Criteria for Analysis of Sorting Algorithms

1. Time Complexity
2. Space Complexity
3. Stability (for same values give higher preference to lower index)
4. Internal SA(All the data is loaded in memory)
5. External SA(All the data is NOT loaded in memory)
6. Adaptive(Already sorted takes less time)
7. Recursive/Non-Recursive

Different Types of Sorting Algorithms

1. Bubble Sort

A. Method
let arr=[7,11,9,2,17,4]
1st pass: 7 11 9 2 17 4 -> 0,1 index
    7 9 11 2 17 4 -> 1,2 index
    7 9 2 11 17 4 -> 2,3 index
    7 9 2 11 17 4 -> 3,4 index
    7 9 2 11 4 17 -> 4,5 index

2nd pass: 7 9 2 11 4 17 -> 0,1 index
    7 2 9 11 4 17 -> 1,2 index
    7 2 9 11 4 17 -> 2,3 index
    7 2 9 4 11 17 -> 3,4 index

3rd pass: 2 7 9 4 11 17 -> 0,1 index
    2 7 9 4 11 17 -> 1,2 index
    2 7 4 9 11 17 -> 2,3 index

4th pass: 2 7 4 9 11 17 -> 0,1 index
    2 4 7 9 11 17 -> 1,2 index

5th pass: 2 4 7 9 11 17 -> 0,1 index Final sorted array

B. Analysis
1. Time Complexity: O(n^2) if not sorted/ O(n) if sorted
2. Stable Algorithm
3. Not adaptive by nature but can be made adaptive

C. Code

```c
#include <stdio.h>

void display(int* arr,int n){
    for(int i=0;i<n;i++){
        printf("%d\t",arr[i]);
    }
}

void swap(int* a,int* b){
    int temp;
    temp=*a;
    *a=*b;
    *b=temp;
}

void bubbleSort(int* a,int n){
    int isSorted;
    for(int i=0;i<n-1;i++){
        isSorted=1;
        for(int j=0;j<n-i-1;j++){
            if(a[j]>a[j+1]){
                swap(&a[j],&a[j+1]);
                isSorted=0;
            }
        }
        if(isSorted==1){
            return;
        }
    }
}

int main(){
    int n;
    printf("Enter size: ");
    scanf("%d",&n);
    int a[n];
    printf("Enter array elements: ");
    for(int i=0;i<n;i++){
        scanf("%d",&a[i]);
    }
    printf("Before Sort:-\n");
    display(a,n);
```

```
    bubbleSort(a,n);
    printf("\nAfter Sort:-\n");
    display(a,n);
    return 0;
}
```

2. Insertion Sort

A. Method

let arr=[7,12,3,4,1]

S<-|->NS

Step 1: 7 **|** 12 3 4 1

Now insert 12 in the sorted array such that the array till 12 is sorted

Here 1 possible comparison

S<-|->NS

Step 2: 7 12 **|** 3 4 1

Now insert 3 in such a way that array is sorted till 3

Here 2 possible comparison

S<-|->NS

Step 3: 3 7 12 **|** 4 1

Now repeat the same for insertion of 4

Here 3 possible comparison

S<-|->NS

Step 4: 3 4 7 12 **|** 1

Now finally do it for insertion of 1

Here 4 possible comparison

Final answer: 1 3 4 7 12 this is the sorted array

Here 5 possible comparison

B. Analysis

1. Time Complexity: O(n^2) for unsorted/ O(n) for sorted
2. Stable Algorithm
3. Adaptive by nature

C. Code

```c
#include <stdio.h>

void display(int* arr,int n){
    for(int i=0;i<n;i++){
        printf("%d\t",arr[i]);
    }
}

void swap(int* a,int* b){
    int temp;
    temp=*a;
    *a=*b;
    *b=temp;
}

void insertionSort(int* a,int n){
    //loop for each step
    for(int i=1;i<=n-1;i++){
        //loop for each step
        int key=a[i];
        int j=i-1;
        while((a[j]>key)&&(j>=0)){
            swap(&a[j+1],&a[j]);
            j--;
        }
    }
}

int main(){
    int n;
    printf("Enter size: ");
    scanf("%d",&n);
    int a[n];
    printf("Enter array elements: ");
    for(int i=0;i<n;i++){
        scanf("%d",&a[i]);
    }
    printf("Before Sort:-\n");
    display(a,n);
    insertionSort(a,n);
    printf("\nAfter Sort:-\n");
    display(a,n);
```

```
    return 0;
}
```

3. Selection Sort

A. Method

let arr={8,0,7,1,3}

here we assume the first number as the smallest number, then we traverse to find if smaller number exists or not. After that we sort the array till index 0 in 1st pass. We repeat the same process till we have sorted the 4th element in an array of length 5.

```
           |->NS
1st pass: 0 | 8 7 1 3
             |->NS
2nd pass: 0 1 | 7 8 3
               |->NS
3rd pass: 0 1 3 | 8 7
4th pass: 0 1 3 7 8
```

B. Analysis

1. Time Complexity: O(n^2)
2. Not Stable Algorithm
3. Not Adaptive Algorithm

C. Code

```
#include <stdio.h>

void display(int* arr,int n){
    for(int i=0;i<n;i++){
        printf("%d\t",arr[i]);
    }
}

void swap(int* a,int* b){
    int temp;
    temp=*a;
    *a=*b;
    *b=temp;
}

void selectionSort(int* arr, int n){
    int index;
    for(int i=0;i<n-1;i++){
        index=i;
        for(int j=i+1;j<n;j++){
            if(arr[j]<arr[index]){
```

```
                    index=j;
               }
          }
          swap(&arr[i],&arr[index]);
     }
}

int main(){
     int n;
     printf("Enter size: ");
     scanf("%d",&n);
     int a[n];
     printf("Enter array elements: ");
     for(int i=0;i<n;i++){
          scanf("%d",&a[i]);
     }
     printf("Before Sort:-\n");
     display(a,n);
     selectionSort(a,n);
     printf("\nAfter Sort:-\n");
     display(a,n);
     return 0;
}
```

4. Quick Sort

A. Method
let arr={2,4,3,9,1,4,8,7,5,6}
Pivot: first element or arr[0]
Partioning:-
1. i=low, j=high, pivot=low
2. i++ until element>=pivot is found
3. j-- until element<=pivot is found
4. Swap arr[i], arr[j] and repeat 2,3 until j<=i
5. Swap pivot and arr[j]

B. Analysis
1. Time Complexity
Worst case (already sorted): O(n^2)
Best Case: O(nlog(n))
2. Not a stable Algorithm
3. Is a inplace Algorithm

C. Code

```
#include <stdio.h>

void display(int* arr,int n){
    for(int i=0;i<n;i++){
        printf("%d\t",arr[i]);
    }
}

void swap(int* a,int* b){
    int temp;
    temp=*a;
    *a=*b;
    *b=temp;
}

int partition(int* a,int low,int high){
    int pivot= a[low];
    int i=low+1;
    int j=high;
    do
    {
        while(a[i]<=pivot){
            i++;
        }
        while(a[j]>=pivot){
            j--;
        }
        if(i<j){
            swap(&a[i],&a[j]);
        }
    } while (i<=j);
    swap(&a[low],&a[j]);
    return j;
}

void quickSort(int* a,int low, int high){
    int partitionIndex; // Index of pivot after partition
    if(low<high){
        partitionIndex=partition(a,low,high);
        quickSort(a,low,partitionIndex-1); // quick sort for left
subarray
```

```
        quickSort(a,partitionIndex+1,high); // quick sort for right
subarray
    }
}

int main(){
    int n;
    printf("Enter size: ");
    scanf("%d",&n);
    int a[n];
    printf("Enter array elements: ");
    for(int i=0;i<n;i++){
        scanf("%d",&a[i]);
    }
    printf("Before Sort:-\n");
    display(a,n);
    quickSort(a,0,n-1);
    printf("\nAfter Sort:-\n");
    display(a,n);
    return 0;
}
```

5. Merge Sort

A. Method

1. The idea is to break the array in two equal parts i.e around mid and the merge the two arrays in a new one.
2. For the sorting part we use recursive function call while for merging we use if-else and loops. Basically we compare each element of first array with the second array and then add them in the new array accordingly

B. Code

```
#include <stdio.h>

void display(int* arr,int n){
    for(int i=0;i<n;i++){
        printf("%d\t",arr[i]);
    }
}

void merge(int* a,int mid,int low,int high){
    int i,j,k,b[100];
    i=low;
    j=mid+1;
```

```c
    k=low;
    while(i<=mid && j<=high){
        if(a[i]<a[j]){
            b[k]=a[i];
            i++;k++;
        }else{
            b[k]=a[j];
            j++;k++;
        }
    }
    while(i<=mid){
        b[k]=a[i];
        i++;k++;
    }
    while(j<=high){
        b[k]=a[j];
        j++;k++;
    }
    for (int i=low;i<=high;i++){
        a[i]=b[i];
    }

}

void mergeSort(int* a,int low,int high){
    int mid;
    if(low<high){
        mid=(low+high)/2;
        mergeSort(a,low,mid);
        mergeSort(a,mid+1,high);
        merge(a,mid,low,high);
    }
}

int main(){
    int n;
    printf("Enter size: ");
    scanf("%d",&n);
    int a[n];
    printf("Enter array elements: ");
    for(int i=0;i<n;i++){
        scanf("%d",&a[i]);
    }
```

```
    printf("Before Sort:-\n");
    display(a,n);
    mergeSort(a,0,n-1);
    printf("\nAfter Sort:-\n");
    display(a,n);
    return 0;
}
```

6. Count Sort

A. Method

1. Find the max value element in given array
2. Create an array of size max value viz count array. Then for each value in given array increment 1 in the count array at that where value==index
3. Now traverse through count array such that if element is 0 the counter++, else copy index back in original array and decrement the value in count array

B. Analysis

1. Extra Space
2. Time Complexity: O(m+n)
3. One of the fastest algorithms for sorting

C. Code

```
#include <stdio.h>
#include <stdlib.h>

void display(int* arr,int n){
    for(int i=0;i<n;i++){
        printf("%d\t",arr[i]);
    }
}

void countSort(int* a,int n){
    int max=0,k=0,j=0;
    for(int i=0;i<n;i++){
        if(max<a[i]){
            max=a[i];
        }
    }
    int* count=(int*)malloc((max+1)*sizeof(int));
    for(int i=0;i<=max;i++){
        count[i]=0;
    }
    for(int i=0;i<n;i++){
        count[a[i]]++;
```

```
    }
    while(k<=max){
        if(count[k]>0){
            a[j]=k;
            count[k]--;
            j++;
        }else{
            k++;
        }
    }
}

int main(){
    int n;
    printf("Enter size: ");
    scanf("%d",&n);
    int a[n];
    printf("Enter array elements: ");
    for(int i=0;i<n;i++){
        scanf("%d",&a[i]);
    }
    printf("Before Sort:-\n");
    display(a,n);
    countSort(a,n);
    printf("\nAfter Sort:-\n");
    display(a,n);
    return 0;
}
```