18/08/22 Sorting techniques -

→ Critera for Analysis

i) Number of comparisons
It decides the time complexity.

ii) No. of swaps

iii) Adaptive -
If any sorting method takes less time or min^m time over already ~~mi~~ sorted list then we call that algorithm as adaptive.

iv) Extra Memory

v) Stable -
If a sorting algorithm is preserving the order of duplicate elements in the sorted list then that algorithm is called as stable.

Eg:-

Name-	A	B	C	D	E	F	G
marks-	5	8	6	4	6	7	10

After sorting, if we get like this then it is called stable -

Name -	D	A	C	E	F	B	G
marks -	4	5	6	6	7	8	10

Here 'C & 'E are preserving their order as previous.

If we get like this then it is not stable -

Name -	D	A	E	C	F	B	G
marks -	4	5	6	6	7	8	10

Here 'C & 'E are not preserving their order as previous

Sorting Algorithm

- Comparision based sorts
  - $O(n^2)$
    1. Bubble
    2. Insertion
    3. Selection
  - $O(n \log n)$
    4. Heap sort
    5. Merge sort
    6. Quick "
    7. tree "
  - $O(n^{3/2})$ → 8. Shell "
- Index based sorts
  - $O(n)$
    9. Count "
    10. Bucket / Bin sort
    11. Radix sort

They are faster but takes lots of space

1 Bubble Sort

A | 8 | 5 | 7 | 3 | 2 |
0 1 2 3 4

$n = 5$

1st Pass

8	5	5	5	5
5	8	7	7	7
7	7	8	3	3
3	3	3	8	2
2	2	2	2	8

Comparision = 4
Swap : 4

2nd Pass

5	5	5	5	5
7	7	3	3	3
3	3	7	2	2
2	2	2	7	7
8	8	8	8	8

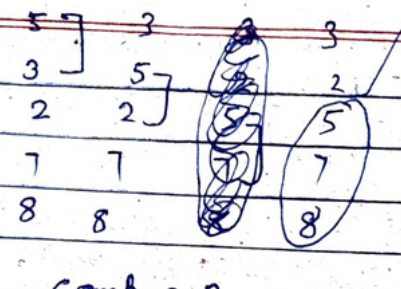
sorted

Comp = 3
Swap = 3

→ When all element compared once then it is called as Pass.

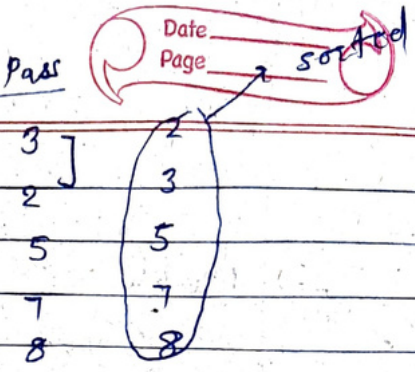
NOTE - Intermediate result of Bubble sort is useful for us.

3rd pass



comp = 2
swap = 2

4th Pass



comp = 1
swap = 1

→ No. of Passes (4) = (n-1) Pass

→ No. of ~~swap~~ comp {1+2+3+4} → 1+2+3+4 --- n-1

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

time - $O(n^2)$

→ No. of swap (1+2+3+4)

for n elements -

$1+2+3+4+\cdots n-1$

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

```
→ Void BubbleSort (int A[], int n)
  {  int flag;
     for (i=0; i<n-1; i++)
     {  flag = 0;
        for (j=0; j<n-1-i; j++)
           if (A[j] > A[j+1])
           {  swap (A[j], &A[j+1]);
              flag = 1;
           }
        if (flag == 0);
        break;
  }
```

Bubble sort time-

min - $o(n)$ → When list is already sorted then it takes time $o(n)$, which is less than n^2, so it is adaptive (made by us)

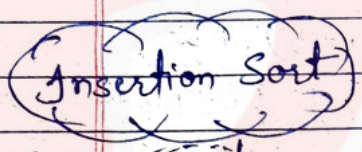
max - $o(n^2)$

→ Bubble sort is also stable.

```
8 8 8
8 8 3
3 3 8
5 5 5
4 4 4
```

Here blue 8 and black 8 does not change its position after sorting so it is stable.

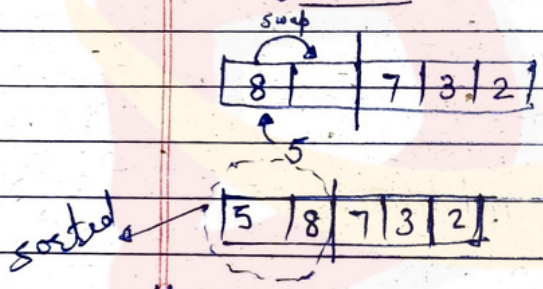
Insertion Sort



→ If we assume that only one element in the list then it is always sorted.

→ To sort all the elements, take an element of right side and insert it in left side in sorted way.

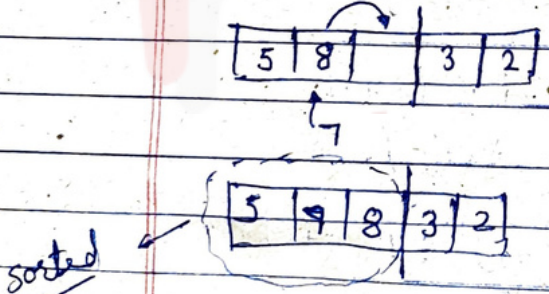
1st pass- Insert 5



1 comp

1 swap

2nd pass- Insert - 7

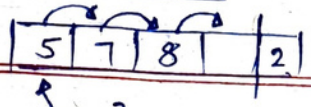


2 comp (max)

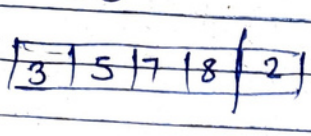
2 swap

3rd Pass

Insert - 3



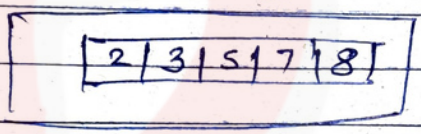
3 comp
3 swap



4th Pass

Insert - 2

4 comp
4 swap



→ No. of passes = 4
for 'n' elements : (n-1) Pass

→ No. of comp = 1+2+3+4
for 'n' elements = 1+2+3+4+ ... n-1

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

→ No. of swap = 1+2+3+4
for 'n' = 1+2+ ... n-1

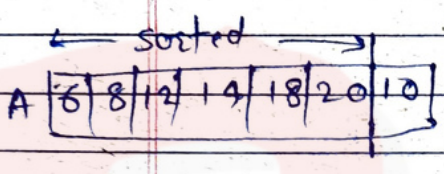
$= \frac{n(n-1)}{2} = O(n^2)$

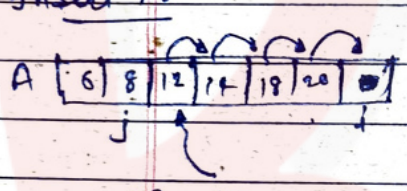
→ It's intermediate result not gives any useful result.

→ Insertion sort is more compatible with linked list not array.

* Program for insertion sort -

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





→ for already sorted list it takes - O(n)

So, it is adaptive. (Signature)

time -	min	max
	O(n)	$O(n^2)$
space -	O(1)	$O(n^2)$

→ Insertion sort is also stable.

* Comparing of Bubble and Insertion sort -

Date
Page

	Bubble	Insertion	case
min comp	$\Omega(n)$	$\Omega(n)$	→ when already in ascending order
max^{m} comp	$O(n^2)$	$O(n^2)$	→ Descending order
min swap	$O(1)$	$O(1)$	Ascending order
max swap	$\Theta(n^2)$	$O(n^2)$	Descending "
Adaptive	✓	✓	
stable	✓	✓	
Linked list	No	Yes	
for 'K' passes useful or not	Yes	No	

Selection sort → In this sorting, firstly we select the first position and finding the suitable element or no. for this position. And this is done for all position.

→ In this sorting we need three Pointers -

i → Points the current position

j → finding or checking the suitable element for current position one by one.

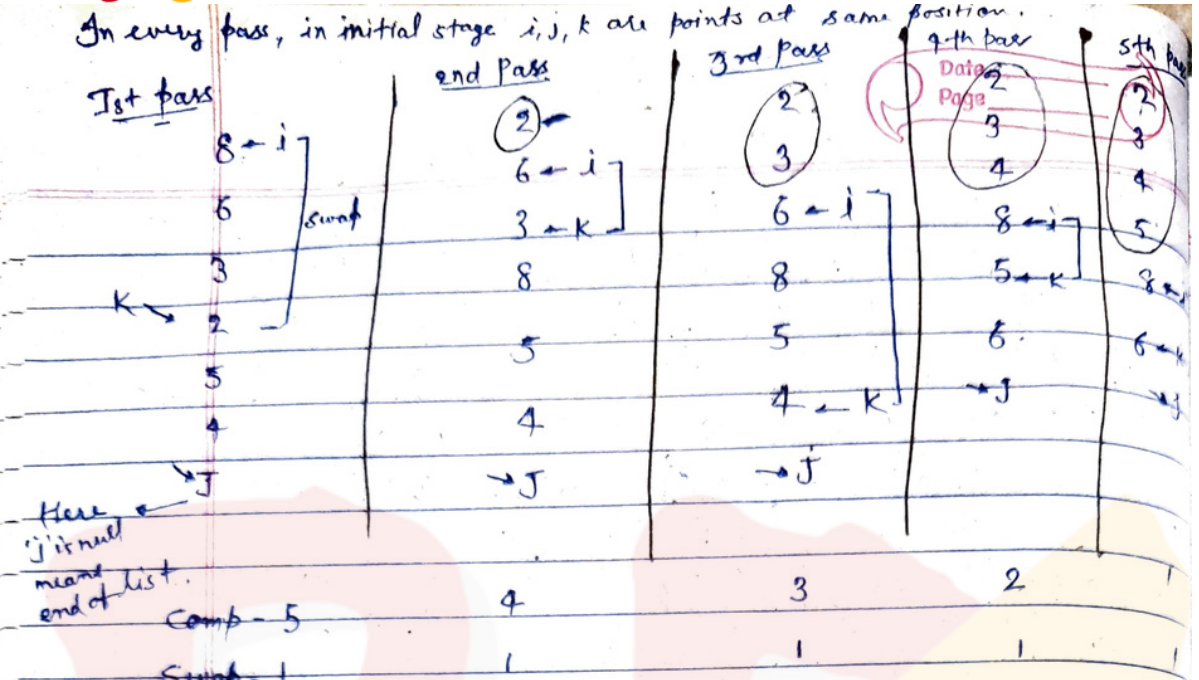
K → It points the suitable element which is found by 'j' pointer.

Eg:- A [8 | 6 | 3 | 2 | 5 | 4]
0 1 2 3 4 5

j → K → 8 ← i
6
3
2
5
4

→ Initial stage

In every pass, in initial stage i, j, k are points at same position.



Comp - 5 | 4 | 3 | 2 | 1

Swap - 1 | 1 | 1 | 1 | 1

→ No. of comparisions = 1+2+3+4+5

for 'n' = 1+2+ - - - n-1 = $\frac{n(n-1)}{2}$ times $O(n^2)$

→ No of swaps = 1+1+1+1+1 = 5

for 'n' = (n-1) = O(n)

→ It is also gives useful result for 'k' passes means intermediate result are also benifially

Program

```
Void SelectionSort (int A[], int n)
{ int i,j,k;
  for(i=0; i<n-1; i++)
  { for(j=k=i, j<n; j++)
    { if (A[j] > A[k])
        k = j;
    }
    swap (A[i], A[k]);
  }
}
```

Sorted
2
3
4
5
6
8

* Selection sort is not adaptive because in sorted list. it takes $o(n^2)$ time. Means ~~what~~ list is sorted or not sorted it always takes $O(n^2)$ time.

* Selection sort is ~~stable~~ also not stable.

Quick Short, Eg:- 20 10 30 (50) 90 70 60 (Here '50' is placed at right position)

→ It uses partitioning procedure.

Initial stage { Pivot → (50) 70 60 90 40 80 10 20 30 ∞
i j

→ Here 'i' is finding the greater element (compare than pivot) and 'j' is finding the smaller element of the list.

→ When 'i' finds a greater element than pivot, and 'j' finds a smaller element than pivot, then both are stopped and interchanged.

(50) 70 60 90 40 80 10 20 30 ∞
i — swap — j

(50) 30 60 90 40 80 10 20 70 ∞
i — j

(50) 30 20 90 40 80 10 60 70 ∞
i — j

(50) 30 20 10 40 80 90 60 70 ∞
j i

When 'i' finds a smaller element than pivot then it doesn't stopped, continues to found greater element.

Eg:- From above, when 'i' find '40' which is smaller than pivot then it moved ahead to finding greater element.

Similarly, when 'j' finds any greater element then it is moves on.

→ When i and j are crossed each other, it means all elements are checked.

→ And after that swap the pivot element with 'j' elem.

(40 30 20 10) (50) (80 90 60 70 ∞)

↓ partitioning position.

→ Same process for other elements.

* If elements are in either ascending order or descending order, quick short will take $O(n^2)$ time.

Best case - if Partioning is in middle.

time - $O(n \log n)$

Worst case - if partioning is in any end. (already sorted)

time - $O(n^2)$

Avg cas - $O(n \log n)$

Program

```
int Partition(int A[], int l, int h)
{
    int Pivot = A[l];
    int i = l, j = h;
    do
    {
        do {i++;} while(A[i] <= Pivot);
        do {j--;} while(A[j] > Pivot);
        if (i < j)
            swap(A[i], A[j]);
    } while(i < j);
    swap(A[l], A[j]); return j;
}
```

23/06/22

→ If we select middle element as pivot then -

Best case - $O(n \log n)$ → sorted list

Worst case - $O(n^2)$ → Partitioning at any end

→ In the quick sort we can select any random element of the list, so it is called randomise quick sort.

* Difference b/w selection sort and quick sort -

→ Selection Sort -

In this sort, we select the position and find out a element for this element position

→ Quick sort -

In this sort, we select a element and find out the suitable position for this element.

→ So, quick sort is also called as selection exchange sort.

* Merging

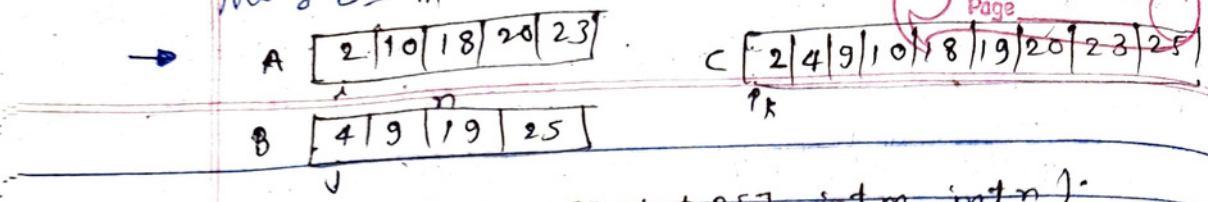
Some important topics for merge sort -

i) Merging 2 list (Array)

ii) Merging 2 list in single Array

iii) Merging multiple list

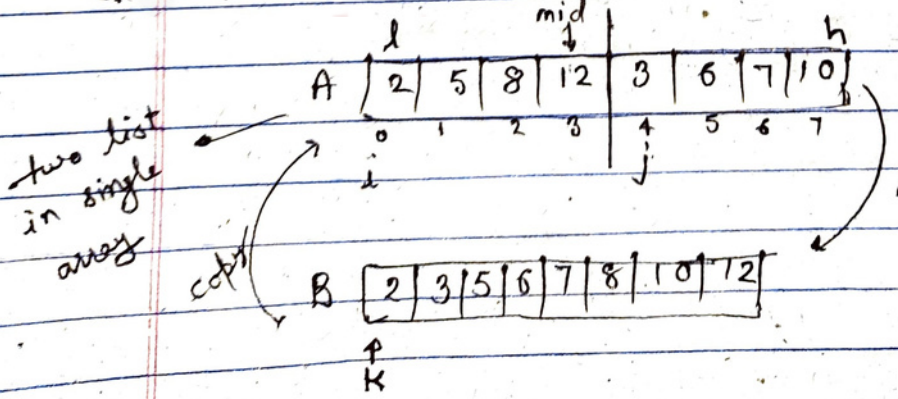
Merging two list –




```
void Merge (int A[], int B[], int m, int n);
{
    int i, j, k;
    i = j = k = 0
    while (i<m && j<n)
    {
        if (A[i] < B[j])
            c[k++] = A[i++];
        else
            c[k++] = B[j++];
    }
    for (; i<m; i++)
        c[k++] = A[i];
    for (; j<n; j++)
        c[k++] = B[j];
}
```

time - $\theta(m+n)$

* If more than one list are there in single array



```
Void Merge (int A[], int l, int mid, int h)
{
    int B[h+1];
    i = l; j = mid+1; k = l;
    while (i <= mid && j <= h)
    {
        if A[i] < A[j]
            B[k++] = A[i++];
        else
            B[k++] = A[j++];
    }
    for (; i <= mid; i++)
        B[k++] = A[i];
    for (; j <= h; j++)
        B[k++] = A[j];
    for (i = l; i <= h; i++)
        A[i] = B[i];
}
```

* Merging Multiple list -

If we merge 4 list in single array then it is called as 4-way merging.

Similarly if we merge 'm' list then it is called as 'm-way merging'.

Date
Page

Eg:-

List-1	List-2	List-3	List-4
A	B	C	D
2	3	5	8
5	6	9	16
15	18	12	20

merge → E

merge → F

E, F → G } Merged list

* Iterative Merged Sort -

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

→ To sort this list, using merge sort, we assume that each element is a list.

→ It means, there are 8 list are present and each list contains only one element

→ So we have to merge 8 list. (m-way merge)

n : 8

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

Log n passes

$\log_2 8 = 3$

1st pass: 3 8 | 4 7 | 2 9 | 5 6 — n

2nd pass: 3 4 7 8 | 2 5 6 9 — n

3rd pass: 2 3 4 5 6 7 8 9 — n

log n times

= O(n log n)

```
Void MergeSort (int A[], int n)
{  int P, i, l, mid, h;
   for (P=2; P<=n; P=P*2)
   {
      for (i=0 ; i+P-1 <n; i=i+P)
      {  l = i ;
         h = l+P-1;
         mid = (l+h)/2 ;
         Merge ( A, l, mid, h);
      }
      if (P/2 <n)
         merge (A, 0, P/2, n-1);
   }
```

* Recursive Merged sort -

27/06/22

Count Sort

A [6|3|9|10|15|6|8|12|3|6]
0 1 2 3 4 5 6 7 8 9

To sort this list using count sort, we need an another array and size of this array should be equal to the largest element of given array. It's all element is initialized with '0'.

In this array, largest element is '15', so take -

count [0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0]
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

→ Scan the all element of given array one by one and whatever the element you get, go at the same index of count array and increment it by one.

A [6|3|9|10|15|6|8|12|3|6] → n
0 1 2 3 4 5 6 7 8 9
j

count [0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0]
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
i

After that copy all index no. as many time they increased. until they become zero.

(In the count array, we get the number of occurences of each number presented in given array

A [3|3|6|6|6|8|9|10|12|15]

time - (n + n) → O(n)

```
Void countSort(int A[], int n)
{  int max, i,
   int * c;   int i=0, j=0
   Max = findMax(A,n) ;
    c = new int [max+1];
for ( i=0; i<max+1; i++)
      c[i] = 0;
for (i=0; i<n; i++)
   {
    c[A[i]]++;
   }
while ( i<= max+1)
{   if (c[i]>0)
      {   A[j++] = i;
          c[i]--;
      }
else
      i++;
}
```

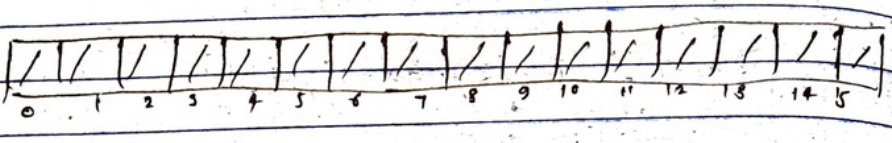
* Bucket/Bin Sort →

Same as count sort.

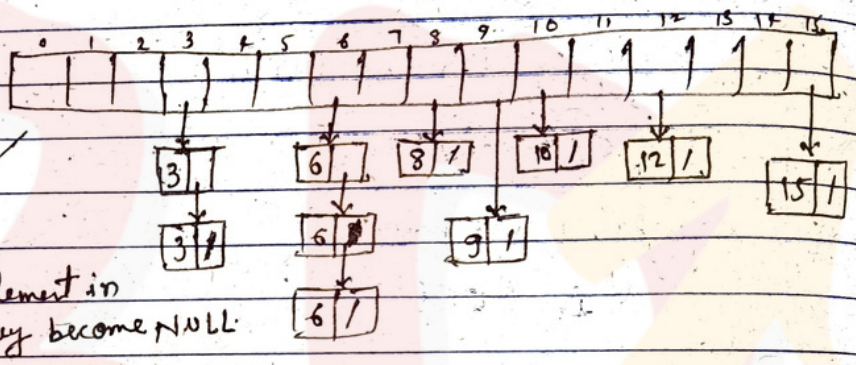
Date
Page

A | 6 | 8 | 3 | 10 | 15 | 6 | 9 | 12 | 6 | 3 |
0 1 2 3 4 5 6 7 8 9

Bins



Array of pointers (All elements are filled with NULL)



Insert all element in array 'A' untill they become NULL

A | 3 | 3 | 6 | 6 | 6 | 8 | 9 | 10 | 12 | 15 |

```
Void BinSort(int A[], int n)
{ int max, i, j;
  Node **Bins;
  Max = findMax(A, n);
  Bins = new Node *[Max+1];
n [ for(i=0; i<max+1; i++)
      Bins[i] = NULL;
n [ for(i=0; i<n; i++)
    {
      Insert(Bin[A[i]], A[i]);
    }
  i=0; j=0;
n [ while(i<max+1)
    {
      while(Bins[i] != NULL)
      {
        A[j++] = Delete(Bin[i]);
      }
      i++;
```

Time = n+n+n
= O(n)