ANALYSIS AND COMPARISION OF DIFFERENT SORTING ALGORITHM

PROJECT REPORT

Submitted for the course: Data Structures and Algorithms (CSE 2003)

By

(Name of students with reg. number)

RAJESH KUMAR SINGH	**15BCE0519**	**G1 SLOT**

Name of faculty: Prof. BOOMINATHAN P

(SCHOOL OF COMPUTING SCIENCES AND ENGINEERING)

VIT® UNIVERSITY

(Estd. u/s 3 of UGC Act 1956)

VELLORE ▪ CHENNAI

MAY,2017

CERTIFICATE

This is to certify that the project work entitled "Analysis and Comparison of different sorting algorithms" that is being submitted by "Rajesh kumar Singh" for Data Structures and Algorithms (CSE 2003) is a record of bonafide work done under my supervision. The contents of this Project work, in full or in parts, have neither been taken from any other source nor have been submitted for any other CAL course.

Place :Vellore

Date: 02 MAY 2015

Signature of students:

(Rajesh Kumar Singh)

Signature of faculty:(Prof. Boominathan P)

ACKNOWLEDGEMENTS

We, group members would like to acknowledge our teacher Mr. Boominathan P for giving us the required inputs in form of thoughts and speech to understand the subject better. Furthermore, we would thank VIT University for being a supporter for the new projects. It could not be done without the whole team getting involved in the topic with their heart and soul.

(Rajesh kumar Singh)

CONTENTS

INTRODUCTION:

What is sorting?

In computer science a sorting algorithm is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order. Sorting is important for optimizing the use of other algorithms (such as search and merge algorithms) which require input data to be in sorted lists. It is also often useful for arranging data and for producing human-readable output. More formally, the output must satisfy two conditions:

1. The output is in non-decreasing order (each element is no smaller than the previous element according to the desired total order)
2. The output is in non-increasing order (each element is no larger than the previous element according to the desired total order)

Further, the data is often taken to be in an array, which allows random access, rather than a list, which only allows sequential access, though often algorithms can be applied with suitable modification to either type of data.

ABSTRACT

From the beginning of computing, the sorting problem has attracted a great deal of research, perhaps due to the complexity of solving it efficiently despite its simple and familiar statement. Among the authors of early sorting algorithms around 1951 was Betty Holberton, who worked on ENIAC and UNIVAC. Bubble sort was analyzed as early as 1956. Comparison sorting algorithms have a fundamental requirement of $O(n \log n)$ comparisons, algorithms not based on comparisons, such as counting sort, can have better performance. Although many consider sorting a solved problem, asymptotically optimal algorithms have been known since the mid-20th century, useful new algorithms are still being invented, with the now widely used Tim sort dating to 2002, and the library sort being first published in 2006. So this attracted us towards this project and we decided to compare some famous sorting algorithms on the basis of their time.

DIFFERENT TYPES OF SORTING ALGORITHMS

Till 2017, there are many sorting algorithms and many more are being developed from the previous algorithms. For a single sorting algorithm, there are many versions of it. So, it is impractical to analyze all the algorithms. Here we have taken six famous sorting algorithms.

1) **Bubble sort:** *Bubble sort* is a simple sorting algorithm. The algorithm starts at the beginning of the data set. It compares the first two elements, and if the first is greater than the second, it swaps them. It continues doing this for each pair of adjacent elements to the end of the data set. It then starts again with the first two elements, repeating until no swaps have occurred on the last pass. This algorithm's average time and worst-case performance is O(n^2), so it is rarely used to sort large, unordered data sets. Bubble sort can be used to sort a small number of items (where its asymptotic inefficiency is not a high penalty).

2) **Insertion sort:** *Insertion sort* is also a simple sorting algorithm that is relatively efficient for small lists and mostly sorted lists. It works by taking elements from the list one by one and inserting them in their correct position into a new sorted list. In arrays, the new list and the remaining elements can share the array's space, but insertion is expensive, requiring shifting all following elements over by one. Shell sort is a variant of insertion sort that is more efficient for larger lists.

3) **Selection sort:** *Selection sort* is an in-place comparison sort. It has O(n^2) complexity, making it inefficient on large lists, and generally performs worse than the similar insertion sort. Selection sort is noted for its simplicity, and has performance advantages over more complicated algorithms in certain situations. The algorithm finds the minimum value, swaps it with the value in the first position, and repeats these steps for the remainder of the list. It does no more than n swaps, and thus is useful where swapping is very expensive.

4) **Merge sort:** *Merge sort* takes advantage of the ease of merging already sorted lists into a new sorted list. It starts by comparing every two elements (i.e., 1 with 2, then 3 with 4...) and swapping them if the first should come after the second. It then merges each of the resulting lists of two into lists of four, then merges those lists of four, and so on; until at last two lists are merged into the final sorted list.[21] Of the algorithms described here, this is the first that scales well to very large lists, because its worst-case running time is O(n log n). It is also easily applied to lists, not only arrays, as it only requires sequential access, not random access. However, it has additional O(n) space complexity, and involves many copies in simple implementations.

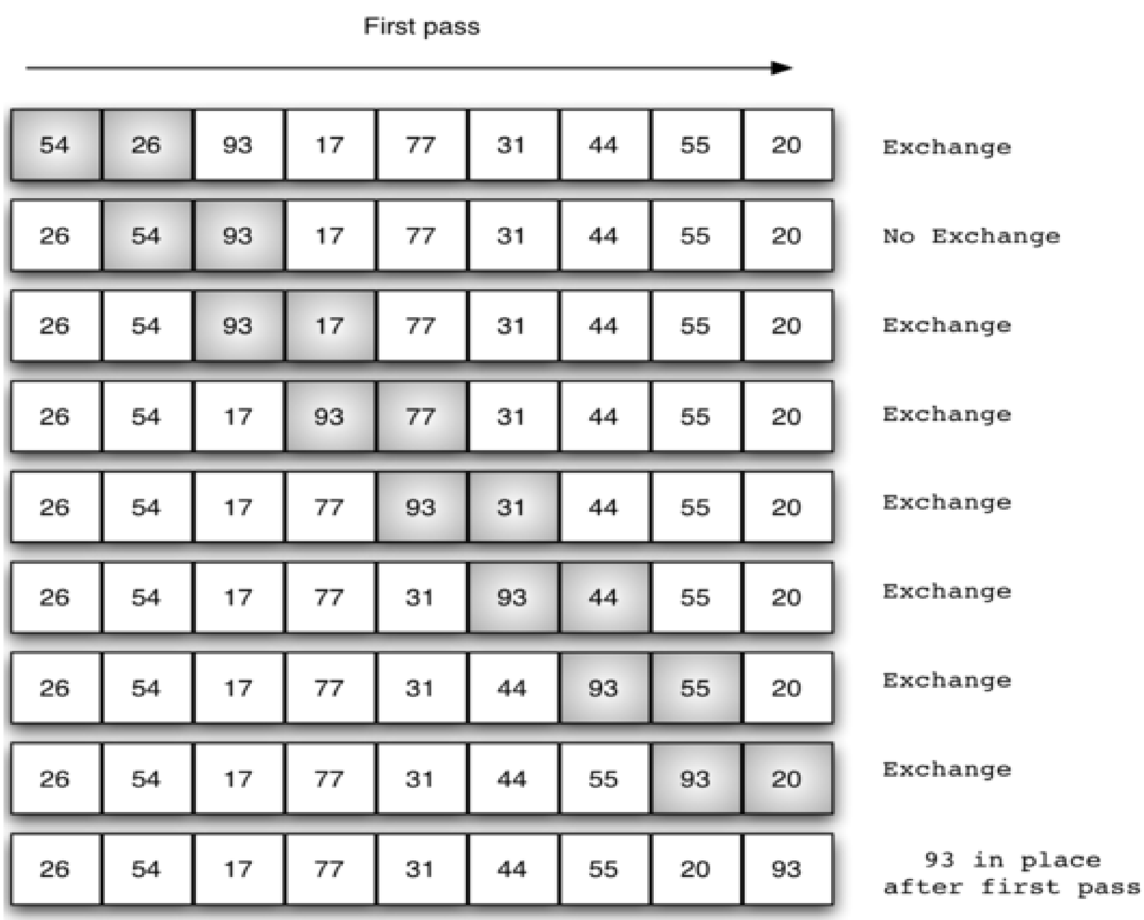
5) **Quick sort:** *Quicksort* is a divide and conquer algorithm which relies on a *partition* operation: to partition an array an element called a *pivot* is selected. All elements smaller than the pivot are moved before it and all greater elements are moved after it. This can be done efficiently in linear time and in-place. The lesser and greater sub-lists are then recursively sorted. This yields average time complexity of O (n log n), with low overhead, and thus this is a popular algorithm. Efficient implementations of quick sort are typically unstable sorts and somewhat complex, but are among the fastest sorting algorithms in practice. Together with its modest O (log n) space usage, quicksort is one of the most popular sorting algorithms and is available in many standard programming libraries.

6) **Heap sort:** *Heapsort* is a much more efficient version of selection sort. It also works by determining the largest (or smallest) element of the list, placing that at the end (or beginning) of the list, then continuing with the rest of the list, but accomplishes this task efficiently by using a data structure called a heap, a special type of binary tree. Once the data list has been made into a heap, the root node is guaranteed to be the largest (or smallest) element. When it is removed, and placed at the end of the list, the heap is rearranged so the largest element remaining moves to the root. Using the heap, finding the next largest element takes O (log n) time, instead of O(n) for a linear scan as in simple selection sort. This allows Heapsort to run in O (n log n) time, and this is also the worst-case complexity.

PSEUDOCODE AND EXAMPLE

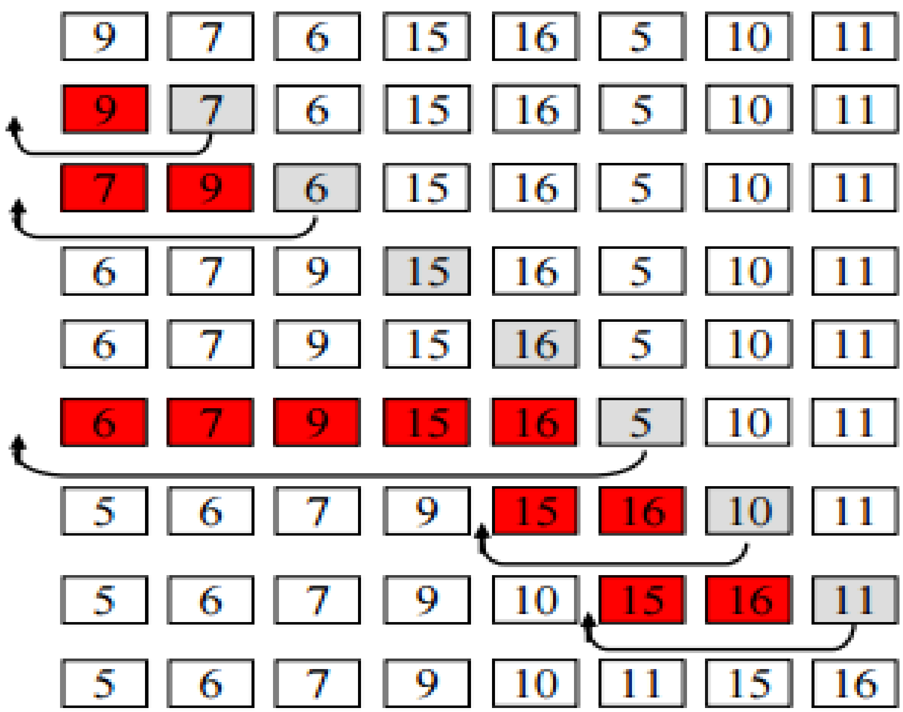
Bubble sort

```
func bubblesort( var a as array )
   for i from 1 to N
      for j from 0 to N - 1
         if a[j] > a[j + 1]
            swap( a[j], a[j + 1] )
end func
```



Insertion sort

```
INSERTION-SORT(A)
for j = 2 to n
    key ← A [j]
    // Insert A[j] into the sorted sequence A[1..j-1]
    j ← i – 1
   while i > 0 and A[i] > key
          A[i+1] ← A[i]
          i ← i – 1
   A[j+1] ← key
```



Selection sort

```
SELECTION-SORT(A)
for j ← 1 to n-1
     smallest ← j
     for i ← j + 1 to n
          if A[ i ] < A[ smallest ]
               smallest ← i
     Exchange A[ j ] ↔ A[ smallest ]
```

64 25 12 22 11 // this is the initial, starting state of the array

11 25 12 22 64 // sorted sublist = {11}

11 12 25 22 64 // sorted sublist = {11, 12}

11 12 22 25 64 // sorted sublist = {11, 12, 22}

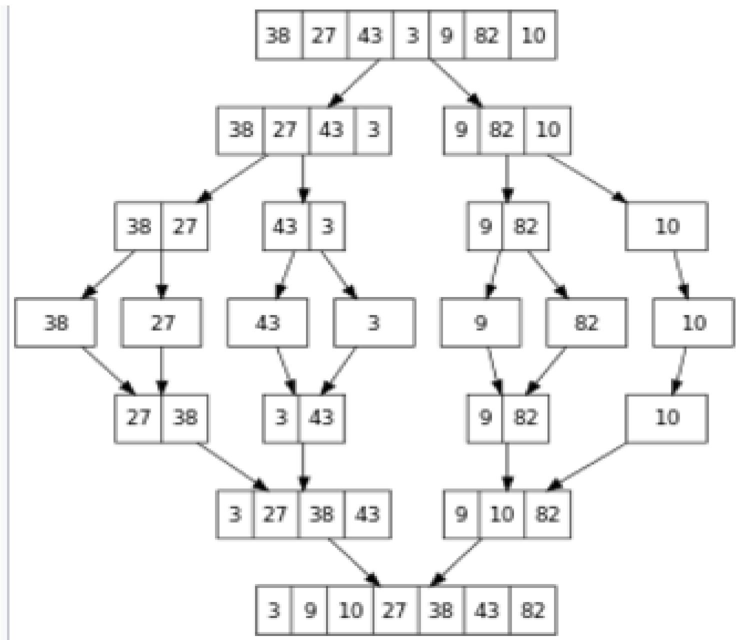
11 12 22 25 64 // sorted sublist = {11, 12, 22, 25}

11 12 22 25 64 // sorted sublist = {11, 12, 22, 25, 64}

Merge sort

```
void mergesort(int *a, int low, int high)
{   if (low < high)
    {   mid=(low+high)/2
        mergesort(a,low,mid)
        mergesort(a,mid+1,high)
        merge(a,low,high,mid)
    }
 }
void merge(int *a, int low, int high, int mid)
{   i = low, k = low,  j = mid + 1,
    while (i <= mid and j <= high)
         if (a[i] < a[j])
              c[k] = a[i]
              k++,    i++
        else
             c[k] = a[j]
              k++,   j++

    while (i <= mid)
         c[k] = a[i]
         k++,  i++
    while (j <= high)
```

```
        c[k] = a[j]
        k++,   j++
   for (i = low; i < k; i++)
           a[i] = c[i]
}
```

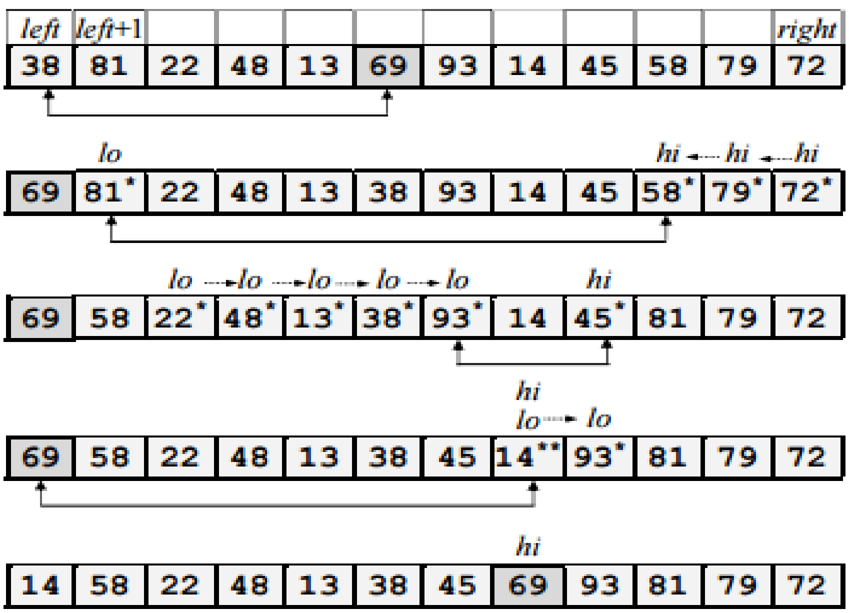


Quick sort

```
algorithm quicksort(A, lo, hi) is
   if lo < hi then
      p := partition(A, lo, hi)
      quicksort(A, lo, p)
      quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
   pivot := A[lo]
   i := lo
   j := hi + 1
   do
   {     do
            i++
         while(A[i]<pivot&&i<=hi);
```

```
do
   j--
while(pivot<A[j]);

if  i<j
   swap A[i] , A[j]
}while(i<j)
a[lo] = a[j]
a[j]=pivot
return j
```

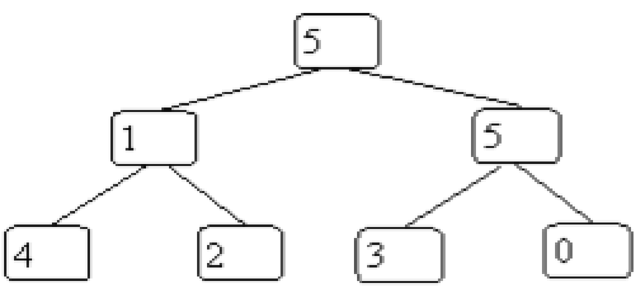
entire array is sorted by quicksort(A, 0, length(A))

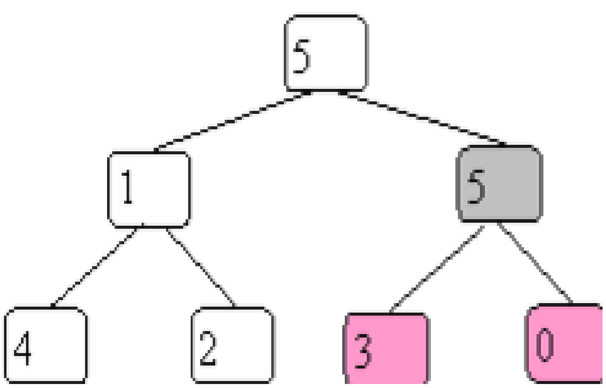


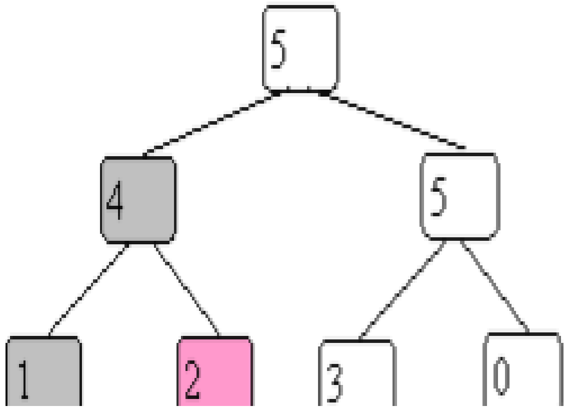
Heap sort

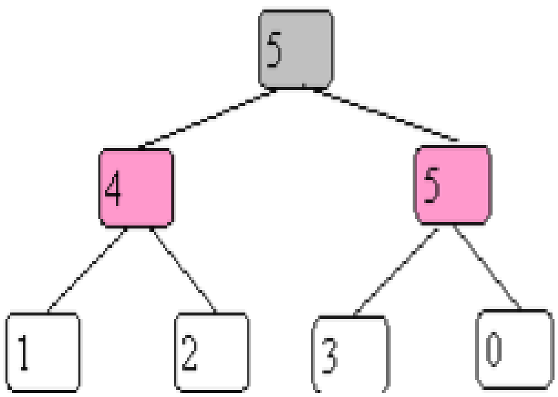
```
max_heapify(a, i,t n)
   temp = a[i]
   j = 2*i
   while (j <= n)
      if   j < n and a[j+1] > a[j]
         j = j+1
      if (temp > a[j])
         end loop
```

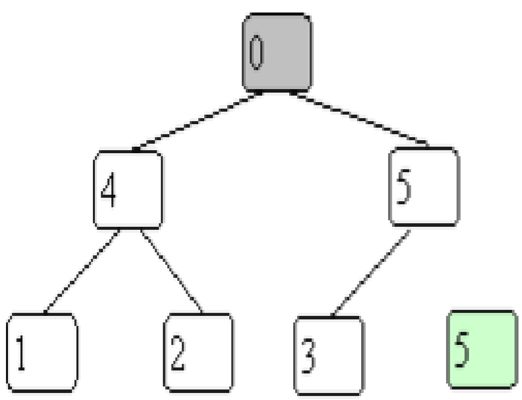
```
        else if temp <= a[j]
            a[j/2] = a[j]
            j = 2*j
 a[j/2] = temp
end func

heapsort(a, n)
     for (i = n, i >= 2, i--)
           swap a[i],a[1]
            call max_heapify(a, 1, i - 1)
end func

build_maxheap(a, n)
    for(i = n/2, i >= 1, i--)
           max_heapify(a, i, n);
end func

all will be called by
     callbuild_maxheap(a,n)
    Call heapsort(a, n)
```






5
4
5
1
2
3
0


5
4
5
1
2
3
0


0
4
5
1
2
3
5

Source code in c++

Bubble sort

```
#include<iostream>
using namespace std;
int main()
{
   int i,j,temp;
   int n=10;
   int a[];
   for(i=1;i<n;++i)
   {
      for(j=0;j<(n-i);++j)
         if(a[j]>a[j+1])
         {
            temp=a[j];
            a[j]=a[j+1];
            a[j+1]=temp;
         }
   }
   cout<<"Array after bubble sort:";
   for(i=0;i<n;++i)
      cout<<" "<<a[i];
   return 0;
}
```

Insertion sort

```
#include <iostream>
using namespace std;
int main()
{
   int i, j, k, temp;
   int n=10;
   int a[];
   for (i = 1; i <n; i++)
   {
      for (j = i; j >= 1; j--)
      {
```

```
            if (a[j] < a[j-1])
            {
               temp = a[j];
               a[j] = a[j-1];
               a[j-1] = temp;
            }
            else
               break;
         }
      }
      cout<<"sorted array\n"<<endl;
      for (k = 0; k <n; k++)
             cout<<a[k]<<endl;
      return 0;
}
```

Selection sort

```
#include<iostream>
 using namespace std;
 int main()
{
   int i,j,loc,temp,min;
   int n=10;
   int a[];
   for(i=0;i<n-1;i++)
   {
      min=a[i];
      loc=i;
      for(j=i+1;j<n;j++)
      {
         if(min>a[j])
         {
            min=a[j];
            loc=j;
         }
      }
       temp=a[i];
       a[i]=a[loc];
```

```
        a[loc]=temp;
    }
     cout<<"\nSorted list is as follows\n";
    for(i=0;i<n;i++)
         cout<<a[i]<<" ";
    return 0;
}
```

Merge sort

```
#include <iostream>
using namespace std;
#include <conio.h>
void merge(int *,int, int , int );
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,low,high,mid);
    }
    return;
}
void merge(int *a, int low, int high, int mid)
{
    int i, j, k, c[50];
    i = low;
    k = low;
    j = mid + 1;
    while (i <= mid && j <= high)
    {
        if (a[i] < a[j])
        {
            c[k] = a[i];
            k++;
            i++;
```

```
        }
        else
        {
            c[k] = a[j];
            k++;
            j++;
        }
    }
    while (i <= mid)
    {
        c[k] = a[i];
        k++;
        i++;
    }
    while (j <= high)
    {
        c[k] = a[j];
        k++;
        j++;
    }
    for (i = low; i < k; i++)
    {
        a[i] = c[i];
    }
}
int main()
{
    int i;
    int n=10,a[10];
    mergesort(a, 0, n);
    cout<<"sorted array\n";
    for (i = 0; i <n; i++)
       cout<<a[i];
   return 0;
}
```

Quick sort

```
#include <iostream>
using namespace std;
void quick_sort(int[],int,int);
int partition(int[],int,int);
int main()
{
   int i;
   int n=10,a[];
   quick_sort(a,0,n-1);
   cout<<"\nArray after sorting:";
   for(i=0;i<n;i++)
      cout<<a[i]<<" ";
   return 0;
}

void quick_sort(int a[],int l,int u)
{
   int j;
   if(l<u)
   {
      j=partition(a,l,u);
      quick_sort(a,l,j-1);
      quick_sort(a,j+1,u);
   }
}

int partition(int a[],int l,int u)
{
   int v,i,j,temp;
   v=a[l];
   i=l;
   j=u+1;
   do
   {   do
         i++;
       while(a[i]<v&&i<=u);
```

```
        do
            j--;
        while(v<a[j]);

        if(i<j)
        {
            temp=a[i];
            a[i]=a[j];
            a[j]=temp;
        }
    }while(i<j);
    a[l]=a[j];
    a[j]=v;
    return(j);
}
```

Heap sort

```
#include <iostream>
#include <conio.h>
using namespace std;
void max_heapify(int *a, int i, int n)
{
    int j, temp;
    temp = a[i];
    j = 2*i;
    while (j <= n)
    {
        if (j < n && a[j+1] > a[j])
            j = j+1;
        if (temp > a[j])
            break;
        else if (temp <= a[j])
        {
            a[j/2] = a[j];
            j = 2*j;
        }
    }
    a[j/2] = temp;
```

```
    return;
}

void heapsort(int *a, int n)
{
    int i, temp;
    for (i = n; i >= 2; i--)
    {
        temp = a[i];
        a[i] = a[1];
        a[1] = temp;
        max_heapify(a, 1, i - 1);
    }
}

void build_maxheap(int *a, int n)
{
    int i;
    for(i = n/2; i >= 1; i--)
    {
        max_heapify(a, i, n);
    }
}

int main()
{
    int i, x;
    int n=10,a[];

    build_maxheap(a,n);
    heapsort(a, n);
    cout<<"sorted output\n";
    for (i = 1; i <= n; i++)
    {
        cout<<a[i]<<endl;
    }
    return 0;
}
```

Results obtained

BUBBLE SORT

10 ELEMENTS

options | compilation | execution

```
bubble sort 10 elements random
Array after bubble sort: 1 2 3 4 5 6 7 8 9 10
37613ns

Exit code: 0 (normal program termination)
```

options | compilation | execution

```
bubble sort 10 elements partially
Array after bubble sort: 1 2 3 4 5 6 7 8 9 10
24644ns

Exit code: 0 (normal program termination)
```

options | compilation | execution

```
bubble sort 10 elements sorted
Array after bubble sort: 1 2 3 4 5 6 7 8 9 10
24170ns

Exit code: 0 (normal program termination)
```

options | compilation | execution

```
bubble sort 10 elements reverse
Array after bubble sort: 1 2 3 4 5 6 7 8 9 10
38672ns

Exit code: 0 (normal program termination)
```

options | compilation | execution

```
bubble sort 10 elements same
Array after bubble sort: 1 1 1 1 1 1 1 1 1 1
22157ns

Exit code: 0 (normal program termination)
```

50 ELEMENTS

options | compilation | execution | Stop

```
bubble sort 50 elements different
Array after bubble sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
45773ns

Exit code: 0 (normal program termination)
```

options | compilation | execution

```
bubble sort 50 elements partially
Array after bubble sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
51800ns

Exit code: 0 (normal program termination)
```

options | compilation | execution | Stop

```
bubble sort 50 elements sorted
Array after bubble sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
30107ns

Exit code: 0 (normal program termination)
```

options | compilation | execution | Stop

```
bubble sort 50 elements reverse
Array after bubble sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
47878ns

Exit code: 0 (normal program termination)
```

options | compilation | execution | Stop

```
bubble sort 50 elements same
Array after bubble sort: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
29193ns

Exit code: 0 (normal program termination)
```

100 ELEMENTS

```
options compilation execution                                                    Stop
bubble sort 100 elements different
Array after bubble sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 6
73643ns

Exit code: 0 (normal program termination)
```

```
options compilation execution                                                    Stop
bubble sort 100 elements partially
Array after bubble sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 6
48300ns

Exit code: 0 (normal program termination)
```

```
options compilation execution                                                    Stop
bubble sort 100 elements sorted
Array after bubble sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 6
43416ns

Exit code: 0 (normal program termination)
```

```
options compilation execution                                                    Stop
bubble sort 100 elements reverse
Array after bubble sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 6
65372ns

Exit code: 0 (normal program termination)
```

```
options compilation execution                                                    Stop
bubble sort 100 elements same
Array after bubble sort: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
39044ns

Exit code: 0 (normal program termination)
```

INSERTION SORT

10 ELEMENTS

options | compilation | execution

```
insertion sort 10 elements random
array after insertion sort: 1 2 3 4 5 6 7 8 9 10
25410ns

Exit code: 0 (normal program termination)
```

options | compilation | execution | Stop

```
insertion sort 10 elements partially
array after insertion sort: 1 2 3 4 5 6 7 8 9 10
25894ns

Exit code: 0 (normal program termination)
```

options | compilation | execution | Run

```
insertion sort 10 elements sorted
array after insertion sort: 1 2 3 4 5 6 7 8 9 10
24740ns

Exit code: 0 (normal program termination)
```

options | compilation | execution | Run

```
insertion sort 10 elements reverse
array after insertion sort: 1 2 3 4 5 6 7 8 9 10
25872ns

Exit code: 0 (normal program termination)
```

options | compilation | execution | Stop

```
insertion sort 10 elements same
array after insertion sort: 1 1 1 1 1 1 1 1 1 1
24859ns

Exit code: 0 (normal program termination)
```

50 ELEMENTS

options | compilation | execution | Run

```
insertion sort 50 elements different
array after insertion sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
30932ns

Exit code: 0 (normal program termination)
```

options | compilation | execution | Stop

```
insertion sort 50 elements partially
array after insertion sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
31753ns

Exit code: 0 (normal program termination)
```

options | compilation | execution | Stop

```
insertion sort 50 elements sorted
array after insertion sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
27981ns

Exit code: 0 (normal program termination)
```

options | compilation | execution | Cancel

```
insertion sort 50 elements reverse
array after insertion sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
49363ns

Exit code: 0 (normal program termination)
```

options | compilation | execution | Stop

```
insertion sort 50 elements same
array after insertion sort: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
27983ns

Exit code: 0 (normal program termination)
```

100 ELEMENTS

options | compilation | execution | Stop

```
insertion sort 100 elements different
array after insertion sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 6
37396ns
Exit code: 0 (normal program termination)
```

options | compilation | execution

```
insertion sort 100 elements partially
array after insertion sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 6
39299ns

Exit code: 0 (normal program termination)
```

options | compilation | execution | Stop

```
insertion sort 100 elements sorted
array after insertion sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 6
33699ns

Exit code: 0 (normal program termination)
```

options | compilation | execution | Stop

```
insertion sort 100 elements reverse
array after insertion sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 6
40853ns

Exit code: 0 (normal program termination)
```

options | compilation | execution | Stop

```
insertion sort 100 elements same
array after insertion sort: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
53860ns

Exit code: 0 (normal program termination)
```

SELECTION SORT

10 ELEMENTS

options | compilation | execution | Cancel

```
selection sort 10 elements random
array after selection Sort: 1 2 3 4 5 6 7 8 9 10
24966ns

Exit code: 0 (normal program termination)
```

options | compilation | execution | Stop

```
selection sort 10 elements partially
array after selection Sort: 1 2 3 4 5 6 7 8 9 10
25777ns

Exit code: 0 (normal program termination)
```

options | compilation | execution | Stop

```
selection sort 10 elements sorted
array after selection Sort: 1 2 3 4 5 6 7 8 9 10
25758ns

Exit code: 0 (normal program termination)
```

options | compilation | execution | Cancel

```
selection sort 10 elements reverse
array after selection Sort: 1 2 3 4 5 6 7 8 9 10
24749ns

Exit code: 0 (normal program termination)
```

options | compilation | execution | Stop

```
selection sort 10 elements same
array after selection Sort: 1 1 1 1 1 1 1 1 1 1
23756ns

Exit code: 0 (normal program termination)
```

50 ELEMENTS

options | compilation | execution

```
selection sort 50 elements different
array after selection Sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
36057ns

Exit code: 0 (normal program termination)
```

options | compilation | execution | Stop

```
selection sort 50 elements partially
array after selection Sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
31511ns

Exit code: 0 (normal program termination)
```

options | compilation | execution | Stop

```
selection sort 50 elements sorted
array after selection Sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
29704ns
```

options | compilation | execution

```
selection sort 50 elements reverse
array after selection Sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
44791ns

Exit code: 0 (normal program termination)
```

options | compilation | execution | Cancel

```
selection sort 50 elements same
array after selection Sort: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
30840ns

Exit code: 0 (normal program termination)
```

100 ELEMENTS

```
options | compilation | execution                                                                 Stop
selection sort 100 elements different
array after selection Sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 6
43762ns

Exit code: 0 (normal program termination)
```

```
options | compilation | execution                                                                 Run
selection sort 100 elements partially
array after selection Sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 6
41657ns

Exit code: 0 (normal program termination)
```

```
options | compilation | execution                                                                 Stop
selection sort 100 elements sorted
array after selection Sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 6
38831ns

Exit code: 0 (normal program termination)
```

```
options | compilation | execution                                                                 Run
selection sort 100 elements reverse
array after selection Sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 6
65838ns
```

```
options | compilation | execution                                                                 Run
selection sort 100 elements same
array after selection Sort: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
38731ns

Exit code: 0 (normal program termination)
```

MERGE SORT

10 ELEMENTS

options | compilation | execution | Sto

```
merge sort 10 elements random
array after merge sort: 1 2 3 4 5 6 7 8 9 10
24618ns

Exit code: 0 (normal program termination)
```

options | compilation | execution | Stop

```
merge sort 10 elements partially
array after merge sort: 1 2 3 4 5 6 7 8 9 10
26091ns

Exit code: 0 (normal program termination)
```

options | compilation | execution | Stop

```
merge sort 10 elements sorted
array after merge sort: 1 2 3 4 5 6 7 8 9 10
25758ns

Exit code: 0 (normal program termination)
```

options | compilation | execution | Stop

```
merge sort 10 elements reverse
array after merge sort: 1 2 3 4 5 6 7 8 9 10
25083ns

Exit code: 0 (normal program termination)
```

options | compilation | execution | Stop

```
merge sort 10 elements same
array after merge sort: 1 1 1 1 1 1 1 1 1 1
27282ns

Exit code: 0 (normal program termination)
```

50 ELEMENTS

options | compilation | execution | Stop

```
merge sort 50 elements different
array after merge sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
50105ns

Exit code: 0 (normal program termination)
```

options | compilation | execution | Stop

```
merge sort 50 elements partially
array after merge sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
49668ns
```

options | compilation | execution | Cancel

```
merge sort 50 elements sorted
array after merge sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
30507ns

Exit code: 0 (normal program termination)
```

options | compilation | execution | Stop

```
merge sort 50 elements reverse
array after merge sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
50510ns

Exit code: 0 (normal program termination)
```

options | compilation | execution | Stop

```
merge sort 50 elements same
array after merge sort: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
29704ns

Exit code: 0 (normal program termination)
```

100 ELEMENTS

options | compilation | execution   Run

```
merge sort 100 elements different
array after merge sort:0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 6
38746ns

Exit code: 0 (normal program termination)
```

options | compilation | execution   Stop

```
merge sort 100 elements partially
array after merge sort:0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 6
39038ns
```

options | compilation | execution   Stop

```
merge sort 100 elements sorted
array after merge sort:0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 6
38235ns

Exit code: 0 (normal program termination)
```

options | compilation | execution   Cancel

```
merge sort 100 elements reverse
array after merge sort:0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 6
59490ns

Exit code: 0 (normal program termination)
```

options | compilation | execution   Stop

```
merge sort 100 elements same
array after merge sort:0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
36994ns

Exit code: 0 (normal program termination)
```

QUICK SORT

10 ELEMENTS

options compilation execution Stop

```
quick sort 10 elements random
Array after quick sort: 1 2 3 4 5 6 7 8 9 10
25153ns

Exit code: 0 (normal program termination)
```

options compilation execution Run

```
quick sort 10 elements partially
Array after quick sort: 1 2 3 4 5 6 7 8 9 10
24598ns

Exit code: 0 (normal program termination)
```

options compilation execution Stop

```
quick sort 10 elements reverse
Array after quick sort: 1 2 3 4 5 6 7 8 9 10
24485ns

Exit code: 0 (normal program termination)
```

options compilation execution Stop

```
quick sort 10 elements same
Array after quick sort: 1 1 1 1 1 1 1 1 1 1
24778ns
```

options compilation execution Stop

```
quick sort 10 elements sorted
Array after quick sort: 1 2 3 4 5 6 7 8 9 10
22002ns

Exit code: 0 (normal program termination)
```

50 ELEMENTS

options | compilation | execution | Stop

```
quick sort 50 elements different
Array after quick sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
29524ns

Exit code: 0 (normal program termination)
```

options | compilation | execution | Stop

```
quick sort 50 elements partially
Array after quick sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
30813ns

Exit code: 0 (normal program termination)
```

options | compilation | execution | Stop

```
quick sort 50 elements sorted
Array after quick sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
29736ns

Exit code: 0 (normal program termination)
```

options | compilation | execution | Cancel

```
quick sort 50 elements reverse
Array after quick sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
48037ns
```

options | compilation | execution | Stop

```
quick sort 50 elements same
Array after quick sort: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
28716ns

Exit code: 0 (normal program termination)
```

100 ELEMENTS

options | compilation | execution | Stop

```
quick sort 100 elements different
Array after quick sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 6
38639ns

Exit code: 0 (normal program termination)
```

options | compilation | execution | Stop

```
quick sort 100 elements partially
Array after quick sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 6
37358ns
```

options | compilation | execution | Stop

```
quick sort 100 elements sorted
Array after quick sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 6
40089ns

Exit code: 0 (normal program termination)
```

options | compilation | execution | Stop

```
quick sort 100 elements reverse
Array after quick sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 6
37494ns

Exit code: 0 (normal program termination)
```

options | compilation | execution | Stop

```
quick sort 100 elements same
Array after quick sort: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
33047ns

Exit code: 0 (normal program termination)
```

HEAP SORT

10 ELEMENTS

options | compilation | execution

```
heap sort 10 elements random
Array after heap sort: 1 2 3 4 5 6 7 8 9 10
38370ns

Exit code: 0 (normal program termination)
```

options | compilation | execution

```
heap sort 10 elements partially
Array after heap sort: 1 2 3 4 5 6 7 8 9 10
26292ns

Exit code: 0 (normal program termination)
```

options | compilation | execution

```
heap sort 10 elements sorted
Array after heap sort: 1 2 3 4 5 6 7 8 9 10
30866ns

Exit code: 0 (normal program termination)
```

options | compilation | execution

```
heap sort 10 elements reverse
Array after heap sort: 1 2 3 4 5 6 7 8 9 10
24389ns

Exit code: 0 (normal program termination)
```

options | compilation | execution

```
heap sort 10 elements same
Array after heap sort: 1 1 1 1 1 1 1 1 1 1
25441ns

Exit code: 0 (normal program termination)
```

50 ELEMENTS

options | compilation | execution

```
heap sort 50 elements different
Array after heap sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
47978ns

Exit code: 0 (normal program termination)
```

options | compilation | execution

```
heap sort 50 elements partially
Array after heap sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
48890ns

Exit code: 0 (normal program termination)
```

options | compilation | execution

```
heap sort 50 elements sorted
Array after heap sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
49978ns
```

options | compilation | execution

```
heap sort 50 elements reverse
Array after heap sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
39376ns

Exit code: 0 (normal program termination)
```

options | compilation | execution

```
heap sort 50 elements same
Array after heap sort: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
33297ns

Exit code: 0 (normal program termination)
```

100 ELEMENTS

options | compilation | execution

```
heap sort 100 elements different
Array after heap sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
60837ns

Exit code: 0 (normal program termination)
```

options | compilation | execution

```
heap sort 100 elements partially
Array after heap sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
61040ns
```

options | compilation | execution

```
heap sort 100 elements sorted
Array after heap sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
64033ns

Exit code: 0 (normal program termination)
```

options | compilation | execution

```
heap sort 100 elements reverse
Array after heap sort: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
59365ns

Exit code: 0 (normal program termination)
```

options | compilation | execution

```
heap sort 100 elements same
Array after heap sort: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
55677ns

Exit code: 0 (normal program termination)
```

EXECUTION TIME FOR ALL ALGORITHMS

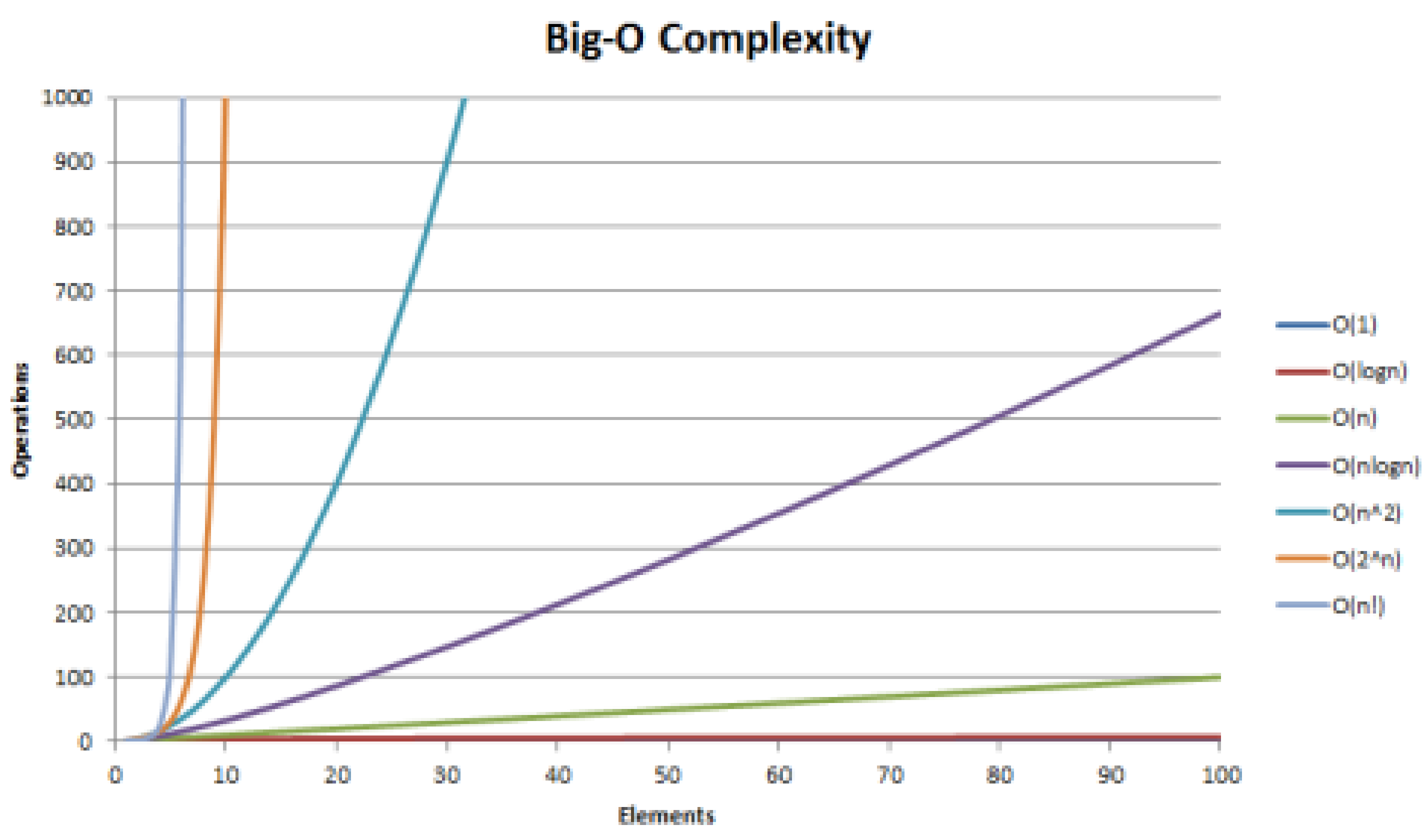
ALGORITHM	TEST CASES	RANDOM	PARTIALLY SORTED	SORTED	REVERSE SORTED	SAME
BUBBLE SORT	10	37613	24644	24170	38672	22157
	50	45773	51800	30107	47878	29193
	100	73643	48300	43416	65372	39044
INSERTION SORT	10	25410	25894	24740	25872	24859
	50	30932	31753	27981	49363	27983
	100	37396	39299	33699	40853	53860
SELECTION SORT	10	24966	25777	25758	24749	23756
	50	36057	31511	29704	44791	30840
	100	43762	41657	38831	65838	38731
MERGE SORT	10	24618	26091	25758	25083	27282
	50	50105	49668	30507	50510	29704
	100	38746	39038	38235	59490	36994
QUICK SORT	10	25153	24598	24485	24778	22002
	50	29524	30813	29736	48037	28716
	100	38639	37358	40089	37494	33047
HEAP SORT	10	38370	26292	30866	24389	25441
	50	47978	48890	49978	39376	33297
	100	60837	61040	64033	59365	55677

Table : execution time ,time are in nanoseconds(ns)

SPACE AND TIME COMPLEXITY

ALGORITHM	DATA STRUCTURE	TIME COMPLEXITY: BEST	TIME COMPLEXITY: AVERAGE	TIME COMPLEXITY: WORST	SPACE COMPLEXITY: WORST
BUBBLE SORT	ARRAY	$O(n)$	$O(n^2)$	$O(n^2)$	O(1)
INSERTION SORT	ARRAY	$O(n)$	$O(n^2)$	$O(n^2)$	O(1)
SELECTION SORT	ARRAY	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)
MERGE SORT	ARRAY	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$	O(n)
QUICK SORT	ARRAY	$O(n \log(n))$	$O(n \log(n))$	$O(n^2)$	O(n)
HEAP SORT	HEAP	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$	O(1)

TABLE : Time and space complexity for all algorithms



CONCLUSION

From the above analysis, it can be said that, Bubble Sort, Selection Sort and Insertion Sort are fairly straightforward, but they are relatively inefficient except for small lists. Merge Sort and Quick Sort are more complicated, but also much faster for large lists. Quick Sort is, on average, the fastest algorithm.

We find heap Sort algorithm and bubble sort algorithm is the slowest.

REFERENCES

[1] Seymour Lipschutz and G A Vijayalakshmi Pai , Data Structures, (Tata McGraw Hill companies), Indian adapted edition 2006-07 West patel nagar,New Delhi-110063.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest , Introduction to Algorithms, Fifth Indian printing (Prentice Hall of India private limited), New Delhi-110001

[3] Eshan Kapur, Parveen Kumar and Sahil Gupta,"Proposal Of A Two Way Sorting Algorithm And Performance Comparison With Existing Algorithms" International Journal of Computer Science, Engineering and Applications (IJCSEA) Vol.2, No.3, June 2012.

[4] Ahmed M. Aliyu, Dr. P. B. Zirra, "A Comparative Analysis of Sorting Algorithms on Integer and Character Arrays", The International Journal Of Engineering And Science (IJES), volume 2,Issue-7.

[5] Tarundeep Singh Sodhi, Surmeet Kaur, Snehdeep Kaur, "Enhanced Insertion Sort Algorithm ",International Journal of Computer Applications (0975 – 8887) Volume 64– No.21, February 2013.