SORTING TECHNIQUES

1. Bubble Sort
2. Insertion Sort — $O(n^2)$
3. Selection Sort

4. Heap Sort
5. Merge Sort — $O(n \log n)$
6. Quick Sort
7. Tree Sort

8. Shell Sort — $O(n^{3/2})$

Comparison based sorts

9. Count Sort
10. Bucket / Bin Sort — $O(n)$
11. Radix Sort

Index Based Sort
// Faster but memory consumption is high

There is no best sorting technique.

You have to chose the technique which matches your requirements

CRITERIA FOR ANALYSIS

1. Number of comparisons
2. Number of swaps
3. Adaptive // If already the array is sorted, it should make minimum comparisons
4. Stable
5. Extra Memory required

1. List sorted on basis of name

Name :	A	B	C	D	E	F	G
Marks :	5	8	6	4	6	7	10

Duplicate elements

2. List sorted on basis of marks

Name :	D	A	C	E	F	B	G
Marks :	4	5	6	6	7	8	10

Here also 'C' should come before 'E' as order must be preserved

This type of algorithms are useful in databases

If the sorting algorithm is preserving the order of duplicate elements in the sorted list then that algorithm is called 'Stable'.

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

Largest element is sorted in first pass

4 comp
4 swap

2nd Pass

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

3 comp
3 swap

3rd Pass

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

2 comp
2 swap

4th Pass

3 2
2 3
5 5
7 7
8 8

1 comp
1 swap

No of passes : 4
: $(n-1)$

→ Not actual numbers of swaps but maximum numbers of swaps

No of comparison : $1+2+3+4$
: $1+2+3+4 \ldots\ldots (n-1)$
: $\frac{n(n-1)}{2}$ $O(n^2)$

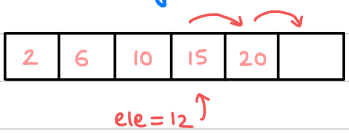
No of swaps : $1+2+3+4$
$1+2+3+4 \ldots\ldots (n-1)$
$\frac{n(n-1)}{2}$ $O(n^2)$

```
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;
    }
}
```

- Called as bubble sort as lighter or smaller elements come up same as bubbles
- Performing selected number of passes can give same number of largest elements as number of passes performed
- Bubble Sort is adaptive
- It is stable
- 'k' passes give 'k' number of largest element

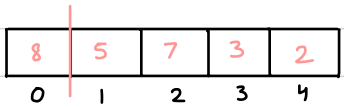
2. INSERTION SORT

Inserting element in a sorted array at a sorted position



// Start comparison from last element

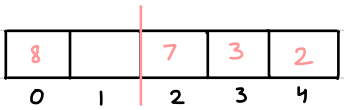
EXAMPLE : A



// Assume only first element to be sorted

n = 5

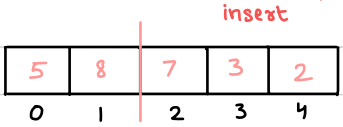
1st Pass



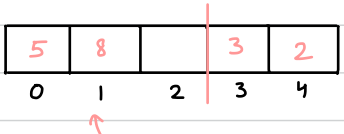
1 comp
1 swap

5

Take out one element and compare and insert

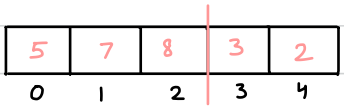


2nd Pass

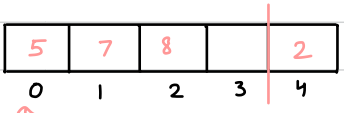


2 comp
2 swap

7

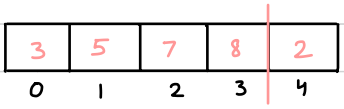


3rd Pass

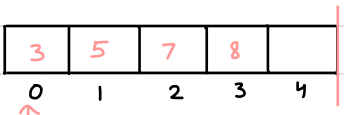


3 comp
3 swap

3

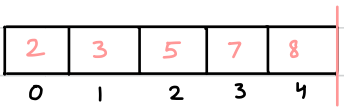


4th Pass



4 comp
4 swap

2



No of passes : (n-1)

No of comparison : $\frac{n(n-1)}{2}$ $O(n^2)$

No of swaps : $\frac{n(n-1)}{2}$ $O(n^2)$

- Insertion sort is adaptive as no swapping is done if array is sorted. Only 'n' comparisons are done which take minimum time
  It is adaptive by nature (no flag used)
- Insertion sort is stable
- Used in linked list.

```
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;
    }
}
```

3. SELECTION SORT

↳ Selecting a position in the array and finding the smallest element suitable for that.

A

8	6	3	2	5	4
0	1	2	3	4	5

n = 6

1st Pass	2nd Pass	3rd Pass	4th Pass	5th Pass	
0 8 ← i	0 2	0 2	0 2	0 2	0 2
1 6	1 6 ← i	1 3	1 3	1 3	1 3
2 3	2 3 ← k	2 6 ← i	2 4	2 4	2 4
3 2 ← k	3 8	3 8	3 8 ← i	3 5	3 5
4 5	4 5	4 5	4 5 ← k	4 8 ← i	4 6
5 4	5 4	5 4 ← k	5 6	5 6 ← k	5 8

No of comparison : $\frac{n(n-1)}{2}$ $O(n^2)$

No of swaps : $n-1$ $O(n)$

- k passes give k number of smallest elements
- Selection Sort is not adaptive
- Selection Sort is not stable
- Selection Sort performs minimum number of swaps

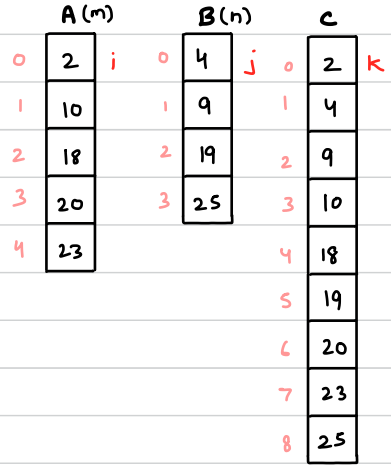
```
void SelectionSort ( int A[] , int n)
{
    int i, j, k;

    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]);
    }
}
```

5. MERGING

- Merging two lists in third array
- Merging two lists in single array
- Merging multiple list

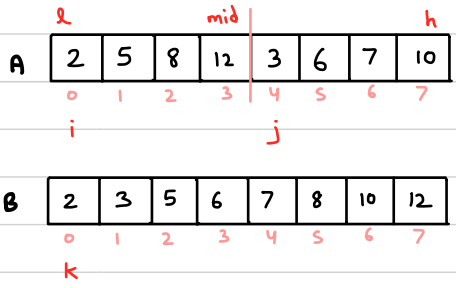
• MERGING TWO LISTS IN THIRD ARRAY

O(m+n)



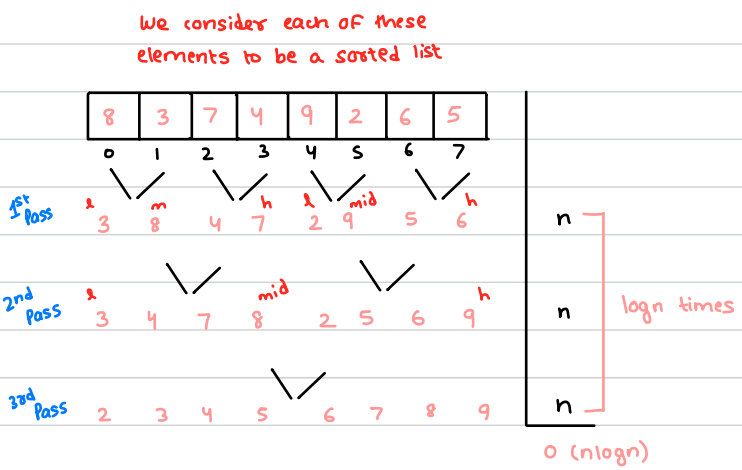

```
Void Merge (int A[], int B[], int m, int n)
{
    int i, j, k;
    i = j = k = 0;

    while(i<m && j<n)
    {
        if (A[i] < B[j])
            c[k++] = A[i++];
        else
            c[k++] = B[j++];
    }

    for (; i<m; i++)          // for remaining elements
        c[k++] = A[i];        in either of list

                              // Only one of these loops
    for (; j<n ; j++)         will work
        c[k++] = B[j];
}
```

• MERGING TWO LISTS IN SINGLE ARRAY




```
Void Merge (int A[], int l, int mid, int h)
{
    int i, j, k;
    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];
}
```

ITERATIVE MERGESORT

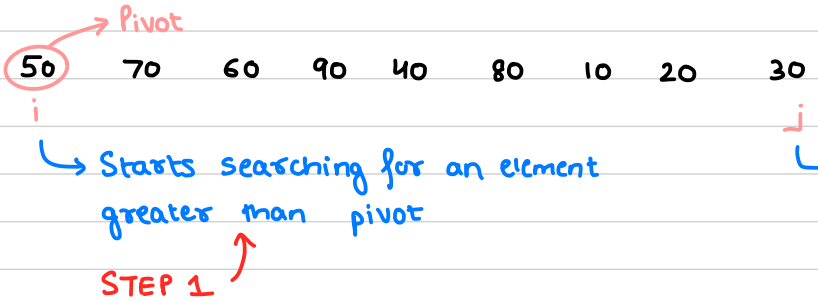



```
void IMergeSort(int A[], int n)
{
    int p, i, l, mid, h;

    for(p=2; p<=n; p=p*2)
    {
        for(i=0; i+p-1<n; i=i+p)
        {
            l = i;
            h = i+p-1;
            mid = ⌊(l+h)/2⌋;

            Merge(A, l, mid, h);
        }
    }

    if(p/2 < n)
        Merge(A, 0, p/2, n-1);
}        // If numbers of elements are odd
```

RECURSIVE MERGE SORT O(nlogn)

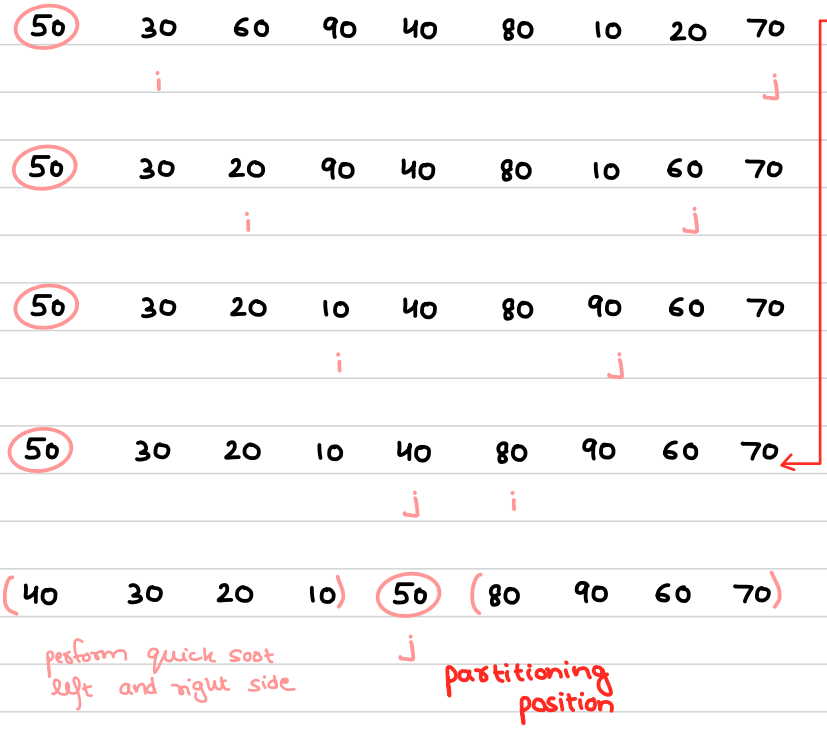
```
void RMergeSort(int A[], int l, int h)
{
    if(l<h)
    {
        mid = ⌊(l+h)/2⌋;
        RMergeSort(A, l, mid);
        RMergeSort(A, mid+1, h);
        Merge(A, l, mid, h);
    }
}
```

6. QUICK SORT → Selecting a element and then finding its position



When larger and smaller elements found, exchange them ← STEP 3

PARTITIONING PROCEDURE



If i becomes greater than j, we have checked entire list Now j^{th} index element and pivot should be swapped ← STEP 4

```
int Partition(int A[], int l, int n)
{
    int Pivot = A[l];
    int i=l, j=h;

    do
    {
        do {i++} while (A[i] ≤ pivot);
        do {j--} while (A[i] > pivot);

        if (i<j)
            swap(A[i], A[j]);
    } while (i<j);
    swap(A[l], A[j]);
    return j;
}
```

Best Case : If partitioning is in middle
$O(n \log n)$

Worst Case : If partioning is on any end
$O(n^2)$

Avg case : $O(n \log n)$

- best case : sorted list
  (first make middle element as pivot)
- worst case : partitioning on any end
  $O(n^2)$

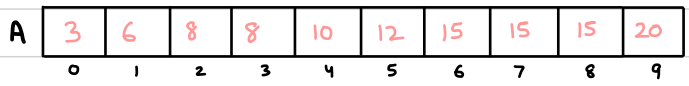
```
void QuickSort(int A[], int l, int h)   // l: lowest index (0), h: highest index (8) in this example
{
    int j;
    if (l<h)
    {
        j = Partition(A, l, h);
        QuickSort(A, l, j);
        QuickSort(A, l+1, h);
    }
}
```

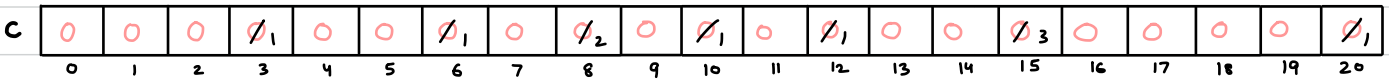
8. SHELL SORT

Used for large arrays

```
void ShellSort(int A[],int n)
{
    int gap,i,j,temp;

    for(gap=n/2;gap>=1;gap/=2)
    {
        for(i=gap;i<n;i++)
        {
            temp=A[i];
            j=i-gap;

            while(j>=0 && A[j]>temp)
            {
                A[j+gap]=A[j];
                j=j-gap;
            }

            A[j+gap]=temp;
        }
    }
}

void main()
{
    int A[]={11,13,7,12,16,9,24,5,10,3},n=10;

    ShellSort(A,n);
}
```

9. COUNT SORT



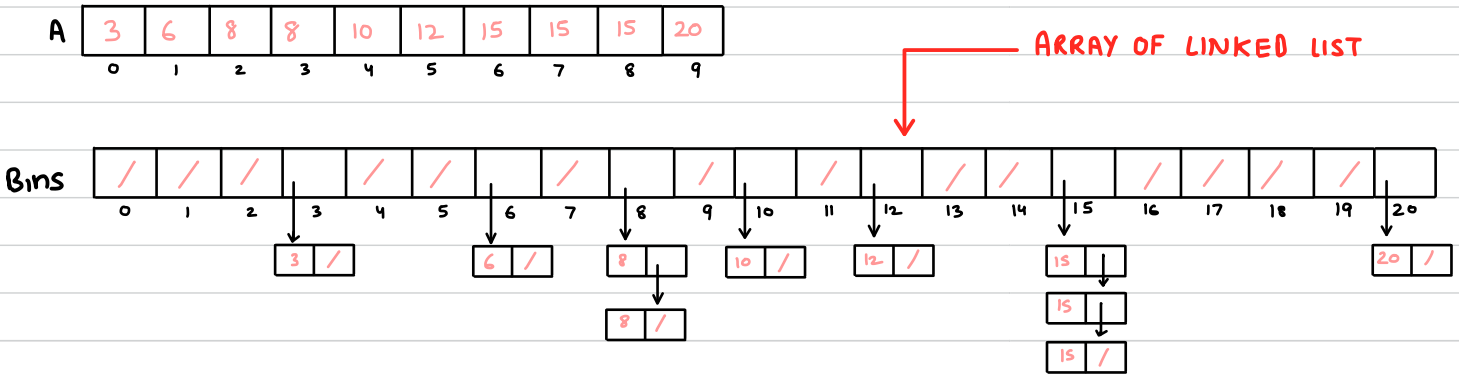



```
Void CountSort(int A[],int n)
{
    int max,i,j;
    int *c;

    max = findMax(A,n);
    C = new int[max+1];          C = (int*)malloc(sizeof(int)*(max+1));

    for(i=0;i<max+1;i++)         n
        c[i]=0;

    for(i=0;i<n;i++)             n
        C[A[i]]++;

    i=0,j=0;

    while(i<max+1)               n
    {
        if(c[i]>0)
        {
            A[j++]=i;
            c[i]--j
        }

        else
            i++;
    }
}
```

Fastest sorting method but high memory consumption

O(n)

10. BUCKET / BIN SORT

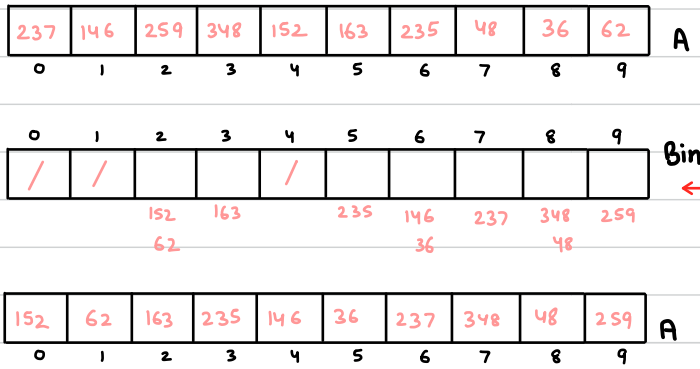



```
void BinSort(int A[], int n)
{
    int max, i, j;
    max = findMax(A,n);
    Bins = new Node*[(max+1)];

    for(i=0; i<max+1; i++)
        Bins[i] = NULL;

    for(i=0; i<n; i++)
        Insert(Bins[A[i]], A[i]);      // Insert at end of linked list

    i=0, j=0;

    while(i<max+1)
    {
        while(Bins[i]!=NULL)
        {
            A[j++] = Delete(Bin[i]);    O(n)
        }
        i++;
    }
}
```

11. RADIX SORT

1st Pass

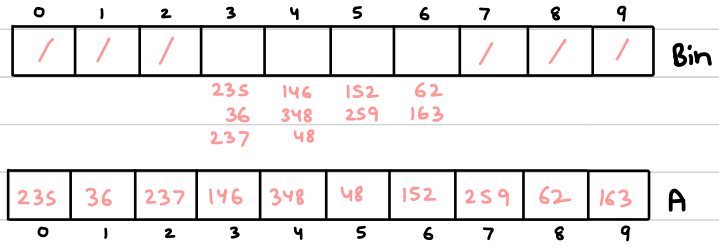
Elements arranged on basis of one's digit



// Take out the elements in FIFO way and store them

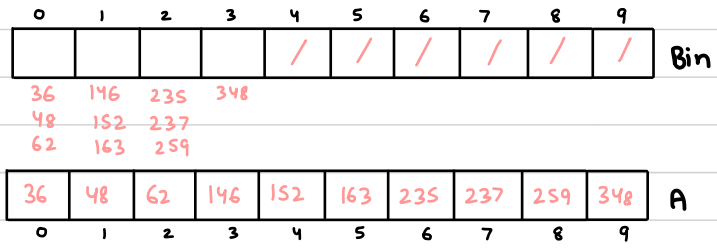
2nd Pass

two's digit



3rd Pass

three's digit



two digit numbers's third digit considered as 0

1. [A[i] / 1] % 10
2. [A[i] / 10] % 10
3. [A[i] / 100] % 10

Decimal numbers require 10 sized bin (0-9)
Binary numbers require 2 sized bin (0,1)
Octal numbers require 8 sized bin (0-7)

O(n) and minimum storage required
Storage depends on maximum number of digits of largest element