**Sorting**

**Introduction**

Sorting is ordering a list of objects. We can distinguish two types of sorting. If the number of objects is small enough to fits into the main memory, sorting is called *internal sorting*. If the number of objects is so large that some of them reside on external storage during the sort, it is called *external sorting*. In this chapter we consider the following internal sorting algorithms

* Bucket sort
* Bubble sort
* Insertion sort
* Selection sort
* Heapsort
* Mergesort

**O(n) algorithms**

**Bucket Sort**

Suppose we need to sort an array of positive integers {3,11,2,9,1,5}. A bucket sort works as follows: create an array of size 11. Then, go through the input array and place integer 3 into a second array at index 3, integer 11 at index 11 and so on. We will end up with a sorted list in the second array.

Suppose we are sorting a large number of local phone numbers, for example, all residential phone numbers in the 412 area code region (about 1 million) We sort the numbers without use of comparisons in the following way. Create an a bit array of size 107. It takes about 1Mb. Set all bits to 0. For each phone number turn-on the bit indexed by that phone number. Finally, walk through the array and for each bit 1 record its index, which is a phone number.

We immediately see two drawbacks to this sorting algorithm. Firstly, we must know how to handle duplicates. Secondly, we must know the maximum value in the unsorted array.. Thirdly, we must have enough memory - it may be impossible to declare an array large enough on some systems.

The first problem is solved by using linked lists, attached to each array index. All duplicates for that bucket will be stored in the list. Another possible solution is to have a counter. As an example let us sort 3, 2, 4, 2, 3, 5. We start with an array of 5 counters set to zero.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | 0 | 0 | 0 | 0 | 0 |

Moving through the array we increment counters:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | 0 | 2 | 2 | 1 | 1 |

Next,we simply read off the number of each occurrence: 2 2 3 3 4 5.

**O(n2) algorithms**

### Bubble Sort

The algorithm works by co mparing each item in the list with the item next to it, and swapping them if required. In other words, the largest element has bubbled to the top of the array. The algorithm repeats this process until it makes a pass all the way through the list without swapping any items.

This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of Ο(n2) where **n** is the number of items.

void bubbleSort(int ar[])

{

for (int i = (ar.length - 1); i >= 0; i--)

{

for (int j = 1; j ≤ i; j++)

{

if (ar[j-1] > ar[j])

{

int temp = ar[j-1];

ar[j-1] = ar[j];

ar[j] = temp;

} } } }

begin BubbleSort(list)

for all elements of list

if list[i] > list[i+1]

swap(list[i], list[i+1])

end if

end for

return list

end BubbleSort

procedure bubbleSort( list : array of items )

loop = list.count;

for i = 0 to loop-1 do:

swapped = false

for j = 0 to loop-1 do:

/\* compare the adjacent elements \*/

if list[j] > list[j+1] then

/\* swap them \*/

swap( list[j], list[j+1] )

swapped = true

end if

end for

/\*if no number was swapped that means

array is sorted now, break the loop.\*/

if(not swapped) then

break

end if

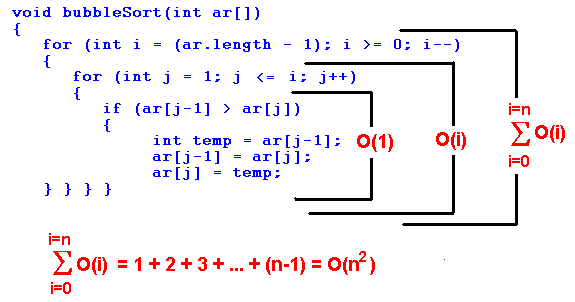
end for

end procedure return list

**Example.** Here is one step of the algorithm. The largest element - 7 - is bubbled to the top:

**7, 5**, 2, 4, 3, 9  
5, **7, 2**, 4, 3, 9  
5, 2, **7, 4**, 3, 9  
5, 2, 4, **7, 3**, 9  
5, 2, 4, 3, **7, 9**  
5, 2, 4, 3, 7, 9

The worst-case runtime complexity is O(n2). See explanation below



### Selection Sort

The algorithm works by selecting the smallest unsorted item and then swapping it with the item in the next position to be filled.

This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

The selection sort works as follows: you look through the entire array for the smallest element, once you find it you swap it (the smallest element) with the first element of the array. Then you look for the smallest element in the remaining array (an array without the first element) and swap it with the second element. Then you look for the smallest element in the remaining array (an array without first and second elements) and swap it with the third element, and so on. Here is an example,

void selectionSort(int[] ar){

for (int i = 0; i ‹ ar.length-1; i++)

{

int min = i;

for (int j = i+1; j ‹ ar.length; j++)

if (ar[j] ‹ ar[min]) min = j;

int temp = ar[i];

ar[i] = ar[min];

ar[min] = temp;

} }

**Example.**

**29**, 64, 73, 34, **20**,   
20, **64**, 73, 34, **29**,   
20, 29, **73**, **34**, 64   
20, 29, 34, **73**, **64**   
20, 29, 34, 64, 73

The worst-case runtime complexity is O(n2).

### Insertion Sort

To sort unordered list of elements, we remove its entries one at a time and then insert each of them into a sorted part (initially empty):

this is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, **insertion sort**.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array).

Following are some of the important **characteristics of Insertion Sort**:

1. It is efficient for smaller data sets, but very inefficient for larger lists.
2. Insertion Sort is adaptive, that means it reduces its total number of steps if a partially sorted array is provided as input, making it efficient.
3. It is better than Selection Sort and Bubble Sort algorithms.
4. Its space complexity is less. Like bubble Sort, insertion sort also requires a single additional memory space.
5. It is a **stable** sorting technique, as it does not change the relative order of elements which are equal.

void insertionSort(int[] ar)

{

for (int i=1; i ‹ ar.length; i++)

{

int index = ar[i]; int j = i;

while (j > 0 && ar[j-1] > index)

{

ar[j] = ar[j-1];

j--;

}

ar[j] = index;

} }

procedure insertionSort( A : array of items )

int holePosition

int valueToInsert

for i = 1 to length(A) inclusive do:

/\* select value to be inserted \*/

valueToInsert = A[i]

holePosition = i

/\*locate hole position for the element to be inserted \*/

while holePosition > 0 and A[holePosition-1] > valueToInsert do:

A[holePosition] = A[holePosition-1]

holePosition = holePosition -1

end while

/\* insert the number at hole position \*/

A[holePosition] = valueToInsert

end for

end procedure

**Example.** We color a sorted part in green, and an unsorted part in black. Here is an insertion sort step by step. We take an element from unsorted part and compare it with elements in sorted part, moving form right to left.

29, 20, 73, 34, 64   
**29**, 20, 73, 34, 64   
**20, 29**, 73, 34, 64   
**20, 29, 73**, 34, 64   
**20, 29, 34, 73**, 64   
**20, 29, 34, 64, 73**

Let us compute the worst-time complexity of the insertion sort. In sorting the most expensive part is a comparison of two elements. Surely that is a dominant factor in the running time. We will calculate the number of comparisons of an array of N elements:

we need 0 comparisons to insert the first element  
we need 1 comparison to insert the second element  
we need 2 comparisons to insert the third element  
...  
we need (N-1) comparisons (at most) to insert the last element

Totally,

1 + 2 + 3 + ... + (N-1) = O(n2)

The worst-case runtimecomplexity is O(n2).What is the best-case runtime complexity? O(n). The advantage of insertion sort comparing it to the previous two sorting algorithm is that insertion sort runs in linear time on nearly sorted data.

## O(n log n) algorithms

# Counting Sort Algorithm

Counting Sort Algorithm is an efficient sorting algorithm that can be used for sorting elements within a specific range. This sorting technique is based on the frequency/count of each element to be sorted and works using the following algorithm-

For simplicity, consider the data in the range 0 to 9.

Input data: 1, 4, 1, 2, 7, 5, 2

1) Take a count array to store the count of each unique object.

Index: 0 1 2 3 4 5 6 7 8 9

Count: 0 2 2 0 1 1 0 1 0 0

2) Modify the count array such that each element at each index

stores the sum of previous counts.

Index: 0 1 2 3 4 5 6 7 8 9

Count: 0 2 4 4 5 6 6 7 7 7

The modified count array indicates the position of each object in

the output sequence.

3) Output each object from the input sequence followed by

decreasing its count by 1.

Process the input data: 1, 4, 1, 2, 7, 5, 2. Position of 1 is 2.

Put data 1 at index 2 in output. Decrease count by 1 to place

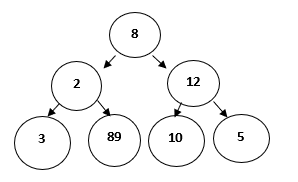
next data 1 at an index 1 smaller than this index.

# Greedy Approach or Technique

As the name implies, this is a simple approach which tries to find the **best** solution at every step. Thus, it aims to find the local optimal solution at every step so as to find the global optimal solution for the entire problem.

Consider that there is an **objective function** that has to be optimized (maximized/ minimized). This approach makes greedy choices at each step and makes sure that the objective function is optimized.

The greedy algorithm has only one chance to compute the optimal solution and thus, cannot go back and look at other alternate solutions. However, in many problems, this strategy fails to produce a global optimal solution. Let's consider the following binary tree to understand how a basic greedy algorithm works:



**When to use Greedy Algorithms?**

For a problem with the following properties, we can use the greedy technique:

* **Greedy Choice Property**: This states that a globally optimal solution can be obtained by locally optimal choices.
* **Optimal Sub-Problem**: This property states that an optimal solution to a problem, contains within it, optimal solution to the sub-problems. Thus, a globally optimal solution can be constructed from locally optimal sub-solutions.

Generally, **optimization problem**, or the problem where we have to find maximum or minimum of something or we have to find some optimal solution, greedy technique is used.

An optimization problem has two types of solutions:

* **Feasible Solution**: This can be referred as approximate solution (subset of solution) satisfying the objective function and it may or may not build up to the optimal solution.
* **Optimal Solution**: This can be defined as a feasible solution that either maximizes or minimizes the objective function.

**Key Terminologies used in Greedy Algorithms**

* **Objective Function**: This can be defined as the function that needs to be either maximized or minimized.
* **Candidate Set**: The global optimal solution is created from this set.
* **Selection Function**: Determines the best candidate and includes it in the solution set.
* **Feasibility Function**: Determines whether a candidate is feasible and can contribute to the solution.

**Advantages of Greedy Approach/Technique**

* This technique is easy to formulate and implement.
* It works efficiently in many scenarios.
* This approach minimizes the time required for generating the solution.

Now, let's see a few disadvantages too,

**Disadvantages of Greedy Approach/Technique**

* This approach does not guarantee a global optimal solution since it never looks back at the choices made for finding the local optimal solution.

Although we have already covered that which type of problem in general can be solved using greedy approach, here are a few popular problems which use greedy technique:

1. Knapsack Problem
2. Activity Selection Problem
3. Dijkstra’s Problem
4. Prim’s Algorithmfor finding Minimum Spanning Tree
5. Kruskal’s Algorithmfor finding Minimum Spanning Tree
6. Huffman Coding
7. Travelling Salesman Problem

**Conclusion**

Greedy Technique is best suited for applications where:

* Solution is required in real-time.
* Approximate solution is sufficien

**Activity Selection Problem**

The Activity Selection Problem is an optimization problem which deals with the selection of non-conflicting activities that needs to be executed by a single person or machine in a given time frame.

Each activity is marked by a start and finish time. Greedy technique is used for finding the solution since this is an optimization problem.

**What is Activity Selection Problem?**

Let's consider that you have n activities with their start and finish times, the objective is to find solution set having **maximum number of non-conflicting activities** that can be executed in a single time frame, assuming that only one person or machine is available for execution.

Some **points to note** here:

* It might not be possible to complete all the activities, since their timings can collapse.
* Two activities, say **i** and **j**, are said to be non-conflicting if si >= fj or sj >= fi where si and sj denote the starting time of activities **i** and **j** respectively, and fi and fj refer to the finishing time of the activities **i** and **j** respectively.
* **Greedy approach** can be used to find the solution since we want to maximize the count of activities that can be executed. This approach will greedily choose an activity with earliest finish time at every step, thus yielding an optimal solution.

**Input Data** for the Algorithm:

* act[] array containing all the activities.
* s[] array containing the starting time of all the activities.
* f[] array containing the finishing time of all the activities.

**Ouput Data** from the Algorithm:

* sol[] array refering to the solution set containing the maximum number of non-conflicting activities.

### Steps for Activity Selection Problem

Following are the steps we will be following to solve the activity selection problem,

**Step 1**: Sort the given activities in ascending order according to their finishing time.

**Step 2**: Select the first activity from sorted array act[] and add it to sol[] array.

**Step 3**: Repeat steps 4 and 5 for the remaining activities in act[].

**Step 4**: If the start time of the currently selected activity is greater than or equal to the finish time of previously selected activity, then add it to the sol[] array.

**Step 5**: Select the next activity in act[] array.

**Step 6**: Print the sol[] array.

**ime Complexity Analysis**

Following are the scenarios for computing the time complexity of Activity Selection Algorithm:

* **Case 1**: When a given set of activities are already sorted according to their finishing time, then there is no sorting mechanism involved, in such a case the complexity of the algorithm will be O(n)
* **Case 2**: When a given set of activities is unsorted, then we will have to use the sort() method defined in **bits/stdc++** header file for sorting the activities list. The time complexity of this method will be O(nlogn), which also defines complexity of the algorithm.

**Real-life Applications of Activity Selection Problem**

Following are some of the real-life applications of this problem:

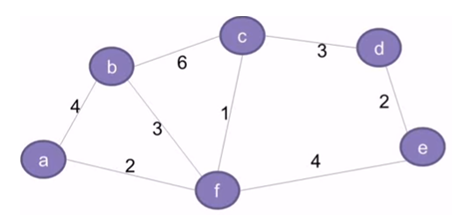
* Scheduling multiple competing events in a room, such that each event has its own start and end time.
* Scheduling manufacturing of multiple products on the same machine, such that each product has its own production timelines.
* Activity Selection is one of the most well-known generic problems used in Operations Research for dealing with real-life business problems.

**Prim's Minimum Spanning Tree**

In this tutorial we will cover another algorithm which uses greedy approach/technique for finding the solution.

Let's start with a real-life scenario to understant the premise of this algorithm:

1. A telecommunications organization, has offices spanned across multiple locations around the globe.
2. It has to use leased phone lines for connecting all these offices with each other.
3. The cost(in units) of connecting each pair of offices is different and is shown as follows:

Figure 2

1. The organization, thus, wants to use minimum cost for connecting all its offices. This requires that all the offices should be connected using minimum number of leased lines so as to reduce the effective cost.
2. The solution to this problem can be implemented by using the concept of **Minimum Spanning Tree**, which is discussed in the subsequent section.
3. This tutorial also details the concepts related to Prim's Algorithm which is used for finding the minimum spanning tree for a given graph.

**What is a Spanning Tree?**

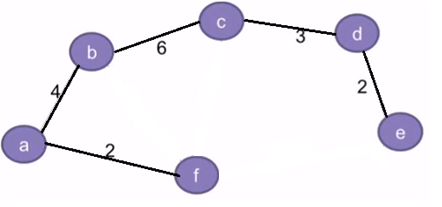
The network shown in the second figure basically represents a graph **G = (V, E)** with a set of vertices **V = {a, b, c, d, e, f}** and a set of edges **E = { (a,b), (b,c), (c,d), (d,e), (e,f), (f,a), (b,f), (c,f) }**. The graph is:

* Connected (there exists a path between every pair of vertices)
* Undirected (the edges do no have any directions associated with them such that (a,b) and (b,a) are equivalent)
* Weighted (each edge has a weight or cost assigned to it)

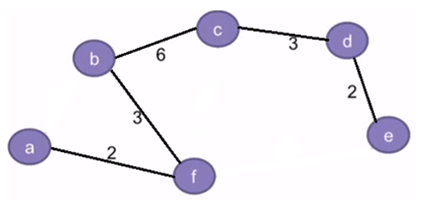
A spanning tree G' = (V, E') for the given graph G will include:

* All the vertices (V) of G
* All the vertices should be connected by minimum number of edges (E') such that E' ⊂ E
* G' can have maximum n-1 edges, where n is equal to the total number of edges in G
* G' should not have any cycles. This is one of the basic differences between a tree and graph that **a graph can have cycles, but a tree cannot**. Thus, a tree is also defined as an **acyclic graph**.

Following is an example of a spanning tree for the above graph. Please note that only the highlighted edges are included in the spanning tree,

Figure 3

Also, there can be multiple spanning trees possible for any given graph. For eg: In addition to the spanning tree in the above diagram, the graph can also have another spanning tree as shown below:

Figure 4

By convention, the total number of spanning trees for a given graph can be defined as:

nCm = n!/(m!\*(n-m)!), where,

* n is equal to the total number of edges in the given graph
* m is equal to the total number of edges in the spanning tree such that m <= (n-1).

Hence, the total number of spanning trees(S) for the given graph(second diagram from top) can be computed as follows:

* **n = 8**, for the given graph in Fig. 2
* **m = 5**, since its corresponding spanning tree can have only 5 edges. Adding a 6th edge can result in the formation of cycles which is not allowed.
* So, **S = nCm = 8C5 = 8!/ (5! \* 3!) = 56**, which means that **56** different variations of spannig trees can be created for the given graph.

**What is a Minimum Spanning Tree?**

The cost of a spanning tree is the total of the weights of all the edges in the tree. For example, the cost of spanning tree in Fig. 3 is **(2+4+6+3+2) = 17** units, whereas in Fig. 4 it is **(2+3+6+3+2) = 16** units.

Since we can have multiple spanning trees for a graph, each having its own cost value, the objective is to find the spanning tree with minimum cost. This is called a **Minimum Spanning Tree(MST)**.

**Note**: There can be multiple minimum spanning trees for a graph, if any two edges in the graph have the same weight. However, if each edge has a distinct weight, then there will be only one minimum spanning tree for any given graph.

**Problem Statement for Minimum Spanning Tree**

Given a weighted, undirected and connected graph **G**, the objective is to find the minimum spanning tree **G'** for G.

Apart from the Prim's Algorithm for minimum spanning tree, we also have Kruskal's Algorithm for finding minimum spanning tree.

However, this tutorial will only discuss the fundamentals of **Prim's Algorithm.**

Since this algorithm aims to find the spanning tree with minimum cost, it uses **greedy approach** for finding the solution.

As part of finding the or creating the minimum spanning tree fram a given graph we will be following these steps:

* Initially, the tree is empty.
* The tree starts building from a random source vertex.
* A new vertex gets added to the tree at every step.
* This continues till all the vertices of graph are added to the tree.

**Input Data** will be:

A **Cost Adjacency Matrix** for out graph **G**, say cost

**Output** will be:

A Spanning tree with minimum total cost

**Algorithm for Prim's Minimum Spanning Tree**

Below we have the complete logic, stepwise, which is followed in prim's algorithm:

**Step 1**: Keep a track of all the vertices that have been visited and added to the spanning tree.

**Step 2**: Initially the spanning tree is empty.

**Step 3**: Choose a random **vertex**, and add it to the spanning tree. This becomes the **root node**.

**Step 4**: Add a new vertex, say **x**, such that

1. **x** is not in the already built spanning tree.
2. **x** is connected to the built spanning tree using minimum weight edge. (Thus, **x** can be adjacent to any of the nodes that have already been added in the spanning tree).
3. Adding **x** to the spanning tree should not form cycles.

**Step 5**: Repeat the Step 4, till all the vertices of the graph are added to the spanning tree.

**Step 6**: Print the total cost of the spanning tree.

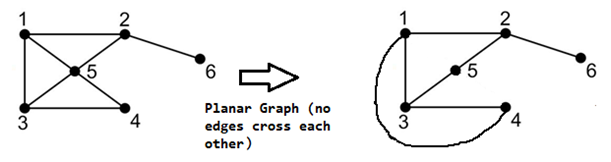
**Time Complexity Analysis for Prim's MST**

Time complexity of the above C++ program is **O(V2)** since it uses adjacency matrix representation for the input graph. However, using an adjacency list representation, with the help of binary heap, can reduce the complexity of Prim's algorithm to **O(ElogV)**.

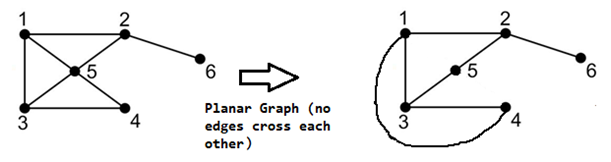
**Real-world Applications of a Minimum Spanning Tree**

Finding an MST is a fundamental problem and has the following real-life applications:

1. Designing the networks including computer networks, telecommunication networks, transportation networks, electricity grid and water supply networks.
2. Used in algorithms for approximately finding solutions to problems like Travelling Salesman problem, minimum cut problem, etc.
   * The objective of a **Travelling Salesman problem** is to find the shortest route in a graph that visits each vertex only once and returns back to the source vertex.
   * A **minimum cut problem** is used to find the minimum number of cuts between all the pairs of vertices in a planar graph. A graph can be classified as planar if it can be drawn in a plane with no edges crossing each other. For example,

Figure 24: Planar Graph

1. Also, a cut is a subset of edges which, if removed from a planar graph, increases the number of components in the graph

Figure 25: Cut-Set in a Planar Graph

1. Analysis of clusters.
2. Handwriting recognition of mathematical expressions.
3. Image registration and segmentation

# Huffman Coding Algorithm

Every information in computer science is **encoded** as strings of **1s and 0s**. The objective of information theory is to usually transmit information using fewest number of bits in such a way that every encoding is unambiguous. This tutorial discusses about fixed-length and variable-length encoding along with Huffman Encoding which is the basis for all data encoding schemes

Encoding, in computers, can be defined as the process of transmitting or storing sequence of characters efficiently. Fixed-length and variable lengthare two types of encoding schemes, explained as follows-

**Fixed-Length encoding** - Every character is assigned a binary code using same number of bits. Thus, a string like “aabacdad” can require 64 bits (8 bytes) for storage or transmission, assuming that each character uses 8 bits.

**Variable- Length encoding** - As opposed to Fixed-length encoding, this scheme uses variable number of bits for encoding the characters depending on their frequency in the given text. Thus, for a given string like “aabacdad”, frequency of characters ‘a’, ‘b’, ‘c’ and ‘d’ is 4,1,1 and 2 respectively. Since ‘a’ occurs more frequently than ‘b’, ‘c’ and ‘d’, it uses least number of bits, followed by ‘d’, ‘b’ and ‘c’. Suppose we randomly assign binary codes to each character as follows-

**a 0**   
**b 011**   
**c 111**   
**d 11**

Thus, the string “aabacdad” gets encoded to **00011011111011 (0 | 0 | 011 | 0 | 111 | 11 | 0 | 11),** using fewer number of bits compared to fixed-length encoding scheme.

### Problem Statement-

**Input:** Set of symbols to be transmitted or stored along with their frequencies/ probabilities/ weights

**Output:** Prefix-free and variable-length binary codes with minimum expected codeword length. Equivalently, a tree-like data structure with minimum weighted path length from root can be used for generating the binary codes

#### Huffman Encoding-

Huffman Encoding can be used for finding solution to the given problem statement.

* Developed by **David Huffman** in 1951, this technique is the basis for all data compression and encoding schemes
* It is a famous algorithm used for lossless data encoding
* It follows a Greedy approach, since it deals with generating minimum length prefix-free binary codes
* It uses variable-length encoding scheme for assigning binary codes to characters depending on how frequently they occur in the given text. The character that occurs most frequently is assigned the smallest code and the one that occurs least frequently gets the largest code

The major steps involved in Huffman coding are-

**Step I** - Building a Huffman tree using the input set of symbols and weight/ frequency for each symbol

* A Huffman tree, similar to a binary tree data structure, needs to be created having **n** leaf nodes and **n-1** internal nodes
* Priority Queue is used for building the Huffman tree such that nodes with lowest frequency have the highest priority. A Min Heap data structure can be used to implement the functionality of a priority queue.
* Initially, all nodes are leaf nodes containing the character itself along with the weight/ frequency of that character
* Internal nodes, on the other hand, contain weight and links to two child nodes

**Step II** - Assigning the binary codes to each symbol by traversing Huffman tree

* Generally, bit ‘0’ represents the left child and bit ‘1’ represents the right child

**Algorithm for creating the Huffman Tree-**

**Step 1**- Create a leaf node for each character and build a min heap using all the nodes (The frequency value is used to compare two nodes in min heap)

Step 2- Repeat Steps 3 to 5 while heap has more than one node

**Step 3**- Extract two nodes, say x and y, with minimum frequency from the heap

**Step 4**- Create a new internal node z with x as its left child and y as its right child. Also frequency(z)= frequency(x)+frequency(y)

**Step 5**- Add z to min heap

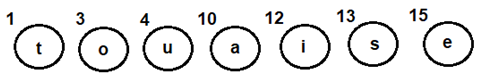
**Step 6**- Last node in the heap is the root of Huffman tree

Let’s try and create Huffman Tree for the following characters along with their frequencies using the above algorithm-

|  |  |
| --- | --- |
| Characters | Frequencies |
| a | 10 |
| e | 15 |
| i | 12 |
| o | 3 |
| u | 4 |
| s | 13 |
| t | 1 |

**Step A**- Create leaf nodes for all the characters and add them to the min heap.

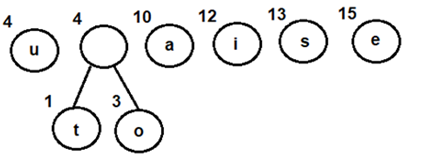
* **Step 1**- Create a leaf node for each character and build a min heap using all the nodes (The frequency value is used to compare two nodes in min heap)

Fig 1: Leaf nodes for each character

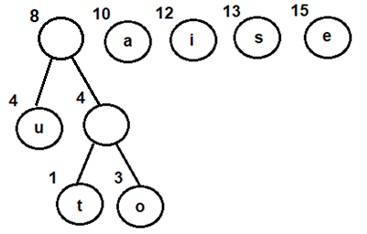
**Step B**- Repeat the following steps till heap has more than one nodes

* **Step 3**- Extract two nodes, say x and y, with minimum frequency from the heap
* **Step 4**- Create a new internal node z with x as its left child and y as its right child. Also frequency(z)= frequency(x)+frequency(y)
* **Step 5**- Add z to min heap

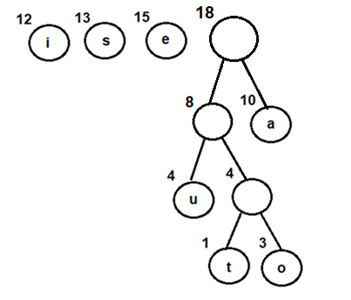
1. Extract and Combinenode u with an internal node having 4 as the frequency
2. Add the new internal node to priority queue-

Fig 2: Combining nodes o and t

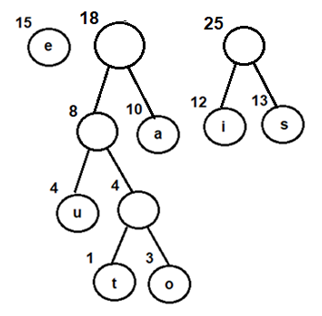
1. Extract and Combine node awith an internal node having 8 as the frequency
2. Add the new internal node to priority queue-

Fig 3: Combining node u withan internal node having 4 as frequency

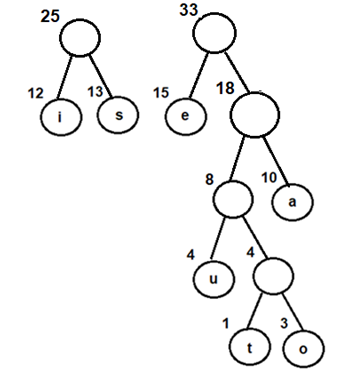
1. Extract and Combine nodes i and s
2. Add the new internal node to priority queue-

Fig 4: Combining node u withan internal node having 4 as frequency

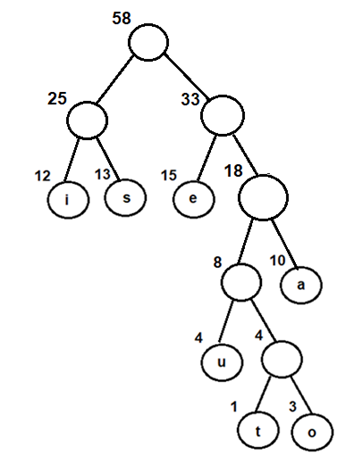
1. Extract and Combine nodes i and s
2. Add the new internal node to priority queue-

Fig 5: Combining nodes i and s

1. Extract and Combine node ewith an internal node having 18 as the frequency
2. Add the new internal node to priority queue-

Fig 6: Combining node e with an internal node having 18 as frequency

1. Finally, Extract and Combine internal nodes having 25 and 33 as the frequency
2. Add the new internal node to priority queue-

Fig 7: Final Huffman tree obtained by combining internal nodes having 25 and 33 as frequency

Now, since we have only one node in the queue, the control will exit out of the loop

**Step C**- Since internal node with frequency 58 is the only node in the queue, it becomes the root of **Huffman tree**.

**Step 6**- Last node in the heap is the root of Huffman tree

### Steps for traversing the Huffman Tree

1. Create an auxiliary array
2. Traverse the tree starting from root node
3. Add 0 to arraywhile traversing the left child and add 1 to array while traversing the right child
4. Print the array elements whenever a leaf node is found

Following the above steps for Huffman Tree generated above, we get prefix-free and variable-length binary codes with minimum expected codeword length-

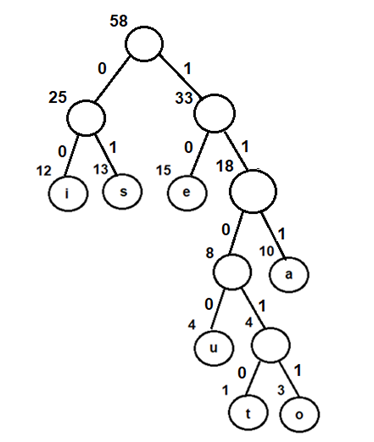


Fig 8: Assigning binary codes to Huffman tree

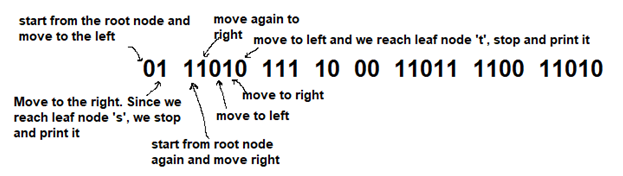
|  |  |
| --- | --- |
| Characters | Binary Codes |
| i | 00 |
| s | 01 |
| e | 10 |
| u | 1100 |
| t | 11010 |
| o | 11011 |
| a | 111 |

**Using the above binary codes-**

Suppose the string “staeiout” needs to be transmitted from computer A (sender) to computer B (receiver) across a network. Using concepts of Huffman encoding, the string gets encoded to **“0111010111100011011110011010” (01 | 11010 | 111 | 10 | 00 | 11011 | 1100 | 11010)** at the sender side.

Once received at the receiver’s side, it will be decoded back by traversing the Huffman tree. For decoding each character, we start traversing the tree from root node. Start with the first bit in the string. A ‘1’ or ‘0’ in the bit stream will determine whether to go left or right in the tree. Print the character, if we reach a leaf node.

**Thus for the above bit stream**

Fig 9: Decoding the bit stream

**On similar lines-**

* 111 gets decoded to ‘a’
* 10 gets decoded to ‘e’
* 00 gets decoded to ‘i’
* 11011 gets decoded to ‘o’
* 1100 gets decoded to ‘u’
* And finally, 11010 gets decoded to ‘t’, thus returning the string “staeiout” back

**Implementation-**

Following is the C++ implementation of Huffman coding. The algorithmcan be mapped to any programming language as per the requirement.

#include <iostream>

#include <vector>

#include <queue>

#include <string>

using namespace std;

class Huffman\_Codes

{

struct New\_Node

{

char data;

size\_t freq;

New\_Node\* left;

New\_Node\* right;

New\_Node(char data, size\_t freq) : data(data),

freq(freq),

left(NULL),

right(NULL)

{}

~New\_Node()

{

delete left;

delete right;

}

};

struct compare

{

bool operator()(New\_Node\* l, New\_Node\* r)

{

return (l->freq > r->freq);

}

};

New\_Node\* top;

void print\_Code(New\_Node\* root, string str)

{

if(root == NULL)

return;

if(root->data == '$')

{

print\_Code(root->left, str + "0");

print\_Code(root->right, str + "1");

}

if(root->data != '$')

{

cout << root->data <<" : " << str << "\n";

print\_Code(root->left, str + "0");

print\_Code(root->right, str + "1");

}

}

public:

Huffman\_Codes() {};

~Huffman\_Codes()

{

delete top;

}

void Generate\_Huffman\_tree(vector<char>& data, vector<size\_t>& freq, size\_t size)

{

New\_Node\* left;

New\_Node\* right;

priority\_queue<New\_Node\*, vector<New\_Node\*>, compare > minHeap;

for(size\_t i = 0; i < size; ++i)

{

minHeap.push(new New\_Node(data[i], freq[i]));

}

while(minHeap.size() != 1)

{

left = minHeap.top();

minHeap.pop();

right = minHeap.top();

minHeap.pop();

top = new New\_Node('$', left->freq + right->freq);

top->left = left;

top->right = right;

minHeap.push(top);

}

print\_Code(minHeap.top(), "");

}

};

int main()

{

int n, f;

char ch;

Huffman\_Codes set1;

vector<char> data;

vector<size\_t> freq;

cout<<"Enter the number of elements \n";

cin>>n;

cout<<"Enter the characters \n";

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

{

cin>>ch;

data.insert(data.end(), ch);

}

cout<<"Enter the frequencies \n";

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

{

cin>>f;

freq.insert(freq.end(), f);

}

size\_t size = data.size();

set1.Generate\_Huffman\_tree(data, freq, size);

return 0;

}

**Time Complexity Analysis-**

Since Huffman coding uses min Heap data structure for implementing priority queue, the complexity is O(nlogn). This can be explained as follows-

* Building a min heap takes O(nlogn) time (Moving an element from root to leaf node requires O(logn) comparisons and this is done for n/2 elements, in the worst case).
* Building a min heap takes O(nlogn) time (Moving an element from root to leaf node requires O(logn) comparisons and this is done for n/2 elements, in the worst case).

Since building a min heap and sorting it are executed in sequence, the algorithmic complexity of entire process computes to O(nlogn)

We can have a linear time algorithm as well, if the characters are already sorted according to their frequencies.

**Advantages of Huffman Encoding-**

* This encoding scheme results in saving lot of storage space, since the binary codes generated are variable in length
* It generates shorter binary codes for encoding symbols/characters that appear more frequently in the input string
* The binary codes generated are prefix-free

**Disadvantages of Huffman Encoding-**

* Lossless data encoding schemes, like Huffman encoding, achieve a lower compression ratio compared to lossy encoding techniques. Thus, lossless techniques like Huffman encoding are suitable only for encoding text and program files and are unsuitable for encoding digital images.
* Huffman encoding is a relatively slower process since it uses two passes- one for building the statistical model and another for encoding. Thus, the lossless techniques that use Huffman encoding are considerably slower than others.
* Since length of all the binary codes is different, it becomes difficult for the decoding software to detect whether the encoded data is corrupt. This can result in an incorrect decoding and subsequently, a wrong output.

**Real-life applications of Huffman Encoding-**

* Huffman encoding is widely used in compression formats like GZIP, PKZIP (winzip) and BZIP2.
* Multimedia codecs like JPEG, PNG and MP3 uses Huffman encoding (to be more precised the prefix codes)
* Huffman encoding still dominates the compression industry since newer arithmetic and range coding schemes are avoided due to their patent issues.

**Dijkstra's Algorithm**

Dijkstra's algorithm, published in 1959, is named after its discoverer Edsger Dijkstra, who was a Dutch computer scientist. This algorithm aims to find the shortest-path in a directed or undirected graph with non-negative edge weights.

Before, we look into the details of this algorithm, let’s have a quick overview about the following:

* **Graph**: A graph is a non-linear data structure defined as G=(V,E) where V is a finite set of vertices and E is a finite set of edges, such that each edge is a line or arc connecting any two vertices.
* **Weighted graph**: It is a special type of graph in which every edge is assigned a numerical value, called weight
* **Connected graph**: A path exists between each pair of vertices in this type of graph
* **Spanning tree** for a graph G is a subgraph G’ including all the vertices of G connected with minimum number of edges. Thus, for a graph G with n vertices, spanning tree G’ will have n vertices and maximum n-1 edges.

**Problem Statement**

Given a weighted graph G, the objective is to find the shortest path from a given source vertex to all other vertices of G. The graph has the following characteristics-

* Set of vertices V
* Set of weighted edges E such that (q,r) denotes an **edge** between **vertices** q and r and cost(q,r) denotes its weight

**Dijkstra's Algorithm:**

* This is a single-source shortest path algorithm and aims to find solution to the given problem statement
* This algorithm works for both directed and undirected graphs
* It works only for connected graphs
* The graph should not contain negative edge weights
* The algorithm predominantly follows Greedy approach for finding locally optimal solution. But, it also uses Dynamic Programming approach for building globally optimal solution, since the previous solutions are stored and further added to get final distances from the source vertex
* The main logic of this algorithm is basedon the following formula-   
  dist[r]=min(dist[r], dist[q]+cost[q][r])

This formula states that distance vertex r, which is adjacent to vertex q, will be updated if and only if the value of dist[q]+cost[q][r] is less than dist[r]. Here-

* dist is a 1-D array which, at every step, keeps track of the shortest distance from source vertex to all other vertices, and
* cost is a 2-D array, representing the cost adjacency matrix for the graph
* This formula uses both Greedy and Dynamic approaches. The Greedy approach is used for finding the minimum distance value, whereas the Dynamic approach is used for combining the previous solutions (**dist[q]** is already calculated and is used to calculate **dist[r]**)

**Algorithm-**

**Input Data-**

* Cost Adjacency Matrix for Graph G, say cost
* Source vertex, say s

**Output Data-**

* Spanning tree having shortest path from s to all other vertices in G

**implementation-**

Following is the C++ implementation for Dijkstra’s Algorithm

**Note :**   
The algorithm can be mapped to any programming language as per the requirement.

#include<iostream>

using namespace std;

#define V 5 //Defines total number of vertices in the graph

#define INFINITY 999

int min\_Dist(int dist[], bool visited[])

//This method used to find the vertex with minimum distance and is not yet visited

{

int min=INFINITY,index; //Initialize min with infinity

for(int v=1;v<=V;v++)

{

if(visited[v]==false &&dist[v]<=min)

{

min=dist[v];

index=v;

}

}

return index;

}

void Dijkstra(int cost[V][V],int src) //Method to implement shortest path algorithm

{

int dist[V];

bool visited[V];

for(int i=1;i<=V;i++) //Initialize dist[] and visited[]

{

dist[i]=INFINITY;

visited[i]=false;

}

//Initialize distance of the source vertec to zero

dist[src]=0;

for(int c=2;c<=V;c++)

{

//u is the vertex that is not yet included in visited and is having minimum

int u=min\_Dist(dist,visited); distance

visited[u]=true; //vertex u is now visited

for(int v=1;v<=V;v++)

//Update dist[v] for vertex v which is not yet included in visited[] and

//there is a path from src to v through u that has smaller distance than

// current value of dist[v]

{

if(!visited[v] && cost[u][v] &&dist[u]+cost[u][v]<dist[v])

dist[v]=dist[u]+cost[u][v];

}

}

//will print the vertex with their distance from the source

cout<<"The shortest path "<<src<<" to all the other vertices is: \n";

for(int i=1;i<=V;i++)

{

if(i!=src)

cout<<"source:"<<src<<"\t destination:"<<i<<"\t MinCost is:"<<dist[i]<<"\n";

}

}

int main()

{

int cost[V][V], i,j, s;

cout<<"\n Enter the cost matrix weights";

for(i=1;i<=V;i++) //Indexing ranges from 1 to n

for(j=1;j<=V;j++)

{

cin>>cost[i][j];

//Absence of edge between vertices i and j is represented by INFINITY

if(cost[i][j]==0)

cost[i][j]=INFINITY;

}

cout<<"\n Enter the Source Vertex";

cin>>s;

Dijkstra(cost,s);

return 0;

}

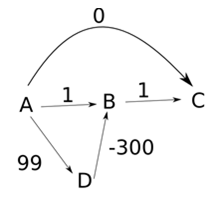
**ime Complexity Analysis-**

Following are the cases for calculating the time complexity of Dijkstra’s Algorithm-

* **Case1**- When graph G is represented using an adjacency matrix -This scenario is implemented in the above **C++** based program. Since the implementation contains two nested for loops, each of complexity **O(n)**, the complexity of Dijkstra’s algorithm is **O(n2)**. Please note that n here refers to total number of vertices in the given graph
* **Case 2**- When graph G is represented using an adjacency list - The time complexity, in this scenario reduces to **O(|E| + |V| log |V|)** where |E|represents number of edges and |V| represents number of vertices in the graph

**Disadvantages of Dijkstra’s Algorithm-**

Dijkstra’s Algorithm cannot obtain correct shortest path(s)with weighted graphs having negative edges. Let’s consider the following example to explain this scenario-

Fig 5: Weighted graph with negative edges

Choosing source vertex as A, the algorithm works as follows-

**Step A**- Initialize the distance array (dist)-

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Set of visited vertices (S) | A | B | C | D |
|  | 0 | ∞ | ∞ | ∞ |

**Step B**- Choose vertex A as dist[A] is minimum and A is not in S. Visit A and add it to S. For all adjacent vertices of A which have not been visited yet (are not in S) i.e C, B and D, update the distance array

dist[C]= min(dist[C], dist[A]+cost(A, C)) = min(∞, 0+0) = 0

dist[B] = min(dist[B], dist[A]+cost(A, B)) = min(∞, 0+1) = 1

dist[D]= min(dist[D], dist[A]+cost(A, D)) = min(∞, 0+99) = 99

**Thus dist[] gets updated as follows-**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Set of visited vertices (S) | A | B | C | D |
| [A] | 0 | 1 | 0 | 99 |

**Step C**- Repeat Step B by

1. Choosing and visiting vertex C since it has not been visited (not in S) and dist[C] is minimum
2. The distance array does not get updated since there are no adjacent vertices of C

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Set of visited vertices (S) | A | B | C | D |
| [A] | 0 | 1 | 0 | 99 |
| [A, C] | 0 | 1 | 0 | 99 |

Continuing on similar lines, Step B gets repeated till all the vertices are visited (added to S). dist[] also gets updated in every iteration, resulting in the following –

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Set of visited vertices (S) | A | B | C | D |
| [A] | 0 | 1 | 0 | 99 |
| [A, C] | 0 | 1 | 0 | 99 |
| [A, C, B] | 0 | 1 | 0 | 99 |
| [A, C, B, D] | 0 | 1 | 0 | 99 |

**Thus, following are the shortest distances from A to B, C and D-**

A->C = 0 A->B = 1 A->D = 99

But these values are not correct, since we can have another path from **A** to **C, A->D->B->C** having **total cost= -200** which is smaller than 0. This happens because once a vertex is visited and is added to the set S, it is never “looked back” again. Thus, Dijkstra’s algorithm does not try to find a shorter path to the vertices which have already been added to S.

* It performs a blind search for finding the shortest path, thus, consuming a lot of time and wasting other resources

**Applications of Dijkstra’s Algorithm-**

* Traffic information systems use Dijkstra’s Algorithm for tracking destinations from a given source location
* **Open Source Path First (OSPF)**, an Internet-based routing protocol, uses Dijkstra’s Algorithm for finding best route from source router to other routers in the network
* It is used by **Telephone and Cellular networks** for routing management
* It is also used by **Geographic Information System (GIS)**, such as Google Maps, for finding shortest path from point A to point B

**introduction to Searching Algorithms**

Not even a single day pass, when we do not have to search for something in our day to day life, car keys, books, pen, mobile charger and what not. Same is the life of a computer, there is so much data stored in it, that whenever a user asks for some data, computer has to search it's memory to look for the data and make it available to the user. And the computer has it's own techniques to search through it's memory fast, which you can learn more about in our [Operating System tutorial](https://www.studytonight.com/operating-system/) series.

What if you have to write a program to search a given number in an array? How will you do it?

Well, to search an element in a given array, there are two popular algorithms available:

1. Linear Search
2. Binary Search

**Linear Search**

Linear search is a very basic and simple search algorithm. In Linear search, we search an element or value in a given array by traversing the array from the starting, till the desired element or value is found.

It compares the element to be searched with all the elements present in the array and when the element is **matched** successfully, it returns the index of the element in the array, else it return -1.

Linear Search is applied on unsorted or unordered lists, when there are fewer elements in a list.

**Features of Linear Search Algorithm**

1. It is used for unsorted and unordered small list of elements.
2. It has a time complexity of **O(n)**, which means the time is linearly dependent on the number of elements, which is not bad, but not that good too.
3. It has a very simple implementation.

We will implement the [Linear Search algorithm](https://www.studytonight.com/data-structures/linear-search-algorithm) in the next tutorial.

**Binary Search**

Binary Search is used with sorted array or list. In binary search, we follow the following steps:

1. We start by comparing the element to be searched with the element in the middle of the list/array.
2. If we get a match, we return the index of the middle element.
3. If we do not get a match, we check whether the element to be searched is less or greater than in value than the middle element.
4. If the element/number to be searched is greater in value than the middle number, then we pick the elements on the right side of the middle element(as the list/array is sorted, hence on the right, we will have all the numbers greater than the middle number), and start again from the step 1.
5. If the element/number to be searched is lesser in value than the middle number, then we pick the elements on the left side of the middle element, and start again from the step 1.

Binary Search is useful when there are large number of elements in an array and they are sorted.

So a necessary condition for Binary search to work is that the list/array should be sorted.

**Features of Binary Search**

1. It is great to search through large sorted arrays.
2. It has a time complexity of **O(log n)** which is a very good time complexity. We will discuss this in details in the Binary Search tutorial.
3. It has a simple implementation.