] ↔ A[2]
- j=2: 10 14 29 37 13 — –
- j=3: 10 14 29 13 37 — A[3] ↔ A[4]

37 → bubbled

Pass 2 (i=1)

$n-i-1 \Rightarrow 3$

- j=0: 10 14 29 13 37 — –
- j=1: 10 14 29 13 37 — –
- j=2: 10 14 13 29 37 — A[2] ↔ A[3]

29 → bubbled 2

Pass 3 (i=2)

- j=0: 10 14 13 29 37 — –
- j=1: 10 13 14 29 37 — A[1] ↔ A[2]

Pass 4 (i=3)

- j=0: 10 13 14 29 37 — –

Time complexity

Best case — $O(n^2)$

Worst case — $O(n^2)$

Average case — $O(n^2)$

Insertion Sort!—

Similar to how most people arrange a hand of cards.

→ Start with one card in your hand

→ Pick the next card and insert it into its proper sorted order

→ Repeat previous step for all the card.

1st card 10 → | 10 |

2nd card 5 → | 5 | 10 |

3rd card K → | 5 | 10 | K |

A | 40 | 13 | 20 | 8 | 5 |

Iteration 1 | 40 |

Iteration 2 | 13 | 40 |

" 3 | 13 | 20 | 40 |

" 4 | 8 | 13 | 20 | 40 |

" 5 | 5 | 8 | 13 | 20 | 40 |

function. void InsertionSort (int A[], int n) {

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

Analysis :- Number of times inner-loop is executed depends on the input.

* Best Case :- The array is already sorted and (A[j] > key) is always false

→ No shifting of data is necessary

→ O(n)

* Worst-Case :- Array is reversely sorted (A[j] > key) is always true

→ Insertion always occur at the front

→ $O(n^2)$

Merge Sort

→ Merge Sort is based on the Divide and conquer paradigm.

→ Suppose we have to sort a array $A[p..r]$. Initially $p = 0$ and $r = n-1$; but these values changes as we recurse through sub-problems.

① Divide Step :- If the given array A has zero or one element simply return.

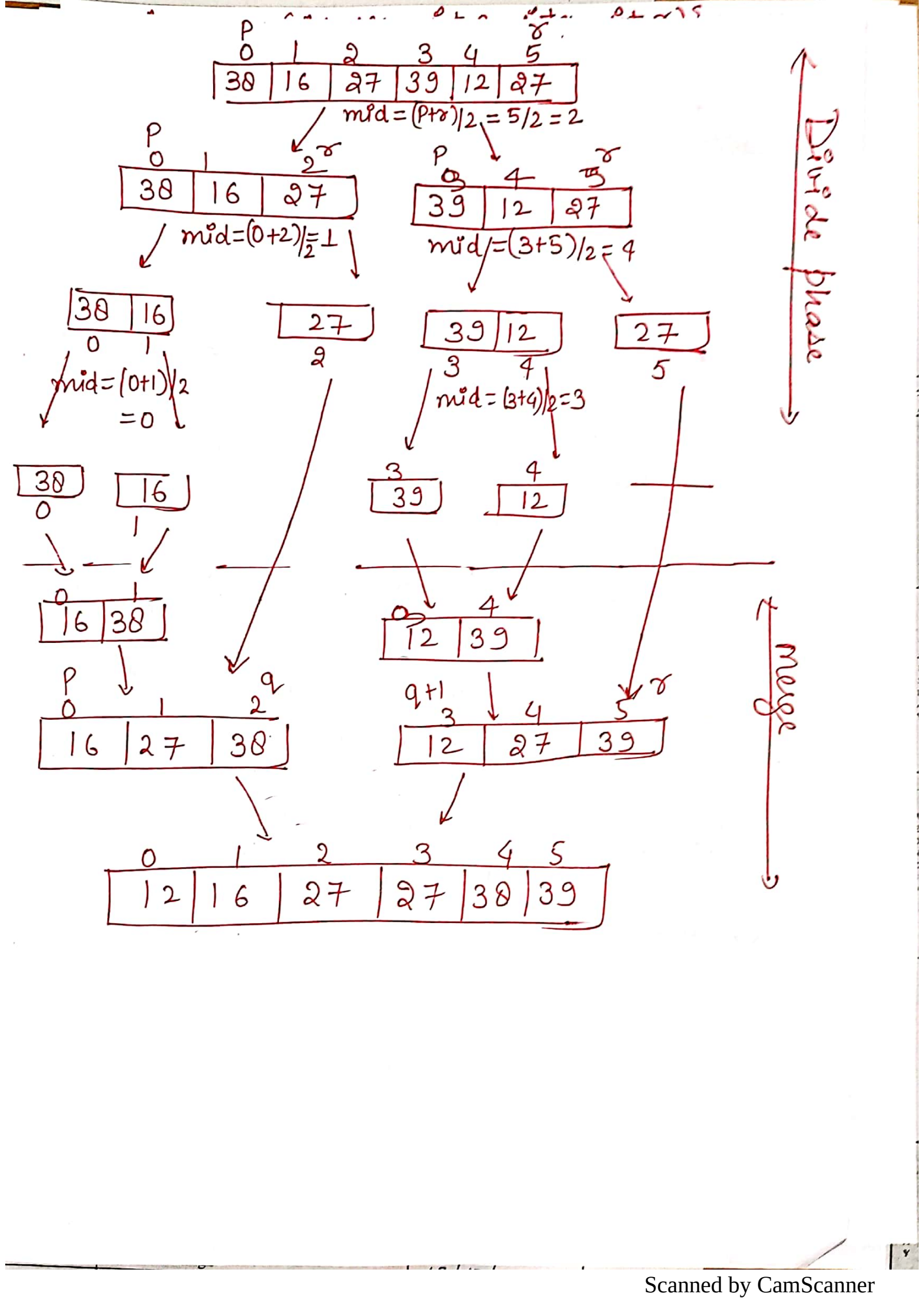
Otherwise, split $A[p..r]$ into two sub-array

$A[p..r]$

$A[p..q]$ $A[q+1..r]$ q is the middlepoint

② Conquer Step :- Conquer by recursively sorting the two sub-arrays.

③ Combine Step :- Merge the two sorted sub-arrays $A[p-q]$ and $A[q+1..r]$ into a sorted sequence.



```
void merge(int A[], int p, int q, int r){
  int i, j, k;
  int n1 = q-p+1;        // UB-LB+1
  int n2 = r-q;

  int L1[n1], R[n2];

  for (i=0; i<n1; i++)
      L[i] = A[p+i];

  for (j=0; j<n2; j++)
      R[j] = A[q+1+j];

  i=0; j=0; k=p;

  while (i<n1 && j<n2)
{   if (L[i]<=R[j]){
        A[k]=L[i];
        i++; k++; }
    else{
        A[k]=R[j];
        j++; k++; } }

  while (i<n1){
        A[k]=PL[i];
        i++; k++; }

  while (j<n2){
        A[k]=R[j];
        j++; k++; }
}
```

```
void mergeSort (int A[], int p, int r){
 if(p<r){
    int mid = (p+r)/2;
     mergeSort (A, p, mid);
     mergeSort (A, mid+1, r);
      merge (A, p, mid, r);
     }
}
```

→ Time Complexity – $O(n \lg n)$

→ optimal comparison based sorting method.

→ $O(n)$ extra storage needed.

Final answer.

Quick Sort :- Like Merge Sort, Quick Sort is a Divide and Conquer Algorithm.

→ choose an item pivot and partition the items of A[p...r] into two parts

① item that are smaller than pivot [A[p..q-1]]

② item that are greater & equal than pivot. [A[q+1...r]]

Recursively sort the two parts.

```
int partition (int A[], int p, int r) {
    int temp;
    int pivot = A[r];
    int i = p-1;

    for (int j=p; j< r; j++)
      {
          if [A[j] < pivot) {
                i++;
                temp = A[j];
                A[j] = A[i];
                A[i] = temp; }

    Af temp = A[i+1];
       A[i+1] = A[r];
       A[r] = temp;

return (i+1);
}
```

```
void quickSort(int A[], int p, int r){
    if (p<r){
        int pivot = partition(A, p, r);
        quickSort(A, p, pivot-1);
        quickSort(A, pivot+1, r);
    }
```

Analysis :-

Best case & Average case — $O(n \log_2 n)$

Worst case :- when array is already in ascending order

$\longrightarrow O(n^2)$

7.a Write algorithm for quick sort. Trace your algorithm on the following data to sort the list:

2, 13, 4, 21, 7, 56, 51, 05, 59, 1, 9, 10

A =

0	1	2	3	4	5	6	7	8	9	10	11
2	13	4	21	7	56	51	05	59	1	9	10

low = 0, high = 11 pivot = A[high] = 10

i = -1

Traverse elements from j = low to high - 1

j = 0; since A[j] <= pivot, then i++ and swap A[i] & A[j]

i = 0

A =

											pivot
2	13	4	21	7	56	51	05	59	1	9	10

i=0 j=1

Since A[j] > pivot then do nothing

for i = 0 & j = 2

since (A[j] <= pivot) ~~swap~~ then i++ and swap A[i] & A[j]

A =

0	1	2	3	4	5	6	7	8	9	10	pivot
2	4	13	21	7	56	51	05	59	1	9	10

i=1 j=3

Since A[j] > pivot, then do nothing

for i = 1 and j = 4; then i++ and swap A[i] & A[j]

A =

0	1	2	3	4	5	6	7	8	9	10	11 pivot
2	4	7	13	21	56	51	05	59	1	9	10

i=2 j=5

A=

0	1	2	3	4	5	6	7	8	9	10	11
2	4	7	1	21	56	51	85	59	13	9	10

i=3 j=10

since A[j] <= pivot then i++ & swap A[i] and A[j]

A =

0	1	2	3	4	5	6	7	8	9	10	11
2	4	7	1	9	56	51	85	59	13	21	10

Finally, place the pivot at correct position by swapping A[i+1] and A[high]

A =

0	1	2	3	4	5	6	7	8	9	10	11
2	4	7	1	9	10	56	85	59	13	21	56

pivot

A =

0	1	2	3	4	5	6	7	8	9	10	11
2	4	7	1	9	10	51	85	59	13	21	56

pivot pivot

A =

0	1	2	3	4	5	6	7	8	9	10	11
2	4	7	1	9	10	51	13	21	56	59	85

pivot pivot pivot

0	1	2	3
1	4	7	2

pivot

1	2	3
2	7	4

pivot

2	3
4	7

6	7	8
13	21	51

pivot pivot

6	7	8
13	21	51

10	11
59	85

pivot

10	11
59	85

0	1	2	3	4	5	6	7	8	9	10	11
1	2	4	7	9	10	13	21	51	56	59	85

Radix Sort

No comparison between data is ~~not~~ needed.

The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit.

Ex

170
45
75
90
802
24
2
66

Sort using least significant bit (1s place)

170
90
802
02
24
45
75
66

Sort by next digit (10s place)

802
002
024
045
066
170
075
090

Sort by most significant digit (100s place)

002
24
45
66
75
90
170
802

final answer.

	Worst case	Best case	In-place?	Stable?
Selection Sort	$O(n^2)$	$O(n^2)$	Yes	No
Insertion Sort	$O(n^2)$	$O(n)$	Yes	Yes
Bubble Sort	$O(n^2)$	$O(n^2)$	Yes	Yes
Merge sort	$O(n \log_2 n)$	$O(n \log_2 n)$	No	Yes
Quick Sort	$O(n^2)$	$O(n \log_2 n)$	Yes	No
Radix Sort	$O(dn)$	$O(dn)$	No	Yes