Sorting Techniques -

We know that sorting is a process of arrangement of data / nodes in ascending or descending order.

We have total 8 different sorting techniques which has different algorithms and therefore different time complexities and space complexities.

Following are the commonly used sorting techniques :

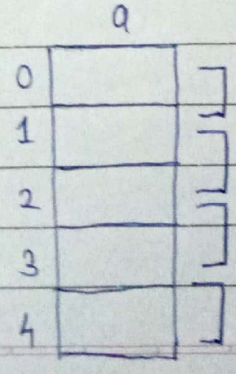
- Bubble sort
- Selection sort
- Insertion sort
- Merge sort
- Quick sort
- Radix sort
- Shell sort
- Heap sort

1) Bubble Sort :

This technique is slowest among all other techniques ie. have highest time complexity.

Principle of Bubble sort -

Each element is compared with it's exact next element and swapping is done based on ascending or descending order.



a[i] ↔ a[i+1]

```
#
#
void main()
{
  int a[10], i, temp, pass;

  printf("Enter 10 elements \n");
  for (i=0; i<=9 ; i++)
  {
    scanf("%d", &a[i]);
  }

  for (pass = 0; pass<=8 ; pass++)
  {
    for( i=0; i<=8-pass ; i++)
    {
      if( a[i] > a[i+1] )
      {
        temp = a[i];
        a[i] = a[i+1];
        a[i+1] = temp;
      }
    }
  }
  printf("sorted elements are: \n");
  for (i=0; i<=9; i++)
  {
    printf("%d \n", a[i]);
  }
  getch();
}
```

Here, we have shrinked the checking condition of array as at last elements will be greatest so no need to go there to compare with other elements so if pass = 2 then 8-2= 6. so last 2 elements no need to compare.

2) Selection sort :

This technique is faster than bubble sort.

Principle of selection sort -

The Pivot element / selected element is compared with all its following elements and swapping is done based on ascending / descending.

#

#

```
void main ()
{
  int a[10], i, temp, pivot_ind ;

printf("Enter 10 elements \n");
for (i=0; i<=9; i++)
{
  scanf ("%d", &a[i]);
}

for (pivot_ind = 0 ; pivot_ind <=8 ; pivot_ind ++ )
{
   for( i= pivot_ind +1 ; i<=9 ; i++ )
   {
      if( a[pivot_ind ] > a[i] )
       {
          temp = a[ pivot_ind ];
          a[pivot_ind ] = a[i];
          a[i] = temp;
       }
} }
```

```
printf("sorted elements are: \n");
for(i=0; i<=9; i++)
    {
    printf("%d \n", a[i]);
    }
getch();
}
```

5:57
C++

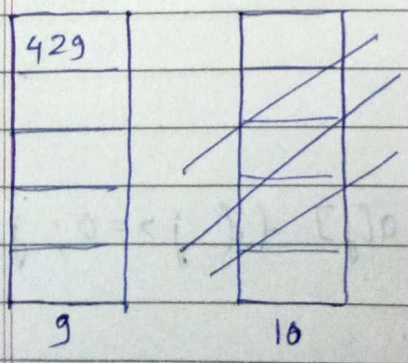
4) Radix sort or Bucket sort :

Principle of radix sort :
In each pass we distribute the element into bucket of its respective digit

Pass I :

347	683	500	195	429	877	75	240	603	768	301
0	1	2	3	4	5	6	7	8	9	10

R0	R1	R2	R3	R4	R5	R6	R7	R8
0 301	0	0 ~~683~~	0 683	0	0 195	0	0 347	0 768
1	1	1 ~~603~~	1 603	1	1 75	1	1 877	1
2	2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	4	4



(used while sorting answer papers)

3) Insertion sort :

Insertion

Principle of ~~selection~~ sort -

Insert the element at its best suitable position by comparing it with it's above elments.

Refer following program :

#
#

```
void main ()
{
   int a[10] , i, j, temp;

  Printf (" Enter 10 elements \n");
   for ( i=0; i<=9 ; i++)
   {
      scanf ("%d", &a[i]) ;
   }

for ( i=0; i<=8 ; i++)
 {
     temp = a[i];
     for (j= i-1; temp < a[j] && j>=0; j--)
     {
        a[j+1] = a[j];
     }
     a[j+1] = temp;

 }
```

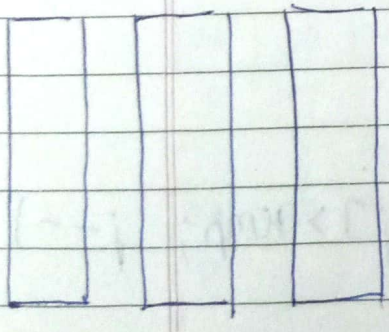
Pass II :

500	240	301	683	603	195	75	347	877	768	429
0	1	2	3	4	5	6	7	8	9	10

Pass III

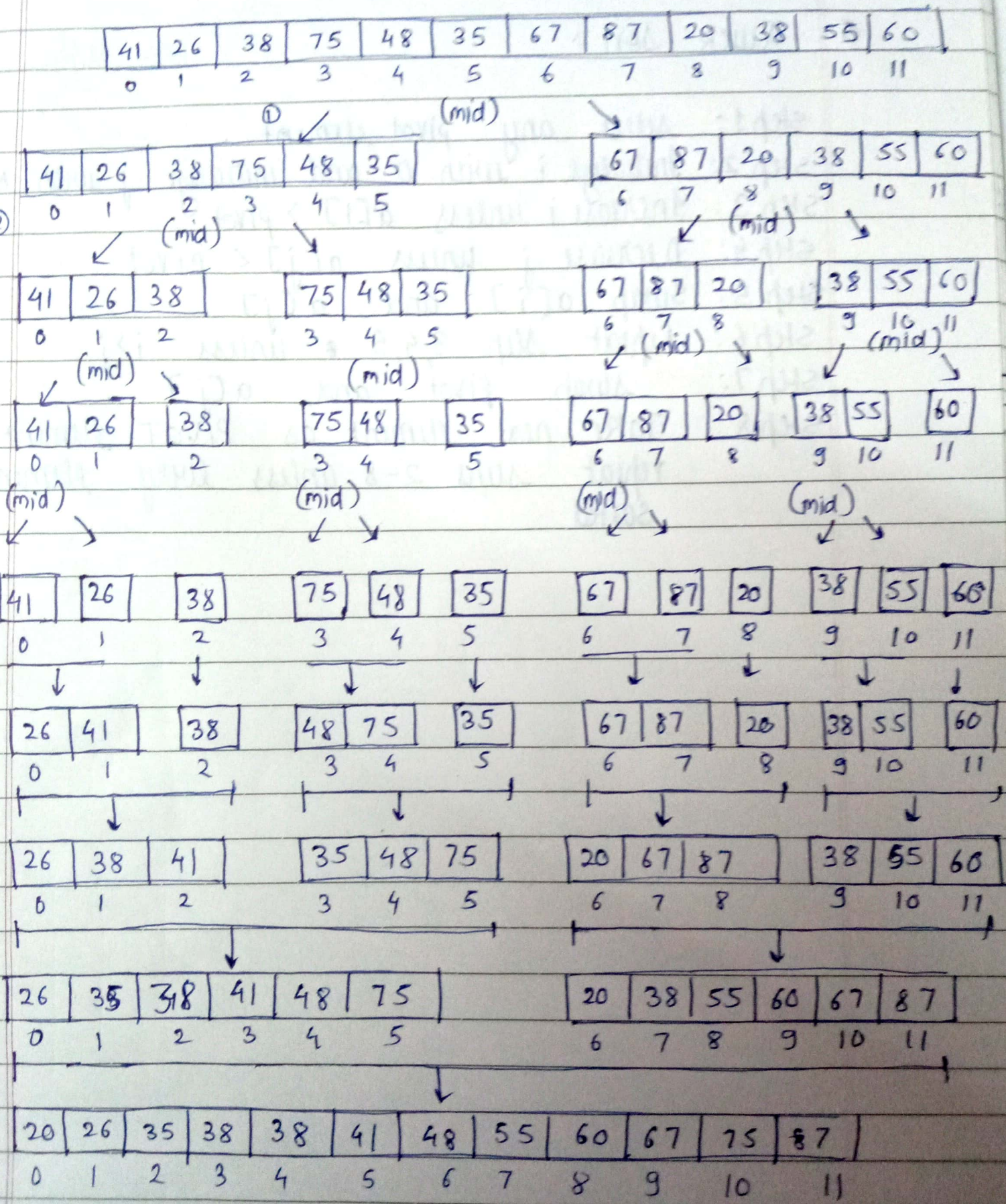
500	301	603	429	240	347	768	75	877	683	195

~~Pass IV~~



5) Merge Sort

```
void divide ( int start , int end , int arr[])
{
  int mid_ind = (start + end)/2;
  if (end != start)
  {
    divide (mid_ind+1 , end, arr) ;
    join_and_sort (mid_ind+1, end, arr) ;

    divide (start , mid_ind , arr);
    join_and_sort (start , mid_ind, arr);

  }
}

void ~~auto~~ join_and_start (int start , int end, int arr[])
 {
   int i, j;
   for( i=st; i<= end; i++)
   {
     temp = arr[i];
   for( j= i-1; j>=0 && arr[j] > temp; j--)
    {
      arr[j+1] = arr[j];

    }
   arr[j+1] = temp;

  }
}
```



6. Quick Sort:

Step 1: Select any pivot element.
Step 2: Initialize i with 0 and initialize j with MAX_SIZE-1
Step 3: Increase i unless a[i] > pivot.
Step 4: Decrease j unless a[j] < pivot.
Step 5: Swap a[i] and a[j]
Step 6: Repeat step 3,4,5 unless i>j.
Step 7: Swap pivot and a[j]
Step 8: Take next element as PIVOT element and repeat steps 2-8 unless every element gets sorted

Hashing and Indexing :

We are already familiar with the concepts of time complexity and space complexity.
We know that to search in data structures, we have two important searching techniques : Linear search and binary search.

Even in case of larger data structures these searching techniques are time consuming, then we have concept of Binary search tree as solution. But in binary search tree, the problem is unbalanced parent nodes (balance factor $(H_L - H_R)$).
Now the solution is AVL trees and we do rotations. Now in case of AVL trees, it is also time consuming for preparing AVL trees and do rotations and then to search.
All previous theories are useful for smaller to medium data structures.

Now in case of huge data structures and data sets, such as database table, all above theories will be less effective and time consuming. In such cases the most useful theory is of hashing and Indexing.

Hashing is the process of applying a formula / algorithm to obtain / decide suitable index / location to place / put that data. The result of hashing is index.

There are numbers of formulae and algorithms for hashing and indexing.

For eg: consider following list of elements and we will place it ~~index~~ in an array by using formula element % max_size

If we apply formula element % MAX_SIZE on below 10 elements then it will be

75, 38, 70, 41, 55, 44, 80, 27, 17, 65

75 % 10 = 5 → place element 75 at index 5
38 % 10 = 8 → place element 38 at index 8
70 % 10 = 0 → place element 70 at index 0
41 % 10 = 1 → place element 41 at index 1
55 % 10 = 5 → place element 55 at index 5

From above list, we can say that by using the formula we get a proper index number to store specific element and this formula will be inversely helpful to search the same element. From above example, we can also say that there can be multiple elements that results same indexing for formula.

For eg: In above list, we have 75, 55 and 65 that results 5 as indexing. The elements 70 and 80 results 0 as indexing. Now in such cases there should be a solution / algorithm to decide another indexing for such elements. This process is called as Probing.

Probing is an algorithm / theory to decide new indexing if result of hashing is same for multiple data. There are many different probing techniques. some of them are-

- Linear probing
- Quadratic probing
- Linked Probing.

1) Linear probing - In linear probing, if result of indexing for multiple elements is same then we place new element at index + 1 location.

2) Quadratic Probing - In quadratic probing, if result of indexing for multiple elements is same then place the new element at (index)^n.

3) Linked Probing - In Linked probing, we apply concept of linked list to link the elements / data with same indexing (having same index). This reduces space complexity.