Time complexity

At the first iteration the length of the array = n

After each iteration length of the array is divided by half.

After 2nd iteration, Length of the array = $\frac{n}{2}$

After 3rd iteration, Length of the Array = $n/2^2$

:

After K^{th} iteration, Length of the Array = $\frac{n}{2^K}$

Also, we know that

After K iteration the length of Array become 1

$\therefore \frac{n}{2^K} = 1$

$\Rightarrow n = 2^K$

$\Rightarrow \log_2(n) = K$

$\therefore$ Time complexity is $O(\log n)$

Binary Search Algorithm

Algorithm : Binary Search (a[], x)

Where a[] is the input sorted Array.

x is the size of the input Array.

Step 1 : START

step 2 : Initialize low = 0, high = size - 1

step 3 : Repeat until low >= high

mid = (low + high)/2

if (a[mid == x)

then, write a[mid] and go to step

Else,

if (a[mid] < x)

low = mid + 1

else

high = mid - 1

Step 4 : write "x not found in the List"

step 5 : STOP.

Output :-

```
Enter the size of the array
5
Enter the elements
1 5 7 14 17
Enter the key to be searched
7
The searched element is present at index : 2
```

Code:

```
#include <iostream>
using namespace std;

bool arraySorted(int arr[], int n){
    if (n==0 or n==1){
        return 1;
    }
    for (int i=0; i<n; i++){
        if (arr[i-1] > arr[i]){
            return false;
        }
    }
    return false
}
int binarySearch(int array[], int n, int low, int high, int x){
    while (low <= high){
        int mid = low + (high - low)/2
    if (array[mid] == x)
        return mid
    if (array[mid] < x)
        low = mid + 1;
    else
        high = mid - 1;
    }
    return -1
}
```

```
int main (){
    int n,key;
    cout << "Enter the size of the Array" << endl;
    cin >> n;
    int * arr = new int [n],
    cout << "Enter the elements " << endl;
    for (int i=0; i<n; i++) {
        cin >> arr[i];
    }
    cout << "Enter the key to be searched" << endl;
    cin >> key;
    bool sort = arraySorted (arr, n);
    if (sort)
    {
        int result = binarySearch (arr, n, 0, n-1, key);
        if (result == -1) {
            cout << "Element not present" << endl;
        } else {
            cout << "The searched element present in
            index : " << result << endl;
        } else {
        cout << " The array is not sorted " << endl;
    }
    return 0;
}
```

Time complexity :-

Let $T(n)$ be the total time taken by Merge sort

1. Sorting two halves will take at most $2T(n/2)$ time.
2. Merging two lists we have to do $n-1$ comparisons because the last element copied down to the combined list.

Thus the relational formula will be

$$T(n) = 2T\left(\frac{n}{2}\right) + (n-1)$$

we ignore '-1' because the element will take some time

$$\text{so } T(n) = 2T\left(\frac{n}{2}\right) + n \longrightarrow (1)$$

Putting $n = \frac{n}{2}$ in place of n in (1)

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{2^2}\right) + \frac{n}{2} \longrightarrow (2)$$

Put (2) in (1)

$$T(n) = 2\left(2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right] + n$$

$$= 2^2 T\left(\frac{n}{2^2}\right) + \frac{2n}{2} + n$$

$$= 2^2 T\left(\frac{n}{2^2}\right) + 2n \longrightarrow (3)$$

similarly by putting $n = \frac{n}{2^2}$ in (1)

$$T(n) = 2^3 T\left(\frac{n}{2^3}\right) + 3n \longrightarrow (4)$$

from (1), (3), (5)

$$T(n) = 2^i T\left(\frac{n}{2^i}\right) + in \longrightarrow (5)$$

From stopping condition, $\frac{n}{2^i} = 1$ and $T\left(\frac{n}{2^i}\right) = 0$

$$n = 2^i$$

$$\Rightarrow \log_2 n = i$$

(5) → $T(n) = 2^i \times 0 + \log_2 n \times n = n \log n$

$\therefore O(n \log n)$

Merge Sort

Algorithm : Mergesort (A, P, R)

where A is the input array.

P, q, R are the indices in the Array

step 1 : START

step 2 : if (P < R)

$q = \lfloor (P+R)/2 \rfloor$

Mergesort (A, P, q)

Mergesort (A, q+1, R)

Mergesort (A, P, q, R)

step 3 : STOP

Algorithm : Merge (A, P, q, R)

where A is the input Array

P, q, R are the indices in the Array

step 1 : START

step 2 : $n_1 = q - P + 1$

step 3 : $n_2 = R - q$

step 4 : initialize $L[1 .. n_1 + 1]$ and $R[1 .. n_2 + 1]$ be new Arrays

step 5 : for $i = 1$ to n_1

$L[i] = A[P + i - 1]$

step 6 : for $j = 1$ to n_2

$R[j] = A[q + j]$

Step 8 : $L[n_1+1] = \infty$ /* Sentinel cards that have the special value to show smallest number */

Step 9 : $R[n_2+1] = \infty$

Step 10 : $i = 1$

Step 11 : $j = 1$

Step 12 : For $K = P$ to r

if $L[i] \leq R[j]$

$A[K] = L[i]$

$i = i + 1$

else $A[K] = R[j]$

$j = j + 1$

Step 13 : END

Output :-

Enter the number of elements

5.

7 9 2 3 1

1 2 3 7 9

Code :-

```
#include <iostream>
using namespace std;

void Merge (int arr[], int miniIndex, int middleIndex, int maxIndex)
{
  int left = miniIndex
  int right = middleIndex + 1;
  int count = maxIndex - minIndex + 1,
  int tempArray [100];
  int index = 0
while ( (left <= middleIndex) && (right <= maxIndex)) {
    if (arr [left] < arr [right])
    { tempArray [index] = arr [left];
      left ++;
    } else {
      tempArray [index] = arr [right]
      right ++;
    }
    index ++;
}
for (int i = left ; i <= middle index ; i++)
  { tempArray [index] = arr [i];
    index ++;
}
for (int i = right; i <= maxIndex ; i ++) {
    tempArray [index] = arr[i];
}
```

```
    for (int i = 0; i < count; i++){
        arr[minIndex + i] = tempArray[i];
    }
}
void MergeSort(int arr[], int minIndex, int maxIndex){
    if (minIndex < maxIndex){
        int middleIndex = (minIndex + maxIndex)/2;
        MergeSort(arr, minIndex, middleIndex);
        MergeSort(arr, middleIndex + 1, maxIndex);
        Merge(arr, minIndex, middleIndex, maxIndex);
    }
}
int main(){
    int n;
    cout << "Enter the number of elements" << endl;
    cin >> n
    int * mass = new int[n];
    for (int i = 0; i < n; i++){
        cin >> mass[i];
    }
    MergeSort(mass, 0, n-1);
    for (int i = 0; i < n; i++){
        cout << mass[i] << " ";
    }
}
```

Time complexity

In the final function of heapsort, we make use of Max-heap() which runs once to creat a heap and has a runtime of $O(n)$. Then using a for-loop we call the Max-heapify for each node, to maintain the max heap property whenever we remove or insert a node in the heap. Since there are 'n' number of nodes therefore, the total runtime of algorithm turns out to be $O(n(\log n))$ and we use the Max-Heapify() method for each node

Mathematically we see that

* The first removal of node takes $\log(n)$ time
* The 2nd removal takes $\log(n-1)$ time

⋮

The last removal takes $\log(1)$ time

so, summing up.

$$\log(n) + \log(n-1) + \log(n-2) + \dots + \log(1)$$

$$= \log(n \times (n-1) \times (n-2) \times \dots \times 2 \times 1)$$

$$= \log(n!)$$

upon further simplification (using stirling's approximation) $\log(n!)$ turns out to be

$$= n \times \log(n) - n + O(\log n)$$

Taking into account highest ordere term, the total runtime turns out to be $O(n \log n)$.

Heapsort :-

Algorithm: Max-Heapify (A, i)

where A is an array

i is the index in the array

step 1 : START

step 2 : l = LEFT (i) // returns 2i

step 3 : r = RIGHT (i) // returns 2i+1

step 4 : if (l ≤ A-heap-size and A[l] > A[i])

largest = i

step 5 : else largest = i

step 6 : if r ≤ A.heap size and A[r] > A [largest]

largest = r

step 7 : if (largest ≠ r)

swap (A [i], A [largest])

MAX-Heapify (A, largest)

step 8 : END

Algorithm : Max-Heap (A) // function to build a Heap

where A is the Array size

step 1 : START

step 2 : A heap-size = A length

step 3 : for i = [A.length/2] down to 1

MAX-Heapify (A, i)

step 4 : STOP.

Algorithm : Heapsort (A)

where A is the Array.

Step 1 : START

Step 2 : Max - Heap (A)

Step 3 : For i = A.length down to 2

swap (A[1], A[i]).

A.heapsize = A.heapsize - 1

Max - Heapify (A, 1).

Code :

```
#include <iostream>
using namespace std;
void heapify (int arr[], int n, int i){
    int largest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;
if (l < n && arr[l] > arr[largest])
    largest = l;
if (r < n && arr[r] > arr[largest])
    largest = r;
 if (largest != i) {
    swap (arr[i], arr[largest]) };
    heapify (arr, n, largest);
  }
}
void heapSort (int arr[], int n){
  for (int i = n/2 - 1; i >= 0; i--)
    heapify (arr, n, i);
  for (int i = n-1; i >= 0; i--) {
    swap(arr[0], arr[i]);
    heapify (arr, i, 0);
  }
}
```

```
void printArray (int arr[ ], int n) {
        for (int i=0 ; i<n ; i++)
            cout << arr[i] << "  " ;
        cout << "\n" ;
    }
int main () {
        int n;
        cin >> n;
        int * arr = new int (n);
        for (int i=0; i<n; i++) {
            cin >> arr[i] ;
        }
        heapsort (arr, n);
        cout << "a Sorted Array is \n" ;
        printArray (arr, n) ;
    }.
```

Time complexity

Let E be the number of edges present in the graph. We want the edges in increasing order by weight

To sort the edges the best sorting algorithm takes $O(N \log N)$ time.

So, it will take $E \log E$ time, where E = number of edges in the graph

Total complexity : Sorting time + edge Traversal (Merging + find - component).

Disjoint set data structure are designed to handle (merging + find - component).

∴ 1st case –

Sorting in $O(E)$, when edges are low

complexity : $E + E*N$, $E + E*\log N$.

2nd case –

Sorting in $O(E \log E)$

complexity : $E\log E + E*N$, $E \log E * \log N$

Algorithm: Kruskal(G)

where G is the graph

Step1 - START

Step2 - Initialize $A \leftarrow \emptyset$

Step3 - For each vertex $v \in G(v)$

create disjoint ~~$v \in G$~~ set MAKE-SET(v)

Step4 - For each edge $(u,v) \in G(E)$ taken in sorted order if (Find-set(u) ! Find-set(v))

$A \leftarrow A \cup \{(u,v)\}$

UNION of (u,v)

Step5 - STOP

Output : Enter total vertices and edges : 4 5

Enter source vertex destination vertex and weight 0 1 2

Enter " " " " " " 0 2 2

" " " " " " 0 3 1

" " " " " " 1 2 3

" " " " " " 2 3 1

0 0 0

3 0 1

2 3 1

1 0 2

Minimum Cost is : 4

Program :

```
#include <iostream>
#include <algorithm>
using namespace std;
class Edge {
public:
    int source;
    int dest;
    int weight;
};
bool compare (Edge e1, Edge e2)
{ return e1.weight < e2.weight;
}int find parent (int v, int *parent){
if (parent[v] == v){
    return v;
} return find parent (parent[v], parent);
}void Kruskal's (edge *input, int n, int E) {
sort (input, input + E, compare);
Edge * output = new Edge [n-1];
    int *parent = new int [n]
for (int i = 0; i < n; i++)
{ parent [i] = i;
}int count = 0;
 int i = 0
```

```
while (count != n-1){
  Edge currentedge = input[i]
  int sourceparent = findparent(currentedge source, parent)
  int destparent = findparent(currentedge-dest, parant);
if. (Source.parent != dest parent)
   output [count] = currentedge;
} i++;
 }
out <<"set the edges that can be constructed are" <<endl;
FOR (int i=0, i<n-1; i++)}
 if(output [i]; source < output [i]. dest]}
(out <<output [i] source <<"  " <<output [i] dest <<"
  <<output [i] weight << endl;
} else {
 (out <<output [i] dest <<" " << output [i]. source <<" "
  <<output [i] . weight << endl;
}}}
 }
 int main (){
 int n, E;
 (out <<" enter the no. of vertices and edges;"
  (in >> n >> E;
  Edge *input = new Edge [E];
```

```
for (int i=0; i<E ; i++){
    int s, d, w;
   (in >> s>> d>> w;
   input[i].source =s;
   input [i] . dest  =d;
   input [i] . weight = w
}Kruskals (input, n, E);
}
```

Time complexity—

using Binary Heap

1. The time complexity required for one call to EXTRACT-MIN (Q) is O(log V) using a min priority queue. The while loop at line 6 is executing total V times, so EXTRACT-MIN (Q) is called V times. So, the complexity of EXTRACT-MIN (Q) is O(log V).

2. The for loop at line 8 is executing total 2E times as length of each adjacency list is 2E for an undirected graph. The time required to execute line 11 is O (log V) by using the DECREASE-KEY operation on the min loop heap. Line 11 also executes total 2E times. So, the total time required to execute line 11 is O (2E log V) = O(E log V).

3. The for loop at line 1 will be executed V times. using the procedure to perform lines 1 to 5 will required a complexity of O(V)

Total time complexity of MST-PRIM is the sum of the time complexity required to execute step 1 through 3 for total of O (V log V + (E log V + V) = O (E log V)

Algorithm : MST - PRIM (G, w, r)

where $G \rightarrow$ Graph

$w \rightarrow$ weight

$r \rightarrow$ source vertex

Step 1 . For each $u \in V[G]$

do key $[u] \leftarrow \infty$

$\pi [u] \leftarrow$ NIL

Step 2 . key $[r] \leftarrow 0$

Step 3 . $Q \leftarrow V[G]$

Step 4 . while $Q \neq \emptyset$

do $u \leftarrow$ EXTRACT - MIN (Q)

For each $v \in$ Adj $[u]$

do if $v \in Q$ and $w(u,v) <$ key $[v]$

then $\pi [v] \leftarrow u$

key $[v] \leftarrow w(u,v)$

Step 5 - STOP

output -

Enter the no. of vertices and edges : 4 6

Enter source vertex destination vertex and weight : 0 1 2

" " " " " " " : 0 2 2

" " " " " " " : 0 3 1

" " " " " " " : 1 2 3

" " " " " " " : 2 3 1

0 2 1

0 1 2

0 3 2

Program

```
# include <iostream>
# include <vector>
# includ (set)
using namespace std;
Const int N=50
vector <vector <int >> g [N];
  int cost.
vector <int> vis (N) ; parent (N); dist (N);
const int INF = 20;
void primeMST (int source) {
for (int i=0; i<N; i++) {
 dist [i] = INF;
}
set <vector <int >>{ ;
 {.insert ({0, source});
while (!s · empty ()) {
auto x = * (s.begin ());
   s. erase (x);
 vis [x[1]] = true;
   int U = x[1];
   int V = parent [x[1]];
   int w = u [0] ;
cout <<U <<"  " <<v <<"  " <<w <<"\n";
    cost = cost + w;
  for (auto it : g[x[1]]){
    if (vis [it [0]])
       Continue
    if (dist [it [0]] it [1]){
       s. erase ({ dist [it[0]], it [0]} );
```

```
    dist [it[0]] = it[1];
    s.insert ({dist[it[0]], it[0]});
    parent[it[0]] = x[1]
    }
  }
}int main (){
int n, m;
cout << "enter total vertices and edges;"
   cin >> n >> m;
for (int i = 0; i < m; i++){
 int u, v, w;
cout << "enter source vertex destination vertex and weight
        respectively:";
   cin >> u >> v >> w;
  g[u].push_back ({v, w});
  g[v].push_back ({u, w});
}
  primesMST (0);
  cout << "Minimum cost is" << cost << endl;
    return 0;
  }
```

Time Complexity –

step 1 : —
step 2 : $O(1)$
step 3 : $O(1)$
step 4 : $O(n)$ → for loop n times.
step 5 : $O(n \log n)$ → time complexity of sorting algo.
step 6 : $O(n)$ → loop goes minimum n no of times
step 7 : $O(1)$
step 8 : $O(1)$
step 9 : $O(1)$
step 10 : $O(1)$
step 11 : $O(1)$

Total time complexity = $O(1) + O(n) + O(n \log n)$
$= O(n \log n)$ [taking highest complexity]

Algorithm : Huffman code (S, Q)

where S = string to be encoded

Q = Priority Queue .

Step 1 : START

step 2 : Read S, Q

Step 3 : Initialize n = S.size ( )

Step 4 : Repeat for i from 0 to n - 1

Create leaf nodes . N for character S [i]

Q . Push (N)

n = n + 1

Step 5 : sort all nodes in increasing order of characters frequency

Step 6 : Repeat Step 7 to 11 untill Q. size ( ) = 1

Step 7 : Create new internal node M

Step 8 : set M . left = Q . POP ( )

Step 9 : set M . right = Q . POP ( )

Step 10 : Set M . frequency = M . left frequency + M right frequency

Step 11 : Q push (M)

Step 12 : STOP

output :

Enter a text : Eerie eyes seen near lake

The huffman codes are :

	0 0
n	0 1 0
S	1 0 1 0
K	0 1 1 0
M	0 1 1 1
E	1 0 0 0 0
a	1 0 0 1
y	1 0 0 0 1
i	1 0 1 1 0
l	1 0 1 1 1
e	1 1

original string was Eerie eyes seen near lake

The encoded string is 1000011011110110611106111101 00

1016 1111610000101111001611160101111 06161011

```
#include <iostream>
#include <string>
#include <unordered_map>
#include <queue>
using namespace std;
  struct Node {
    char ch;
    int freq;
  Node * left* right;
}
Node* getNode (char ch. int freq. Node * left Node * right ) {
Node * node = new Node ();
  node ->ch =ch;
  node → freq = freq.
  node → left = left
  node → right = right
  return node,
} // comparison to order the heap
    struct comp {
    bool operator () (Node*l, Node*r) {
    return l → freq >r → freq;
   }
};
```

```
void encode (Node* root, string str, unordered_map<char, string>&
huffmancode {
if (root == nullptr) {
  return;
}// found a leaf node
if (!root->left && !root->right) {
  huffman Code [root->ch] = str;
} encode (root->left; str + "0" huffmancode);
  encode (root->right, str + "1" huffmancode);
} void decode (Node* root; int & topidx, string str) {
  if (root == nullptr) {
    return;
  }// found, a leaf node
  if (!root->right && !root->left) {
    cout << root->ch;
  return;
  } topidx++;
    if (str [topidx] == 0) {
      decode (root->right, topidx, str);
    }
}
```

```
void buildHuffmanTree(string text){
unordered_map<char, int> freq;
  for(char ch: text){
  freq[ch]++;
}
// to store the leaf nodes of the huffman tree
priority_queue<Node*, vector<Node*>, comp> pq
  for(auto pair: freq){
// make leaf node of each character and push to the priority queue
  pq.push(getNode(pair.first, pair.second, nullptr, nullptr));
}
// do the following till there is only one node in the priority queue
while(pq.size() != 1){
  // remove two nodes and from queue.
Node* left = pq.top(); pq.pop();
Node* right = pq.top; pq.pop();
// create an internal node with frequency equal to the sum of
the frequency of these two popped nodes
  int sum = left->freq + right->freq
  pq.push(getNode("\0" sum, left, right));
}
// root of the huffman tree
  Node* root = pq.top();
  unordered_map<char, string> huffmanCode
  encode(root, "", huffmancode);
```

```
cout << " the huffman codes are.\n";
for (auto pair : huffmancode){
 cout << pair.first << " " << pair.second << '\n';
}
cout << " original string was" << text << endl;
 string str = " ";
for (char ch; text) {
 str = huffmancode [ch];
}
cout << " The encoded string is " << str << endl;
  int index = 1
  cout << " \n decoded string is :\n";
  while (index < (int)str.size () - 2) {
  decode (root, index, str);
 }}
  int main () {
   string text ;
   cout << 'Enter a text" ;
    getline (cin, text );
   build Huffman Tree (text);
    return 0;
  }
```

Time complexity —

Worst case complexity

calculating ratios [0] v → v[i] / w[i] → O(n)

Sort descending order ratio → O(n log n)

```
if (w[i] < m)
m ← m - w[i]
profit → profit + P[i]
```

} O(n)

∴ Worst case time complexity = O(n) + O(n log n) + O(n)

= O(n log n)

Algorithm: func - Knapsack (P, w, M, n)

where : P is the profit earned
w is the weight
M is the maximum capacity
n is the size of the Array

step 1 : START

step 2 : for i = 1 to n

step 3 : compute P_i / w_i

step 4 : sort in descending order of P_i / w_i

step 5 : for i = 1 to n

if ((m > 0) and ($w_i \leq m$))

$m = m - w_i$

$P = P + P_i$

else,

BREAK

step 6 : if (m > 0)

$P = P + P_i \left(\frac{m}{w_i}\right)$;

step 7 : STOP

Enter size of the Array

Enter cost of item 1: 10

Enter weight of item 1: 15

Enter cost of item 2: 20

Enter weight of item 2: 25

Enter cost of item 3: 30

" weight " " 3 : 35

Enter cost of item 4 : 40

" weight " " 4 : 45

Enter cost of item 5 : 50

" weight " " 5 : 55

Enter capacity of Knapsack : 100

profit is 90.

```
#include <iostream>
#include <vector>
#include <utility>
#include <algorithm>
using namespace std;
int n = 50
vector <int> wt(n);
vector <int> cost (n);
float knapsack (float m) {
 vector <pair < float ; int >> ratio (n)
for (int i = 0; i <n ; it+) {
   ratio [i] . first = cost [i] / (wt-[i] *1.0);
   ratio [i] . second = i;
}sort (ratio . u begin( ), ratio . rend( );
 float profit = 0. 0
int i )
 for (i=0; i<n i++ ) {
 int j = ratio [i] . second;
ib {(m> && wt [j] < = m) {
   m = wt [j];
   profit + = cost [j];
} else
    break;
}
```

```
if (m>0) {
 int j=ratio [i].second;
    profit = cost [j]* (m/(wt [j]* 1.0));
}
return profit;
} int main () {
cout <<" Enter size of array ";
 ci >>n
for (int i=0; i<n; i++) {
 cout <<" Enter cost of item " <<i+1 <<" : ";
 cin >> cost [i]
 cout <<" enter weight of item" << i+1 <<" ; ";
 (in >> wt [i]
 cout << endl;
}
  float m;
  cout <<" enter capacity of knapsack; ";
   cin >> m;
   float profit = knapsack (m);
   cout <<" The profit is " << profit << endle;
}
```