**Practical 3**

**Aim: - Divide and Conquer Strategy (Implement & Perform Analysis)**

**3.1 Merge Sort**

**3.2 Quick Sort**

**3.1 Merge Sort**

**Theory:-**

* Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being Ο (n log n), it is one of the most efficient algorithm.
* Most implementations produce a stable sort, which means that the implementation preserves the input order of equal elements in the sorted output.
* Merge sort is an efficient, general-purpose, [comparison-based](https://en.wikipedia.org/wiki/Comparison_sort) [sorting algorithm](https://en.wikipedia.org/wiki/Sorting_algorithm)
* Merge sort first divides the array into equal halves and then combines them in a sorted manner.

**Algorithm: -**

Simple Merge :

SIMPLE\_MERGE\_SORT (K, FIRST, SECOND, THIRD)

1. [INITIALIZE]

2. [FIND SMALLEST ELEMENT]

   I ← FIRST

J ← SECOND

L ← 0

Repeat While I&lt;SECOND and J≤THIRD

If K [I] ≤ K [J]

Then L ← L+1

Else L ← L+1

TEMP [L] ← K [I]

I ← I+1

TEMP [L] ← K [I]

J ← J+1

3. [COPY THE REMAINING ELEMENTS]

If I≥SECOND

Then Repeat while J≤THIRD

L ← L+1

TEMP [L] ← K [J]

J ← J+1

Else Repeat while I&lt; SECOND

L ← L+1

TEMP [L] ← K [I]

I ← I+1

4. [COPY ELEMENTS]

Repeat for I=1, 2…L

K [FIRST-1+I] ← TEMP [I]

5. [FINISHED] Return

**Program: -**

**Code: -**

#include<stdio.h>

void divide(int \*a, int min, int max, int \*tmp);

void merge(int \*a, int min, int mid, int max, int \*tmp);

int count=0,comp=0;

int x=1;

void main()

{

int \*a, i, j, n,\*tmp;

printf("Enter the size of array : ");

scanf("%d", &n);

a = (int \*) malloc(sizeof(int)\*n);

tmp = (int \*) malloc(sizeof(int)\*n);

for(i=0;i<n;i++)

{

printf("Enter element %d : ",i+1);

scanf("%d", &a[i]);

}

count++;

divide(a,0,n-1,tmp);

printf("\n\nSorted array is : \n");

for(i=0;i<n;i++)

{

printf(" %d", tmp[i]);

}

printf("\n\nNo. of Counts: %d", count);

printf("\nNo. of Comparisons: %d" ,comp);

}

void divide(int \*a, int min, int max, int \*tmp)

{

int mid;

if(min!=max)

{

mid=((min + max)/2);

divide(a, min, mid, tmp);

divide(a, mid+1,max,tmp);

merge(a, min, mid, max, tmp);

count+=4;

}

count++;

}

void merge(int \*a, int min, int mid, int max, int \*tmp)

{

int i=min, j=mid+1, k=min;

count++;

while(i<=mid && j<=max)

{

count++; comp++;

if(a[i]<a[j]) { tmp[k]=a[i]; i++; }

else { tmp[k]=a[j]; j++; }

k++;

count+=4;

}

count++;

while(j<=max)

{

tmp[k]=a[j]; k++; j++; count+=4; }

count++;

while(i<=mid)

{ tmp[k]=a[i]; k++; i++; count+=4; }

printf("\n AFTER PASS %d : ",x); x++;

for(i=min; i<=max; i++){

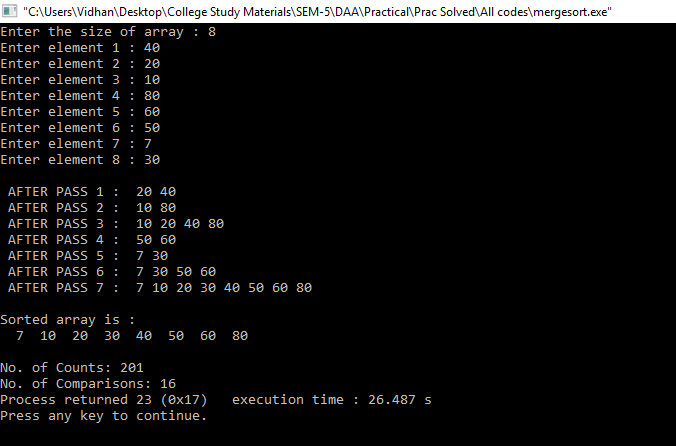
count++;

a[i]=tmp[i];

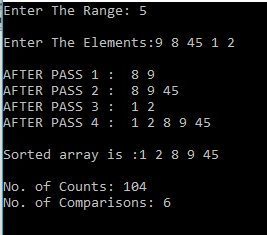
printf(" %d",a[i]);

}

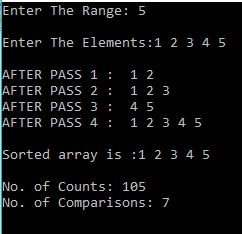
**Output: -**



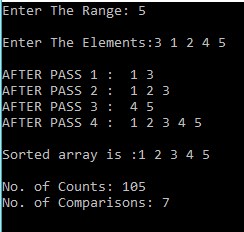
* Best case :



* Average case :



* Worst case :

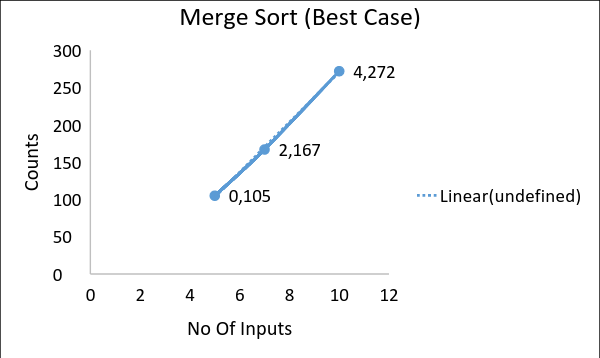


**Graph:**

Best case:

**Time complexity: O (n log (n))**

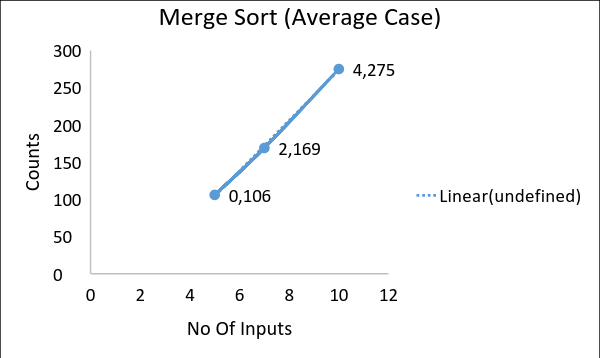
|  |  |  |  |
| --- | --- | --- | --- |
| Merge Sort (Best case) | | | |
| Sr No. | No Of Inputs | Count | Comparisons |
| 1 | 5 | 105 | 7 |
| 2 | 7 | 167 | 11 |
| 3 | 10 | 272 | 19 |



Average case:

**Time complexity: O (n log (n))**

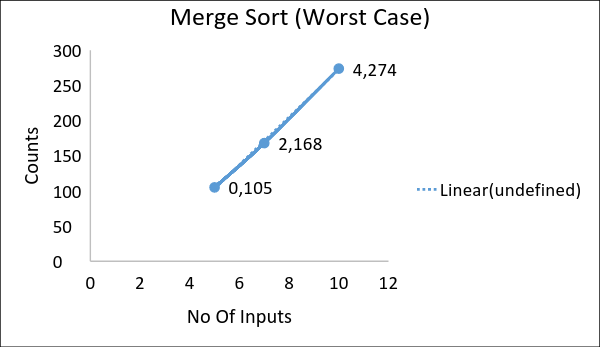
|  |  |  |  |
| --- | --- | --- | --- |
| Merge Sort(Average case) | | | |
| Sr No. | No Of Inputs | Count | Comparisons |
| 1 | 5 | 106 | 8 |
| 2 | 7 | 169 | 13 |
| 3 | 10 | 275 | 22 |



Worst case:

**Time complexity: O (n log (n))**

|  |  |  |  |
| --- | --- | --- | --- |
| Merge Sort (Worst case) | | | |
| Sr No. | No Of Inputs | Count | Comparisons |
| 1 | 5 | 105 | 7 |
| 2 | 7 | 168 | 12 |
| 3 | 10 | 274 | 21 |



**3.2 Quick Sort**

**Theory:**

* Quicksort (sometimes called partition-exchange sort) is an [efficient](https://en.wikipedia.org/wiki/Algorithm_efficiency) [sorting algorithm](https://en.wikipedia.org/wiki/Sorting_algorithm), serving as a systematic method for placing the elements of an [array](https://en.wikipedia.org/wiki/Array_data_structure) in order.
* Quicksort is a [comparison sort](https://en.wikipedia.org/wiki/Comparison_sort), meaning that it can sort items of any type for which a "less-than" relation (formally, a [total order](https://en.wikipedia.org/wiki/Total_order)) is defined.
* In efficient implementations it is not a [stable sort](https://en.wikipedia.org/wiki/Stable_sort), meaning that the relative order of equal sort items is not preserved.
* Quicksort can operate [in-place](https://en.wikipedia.org/wiki/In-place_algorithm) on an array, requiring small additional amounts of [memory](https://en.wikipedia.org/wiki/Main_memory) to perform the sorting.

**Algorithm:**

QUICK\_SORT (K, LB, UB)

1. [INITIALIZE]

    FLAG ← true

2. [PERFORM SORT]

   If LB&lt;UB

   Then I ← LB

   J ← UB+1

   KEY ← K[LB]

   Repeat while FLAG

   I ← I+1

   Repeat while K[I]&lt;KEY

   I ← I+1

   J ← J-1

   Repeat while K[J]&gt;KEY

   J ← J-1

   If I&lt;J

   Then K[I] ↔ K[J]

   Else FLAG ← false

   K[LB] ↔ K[J]

   Call QUICK\_SORT (K, LB, J-1)

   Call QUICK\_SORT (K, J+1, UB)

3. [FINISHED]

   Return

**Program:**

**Code:**

#include<stdio.h>

int count=0, swap=0, comp=0;

void main()

{

   int \*a, i, j, n, start ,end;

   printf("Enter The Range:");

   scanf("%d",&n);

   a = (int \*) malloc(sizeof(int)\*n);

   printf("\nEnter The Elements:");

   for(i=0;i<n;i++)

   {

       scanf("%d", (a + i));

   }

   start=0;

   end=n-1;

   printf("\n Sublists: \n");

   quicksort(a, start, end);

   printf("\nNo. of Counts: %d", count);

   printf("\nNo. of Swaps: %d", swap);

   printf("\nNo. of Comparisons: %d", comp);

   printf("\n\nSorted Array:");

   for(i=0;i<n;i++)

   {

       printf(" %d",\*(a + i));

   }

}

void quicksort(int \*a, int start, int end)

{

   int flag=0, i, j, pivot, temp, m;

   pivot = \*(a + start);

   i=start+1;

   j=end;

   count+=5;

   if(start<end)

   {

   count++;

   while(flag==0)

   {

   count++;

   while(pivot>\*(a + i)) { i++;count++; }

   count++;

   while(pivot<\*(a + j)) { j--;count++; }

   count++;

   comp++;

   if(i<j)

   {

       temp=\*(a + i);

       \*(a + i)=\*(a + j);

       \*(a + j)=temp;

       count+=3;

       swap++;

   }

                else { flag=1;count+=2; }

                }

   swap++;

   temp=\*(a + start);

   \*(a + start)=\*(a + j);

   \*(a + j)=temp;

   count+=3;

   printf("Pivot: %d\n", pivot);

   printf("List 1: [ ");

   for(m=start; m<=j-1; m++) { printf("%d ",\*(a + m)); }

   printf("]\n");

   printf("List 2: [ ");

   for(m=j+1; m<=end; m++) { printf("%d ",\*(a + m)); }

   printf("]\n\n");

   quicksort(a,start,j-1);

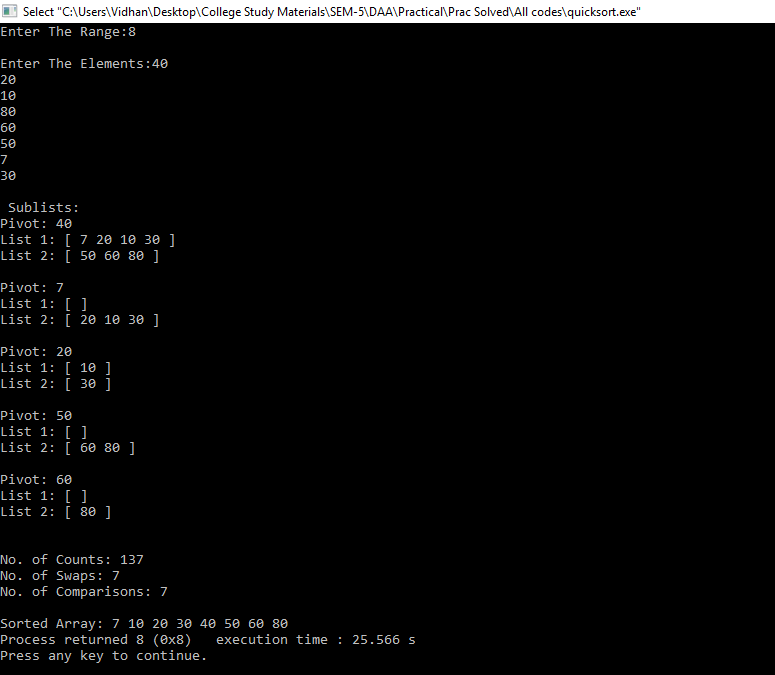
   quicksort(a,j+1,end);

   count+=2;

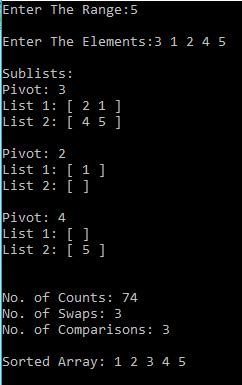
   }

}

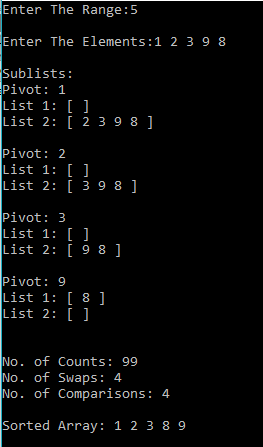
**Output:**



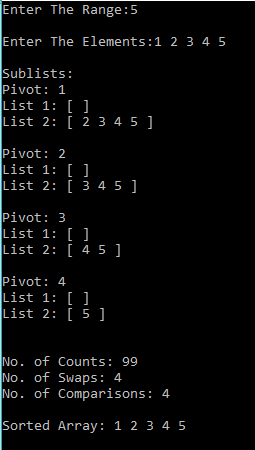
Best case:



Average case:



Worst case:

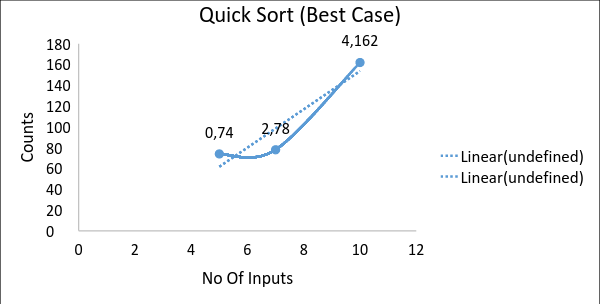


**Graph:**

Best case:

**Time complexity: O (n log (n))**

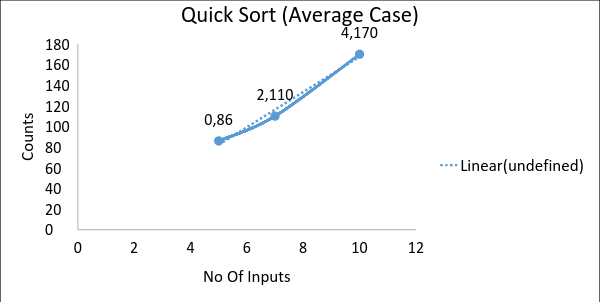
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Quick Sort (Best case) | | | | |
| Sr No. | No Of Inputs | Count | Swap | Comparisons |
| 1 | 5 | 74 | 3 | 3 |
| 2 | 7 | 78 | 3 | 3 |
| 3 | 10 | 162 | 7 | 7 |



Average case:

**Time complexity: O (n log (n))**

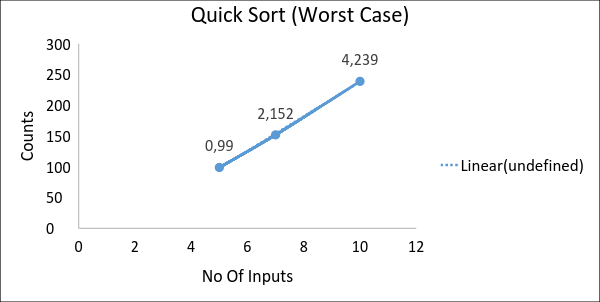
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Quick Sort (Average case) | | | | |
| Sr No. | No Of Inputs | Count | Swap | Comparisons |
| 1 | 5 | 86 | 5 | 5 |
| 2 | 7 | 110 | 5 | 5 |
| 3 | 10 | 170 | 9 | 9 |



Worst case:

**Time complexity: O (n^2)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Quick Sort (Worst case) | | | | |
| Sr No. | No Of Inputs | Count | Swap | Comparisons |
| 1 | 5 | 99 | 4 | 4 |
| 2 | 7 | 152 | 6 | 6 |
| 3 | 10 | 239 | 9 | 9 |



**Comparison Table:**

|  |  |  |  |
| --- | --- | --- | --- |
| Sr. No. | Name | Theatrical | Practical |
| 1 | Merge Sort – Best case | O(n log(n)) | O(n log(n)) |
| 2 | Merge Sort – Worst case | O(n log(n)) | O(n log(n)) |
| 3 | Merge Sort – Average case | O(n log(n)) | O(n log(n)) |
| 4 | Quick Sort – Best case | O(n log(n)) | O(n log(n)) |
| 5 | Quick Sort – Worst case | O(n^2) | O(n^2) |
| 6 | Quick Sort – Average case | O(n log(n)) | O(n log(n)) |

**Conclusion: -**

Merge sort is the most optimal sorting algorithm with the time complexity of O (n log (n)). Merge sort is more preferable when we use linked-list data structure. Merge sort is a stable sorting algorithm and therefore it has many advantages in solving various problems which are not effectively solved by using the quick sort.

Quick sort is also an optimal sorting algorithm with the time complexity of O (n log (n)). In worst case, Quick sort has time complexity of O (n^2). Unlike merge sort, quick sort is not stable. If Quick sort is implemented well, it will be aroundfaster than merge sort.