Linear Search

```
Linear_Search(a[1...n], n, elem) {
    int i, flag=0;
    for(i=1 to n) {
        if(a[i] == elem) {
            display("Elem Found at index %d", i);
            flag=1;
        }
    }
    if(flag==0){
        display("Not Found");
    }
}
```

Binary Search :-

```
Binary_Search(a[1...n], n, elem){
    int FI=1;
    int LI=n;
    int mid, flag=0;
    while(FI!=LI && LI>FI){
        mid=(FI+LI)/2
        if(a[mid]==elem){
            display("Found at index %d", mid);
            flag=1;
            break;
        }
        else if(a[mid]<elem){
            FI=mid+1;
        }
        else{
            LI=mid-1;
        }
    }
    if(flag==0){
        if(a[LI]==elem){
            display("Found at index %d", LI)
        }
        else{
            display("Not Found");
        }
    }
}
```

Sorting :-

→ Insertion_Sort(n){
    int i, j, b, *a;

```
a=(int *) malloc (sizeof (int) * n);
display ("Enter Elements");
for (i = 1 to n) {
        scanf ("%d", &b);
        if (i == 1) {
            a[i] = b
        } else {
            j = i - 1;
            while (a[j] > b and j >= 1) {
                a[j+1] = a[j];
                j--;
            }
            a[j+1] = b;
        }
}
```

```
else {
    for (j = i - 1; a[j] > b && j >= 1; j--) {
        a[j+1] = a[j];
    }
}
```

→ Bubble_Sort (a[1--n], n) {

```
for (i = 1 to n) {
    for (j = 1 to n - i) {
        if (a[j] > a[j+1]) {
            swap (a[j], a[j+1]);
        }
    }
}
```

}

→ Merge_Sort (a[1--n], Fl, Ll) {

```
if (Fl < Ll) {
    int mid = (Fl + Ll) / 2;
    Merge_Sort (a, Fl, mid);
    Merge_Sort (a, mid+1, Ll);
    Merge_Part (a, Fl, Ll, mid);
}
```

}

Merge_Part (a[1-- n], Fl, Ll, mid) {

```
int j = mid + 1;
int k = Fl - 1; int i;
int *b;
b = (int *) malloc ((Ll+1) * sizeof (int));
```

```
while(i<=mid && j<=ll){
    if(a[i]>a[j]){
        k++;
        b[k]=a[j];
        j++;
    }else{
        k++;
        b[k]=a[i];
        i++;
    }
}
while(j<=ll){
    k++;
    b[k]=a[j];
    j++;
}
while(i<=mid){
    k++;
    b[k]=a[i];
    i++;
}
for(i=fl;i<ll;i++){
    a[i]=b[i];
}
}
```

→ Quick_Sort(a[1--n], fl, ll){

```
int q;
if(fl<ll){
    q=Quick_Part(a,fl,ll);
    Quick_Sort(a,fl,q-1);
    Quick_Sort(a,q+1,ll);
}
}
```

```
Quick_Part(a[1--n], fl, ll){
int i,j,elem,temp;
elem=a[ll];
i=fl;
j=fl-1;
while(i<ll){
    if(a[i]>p){
```

```
while (i<=l){
    if(a[i]>p){
        i++;
    }
    else{
        j++;
        swap(a[i], a[j]);
        i++;
    }
}
swap(a[i], a[j+1]);
return j+1;
}
```

→ Count_Sort

```
Count_Sort(a[0---n]){
int m = max of a;
int *c; int *b;
c = (int *) malloc(m*sizeof(int));   b = (int *) malloc(n*sizeof(int));
for(i=0 to n){
    c[a[i]]++;                  → Frequency
}
for(i=2 to m){
    c[i] += c[i-1];             → Cumulative Frequency
}
for(i=n-1 to 0){
    c[a[i]]--;
    b[c[a[i]]]=a[i];
}
for(i=1 to n){
    a[i]=b[i];
    display(a[i]);
}
}
```

Radix_Sort

```
Radix_Sort(a[0---n]){
    int m = max of a;
    int p=1;
    for(p=1; max/p>0; p*=10){
        count_sort(a,p);
    }
```

```
}
Count_Sort(a[0--n], P){
    int i, *b, *c;
    b = (int *) calloc(n, sizeof(int));
    c = (int *) calloc(10, sizeof(int));
    for(i=0; i<n; i++) {
        c[[(a[i]/p)%10)]]++;
    }
    for(i=1; i<10; i++){
        c[i] += c[i-1];
    }
    for(i=n-1; i>=0; i--){
        c[[a[i]/p)%10]]--;
        b[c[[(a[i]/p)%10]]] = a[i];
    }
    for(i=0 to n) {
        a[i] = b[i];
        display(a[i]);
    }
}
```

→ Selection_Sort

```
Selection_Sort(a[1--n], n) {
    int i, j, min_index;
    for(i=1 to n) {
        min_index = i;
        for(j=i+1 to n){
            if(a[j]<a[min_index]){
                min_index = j;
            }
        }
        swap(a[i], a[min_index]);
    }
}
```

★ Bucket Sort Nahi Hai Notes Me
