DEPARTMENT OF COMPUTER ENGINEERING & APPLICATIONS

Practical File

DESIGN & ANALYSIS OF ALGORITHMS LAB

(BCSC0807)

Submitted To :

Mr. Dipak Kumar Shah

Submitted By:

Khushal Agarwal

(201500340)

Section – D(46)

Faculty Signature - ……………..

Index

Insertion Sort

Algorithm

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

Time Complexity

Best Case	n
Average Case	n^2
Worst Case	n^2

Space Complexity

O(n)

```
Run: insertionSort
"C:\Program Files\Java\jdk-16.0.1\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\I
Enter the size of the array
6
Enter the element in the array
45
32
87
46
90
1
45 32 87 46 90 1
1 32 45 46 87 90

Process finished with exit code 0
```

Bubble Sort:

Algorithm

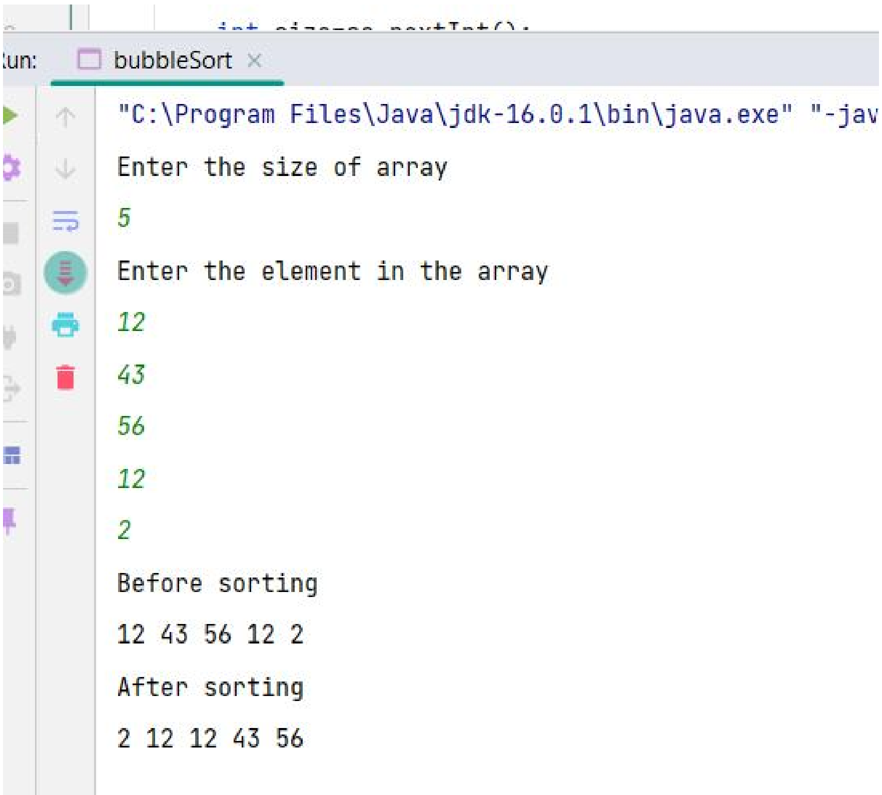
```
void bubble_sort (int a [ ], int n)
{
      int i,j, temp;
      for (i=0;ia[j+1])
      if (a[j]>a[j+1])
      {
            temp=a[j+1];
            a[j+1]=a[j];
            a[j]=temp;
            }
      }
 }
```

Time Complexity

Best Case	n
Average Case	n^2
Worst Case	n^2

Space Complexity

O(1)



Selection Sort

Algorithm

```
procedure selection sort
   list  : array of items
   n     : size of list

   for i = 1 to n - 1
   /* set current element as minimum*/
      min = i

      /* check the element to be minimum */

      for j = i+1 to n
         if list[j] < list[min] then
            min = j;
         end if
      end for

      /* swap the minimum element with the current element*/
      if indexMin != i  then
         swap list[min] and list[i]
```

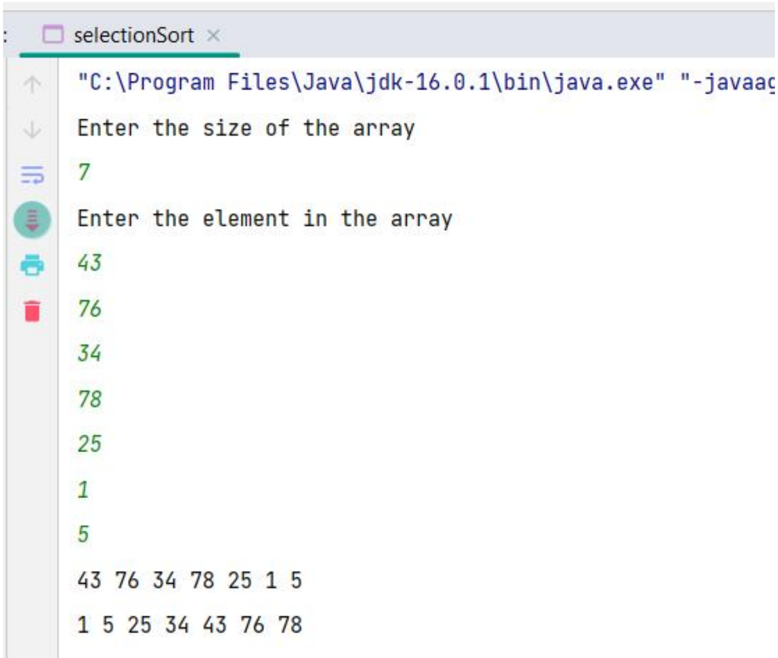
```
        end if
    end for

end procedure
```

Time Complexity

Best Case	n
Average Case	n^2
Worst Case	n^2

Space Complexity

O(1)



Merge Sort

```
procedure mergesort( var a as array )
   if ( n == 1 ) return a

   var l1 as array = a[0] ... a[n/2]
   var l2 as array = a[n/2+1] ... a[n]
```

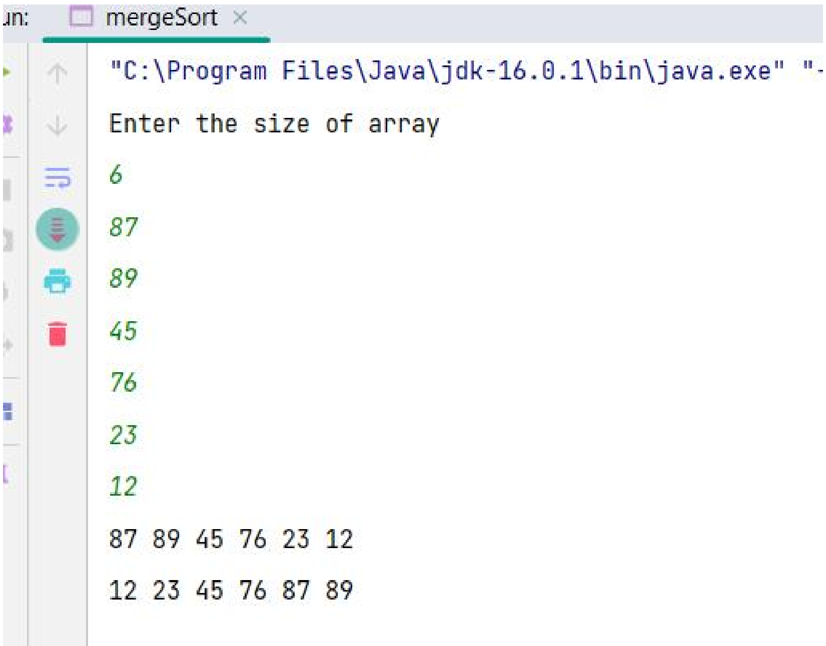
```
   l1 = mergesort( l1 )
   l2 = mergesort( l2 )

   return merge( l1, l2 )
end procedure

procedure merge( var a as array, var b as array )

   var c as array
   while ( a and b have elements )
      if ( a[0] > b[0] )
         add b[0] to the end of c
         remove b[0] from b
      else
         add a[0] to the end of c
         remove a[0] from a
      end if
   end while

   while ( a has elements )
      add a[0] to the end of c
      remove a[0] from a
   end while

   while ( b has elements )
      add b[0] to the end of c
      remove b[0] from b
   end while

   return c

end procedure
```

Time Complexity

```
Best Case       nlog(n)
Average Case    nlog(n)
Worst Case      nlog(n)
```

Space Complexity

```
O(n)
```



Quick Sort

Algorithm

```
function partitionFunc(left, right, pivot)
   leftPointer = left
   rightPointer = right - 1
   while True do
      while A[++leftPointer] < pivot do
         //do-nothing
      end while
      while rightPointer > 0 && A[--rightPointer] > pivot do
         //do-nothing
      end while
      if leftPointer >= rightPointer
```

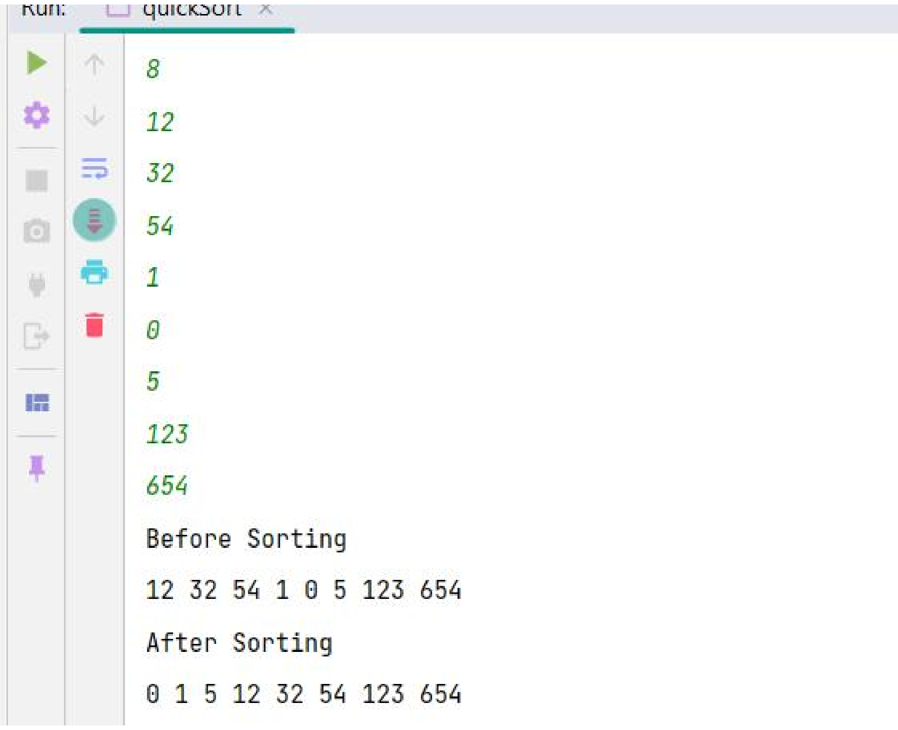
```
            break
        else
            swap leftPointer,rightPointer
        end if
    end while
    swap leftPointer,right
    return leftPointer
end function
```

Time Complexity

```
 Best Case      nlog(n)
 Average Case   nlog(n)
 Worst Case     n^2
```

Space Complexity

```
 O(n)
```



Counting Sort

Algorithm

```
Counting(a,b,k)
Let c[0—k] be a new array
```

```
For i=0 to k
     C[i]=0
For j=0 to a.length
     C[a[j]]=c[a[j]]+1
//c[i] now contain the frequency
For i=1 to k
     C[i]=c[i]+c[i-1]
//c[i] now contain the number of element less than or equal to
i
For j=a.length to 0
     B[c[a[j]]]=a[j]
     C[a[j]]=c[a[j]]-1
```

Time Complexity

Best Case	n+k
Average Case	n+k
Worst Case	n+k

Space Complexity

O(k)

```
int[] arr = { -5, -10, 0, -3, 8, 5, -1, 10 };
```

Output

```
-10 -5 -3 -1 0 5 8 10
```

Heap Sort

Algorithm

```
Heapify(A as array, n as int, i as int)
{
    max = i
    leftchild = 2i + 1
    rightchild = 2i + 2
    if (leftchild <= n) and (A[i] < A[leftchild])
        max = leftchild
```

```
    else
        max = i
    if (rightchild <= n) and (A[max]  > A[rightchild])
        max = rightchild
    if (max != i)
        swap(A[i], A[max])
        Heapify(A, n, max)
}
Heapsort(A as array)
{
   n = length(A)
   for i = n/2 downto 1
     Heapify(A, n ,i)

   for i = n downto 2
     exchange A[1] with A[i]
     A.heapsize = A.heapsize - 1
     Heapify(A, i, 0)
}
```

Time Complexity

```
 Best Case     nlog(n)
 Average Case  nlog(n)
 Worst Case    nlog(n)
```

Space Complexity

```
 O(1)
```

```
Enter the length of array
5
54
234
76
2
0
Before the sorting
54 234 76 2 0
After the sorting
0 2 54 76 234
```

Linear Search

Algorithm

```
procedure linear_search (list, value)
   for each item in the list
      if match item == value
         return the item's location
      end if
   end for
end procedure
```

Time Complexity

```
Best Case      n
Average Case   n
Worst Case     n
```

Space Complexity

```
O(1)
```

```
Enter the length of the array
3
Enter the element in the array
1
2
3
Enter the searching element
3
3
```

Binary Search

Algorithm

```
Procedure binary_search
   A ← sorted array
   n ← size of array
   x ← value to be searched
   Set lowerBound = 1
   Set upperBound = n
   while x not found
      if upperBound < lowerBound
         EXIT: x does not exists.
      set midPoint = lowerBound + ( upperBound - lowerBound )
/ 2
      if A[midPoint] < x
         set lowerBound = midPoint + 1
      if A[midPoint] > x
         set upperBound = midPoint - 1
      if A[midPoint] = x
         EXIT: x found at location midPoint
   end while
end procedure
```

Time Complexity

```
Best Case      o(1)
Average Case   log(n)
Worst Case     log(n)
```

Space Complexity

```
O(1)
```

```
Enter the length of the array
3
Enter the element in the array
1
2
3
Enter the searching element
3
3
```

Matrix Multiplication

```
public class Strassens {
   public static void main(String[] args) {
      int[][] x = {{12, 34}, {22, 10}};
      int[][] y = {{3, 4}, {2, 1}};
      int z[][] = new int[2][2];
      int m1, m2, m3, m4 , m5, m6, m7;

      System.out.print("The first matrix is: ");
      for(int i = 0; i<2; i++) {
         System.out.println();//new line
         for(int j = 0; j<2; j++) {
            System.out.print(x[i][j] + " ");
         }
      }
      System.out.print("\nThe second matrix is: ");
      for(int i = 0; i<2; i++) {
         System.out.println();//new line
         for(int j = 0; j<2; j++) {
            System.out.print(y[i][j] + " ");
         }
      }
      m1 = (x[0][0] + x[1][1]) * (y[0][0] + y[1][1]);
      m2 = (x[1][0] + x[1][1]) * y[0][0];
      m3 = x[0][0] * (y[0][1] - y[1][1]);
```

```
        m4 = x[1][1] * (y[1][0] - y[0][0]);
        m5 = (x[0][0] + x[0][1]) * y[1][1];
        m6 = (x[1][0] - x[0][0]) * (y[0][0]+y[0][1]);
        m7 = (x[0][1] - x[1][1]) * (y[1][0]+y[1][1]);
        z[0][0] = m1 + m4- m5 + m7;
        z[0][1] = m3 + m5;
        z[1][0] = m2 + m4;
        z[1][1] = m1 - m2 + m3 + m6;
        System.out.print("\nProduct achieved using Strassen's
algorithm: ");
        for(int i = 0; i<2; i++) {
           System.out.println();//new line
           for(int j = 0; j<2; j++) {
              System.out.print(z[i][j] + " ");
           }
        }
    }
}
```

Time Complexity

Worst Case $O(n^{\log 7})$

Breadth First Search

Algorithm

```
Procedure BFS(g,s)
     For each vertex v ∈v[g] do
          Explored[v] <- flase
          D[v]<- ∞
     End for
     Explored[s]<- true
     D[s]<-0
Q:- a queue data structure , initialized with s
While Q != φ do
     U<- remove vertex from the front of Q
     For each v adjacent to u do
          IF not explored[v] then
               Explored[v]<- true
               D[v]<-d[u}+1
               Insert v to the end of Q
```

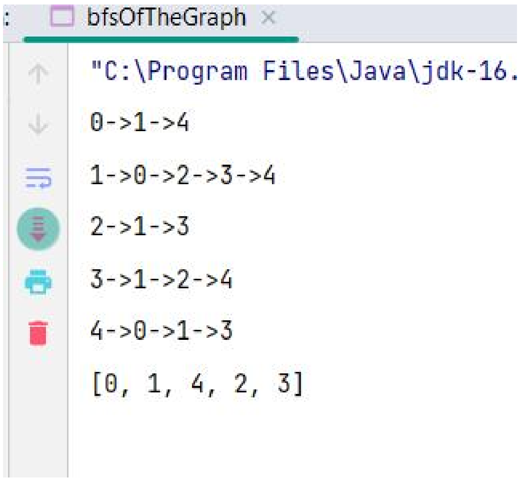
```
            End if
        End for
End while
End procedure
```

Time Complexity

```
 Best Case      o(V+E)
 Average Case   O(V+E)
 Worst Case     O(E+V)
```

Space Complexity

```
 O(V)
```



Depth First Search

Algorithm

```
DFS(G,v)   ( v is the vertex where the search starts )
         Stack S := {};   ( start with an empty stack )
         for each vertex u, set visited[u] := false;
         push S, v;
         while (S is not empty) do
            u := pop S;
            if (not visited[u]) then
               visited[u] := true;
               for each unvisited neighbour w of u
                  push S, w;
```

```
            end if
         end while
      END DFS()
```

Time Complexity

Best Case	o(V+E)
Average Case	O(V+E)
Worst Case	O(E+V)

Space Complexity

O(V)

Input: *n = 4, e = 6*

0 -> 1, 0 -> 2, 1 -> 2, 2 -> 0, 2 -> 3, 3 -> 3

Output: *DFS from vertex 1 : 1 2 0 3*

Prim’s

Algorithm

```
Prim(g,w,r)
     for each u ∈ g.v
          u.key= ∞
          u.pi=NIL
     r.key=0
     q=g.v
     while q!=null
          u=Extract-min(q)
          for each v∈g.Adj[u]
               if v ∈q and w(u, v) <v.key
                    v.pi=u
                    v.key =w(u,v)
```

Time Complexity

Best Case	o(E(log(V))
Average Case	O(V(log(v))+E(log(v)))
Worst Case	O(V(log(v))+E(log(v)))

Space Complexity

O(E+V)

Output

```
Edge     Weight
0 - 1     2
1 - 2     3
0 - 3     6
1 - 4     5
```

Kruskal's

Algorithm

```
A=θ
for each vertex v ∈g.v
     make-set(v)
sort the edge of g.e into non decreasing order by weight w
for each (u,e) ∈g.e taken in non decreasing order by weight
     if find-set(u)!= find-set(v)
          a= a U{(u,v)}
          Union(u,v)
Return A
```

Time Complexity

Best Case	O(ElogV)
Average Case	O(ElogV)
Worst Case	O(ElogV)

Space Complexity

O(E+V)

```
Following are the edges in the constructed MST
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
Minimum Cost Spanning Tree: 19
```

Dijkstra's Algorithm

Algorithm

```
function Dijkstra(Graph, source):

      for each vertex v in Graph.Vertices:
          dist[v] ← INFINITY
          prev[v] ← UNDEFINED
          add v to Q
      dist[source] ← 0

      while Q is not empty:
          u ← vertex in Q with min dist[u]
          remove u from Q

          for each neighbor v of u still in Q:
              alt ← dist[u] + Graph.Edges(u, v)
              if alt < dist[v]:
                  dist[v] ← alt
                prev[v] ← u

      return dist[], prev[]
```

Time Complexity

Best Case	O(ElogV)
Average Case	O(ElogV)
Worst Case	O(ElogV)

Space Complexity

O(V)

Output

```
Vertex          Distance from Source
0                  0
1                  4
2                  12
3                  19
4                  21
5                  11
6                  9
7                  8
8                  14
```

Bellman Ford

```
D[s]=0
R=v-s
C=cardinality(V)
For each vertex k ∈ do
     D[k]=infinity
End
For each vertex I =1 to (c-1) do
     For each edge(e1, e2) ∈ E do
          Relac(e1,e2)
     End
End
For each edge(e1,e2) ∈ E do
     If D[e2]>D[e1] +w[e1,e2] then
          Print("Graph contain negative weight cycle")
     End
End
Procedure Relax(e1, e2)
For each edge (e1,e2) in E do
     If D[e2]>D[e1] +w[e1,e2] then
          D[e2]>D[e1] +w[e1,e2]
     End
End
```

Time Complexity

Best Case	O(E)
Average Case	O(EV)
Worst Case	O(V^3)

Space Complexity

O(V)

Output

```
Vertex   Distance from Source
0          0
1          -1
2          2
3          -2
4          1
```