Greedy Algorithms: Dijkstra’s, Huffman’s

9.3 - 9.4

Intro on greedy algorithms

Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit.

It’s applicable to optimization problems

It doesn't worry whether the current best result will bring the overall optimal result.

Intro on greedy algorithms

On each step—and this is the central point of this technique—the choice made must be:

- feasible, i.e., it has to satisfy the problem’s constraints
- locally optimal, i.e., it has to be the best local choice among all feasible choices available on that step
- irrevocable, i.e., once made, it cannot be changed on subsequent steps of the algorithm

Intro on greedy algorithms

These requirements explain the technique’s name: on each step, it suggests a “greedy” grab of the best alternative available in the hope that a sequence of locally optimal choices will yield a (globally) optimal solution to the entire problem.

Intro on greedy algorithms- Dijkstra’s + Huffman trees

In Section 9.3, we introduce another classic algorithm— Dijkstra’s algorithm for the shortest-path problem in a weighted graph. Section 9.4 is devoted to Huffman trees and their principal application, Huffman codes—an important data compression method that can be interpreted as an application of the greedy technique.

Dijkstra’s algorithm

Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph.

It differs from the minimum spanning tree because the shortest distance between two vertices might not include all the vertices of the graph.

Dijkstra’s algorithm

Dijkstra’s algorithm is very similar to Prim’s algorithm for minimum spanning tree. Like Prim’s MST, we generate a SPT (shortest path tree) with a given source as a root. We maintain two sets, one set contains vertices included in the shortest-path tree, other set includes vertices not yet included in the shortest-path tree. At every step of the algorithm, we find a vertex that is in the other set (set of not yet included) and has a minimum distance from the source.

Dijkstra’s algorithm

Below are the detailed steps used in Dijkstra’s algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.

1) Create a set sptSet (shortest path tree set) that keeps track of vertices included in the shortest-path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.

Dijkstra’s algorithm

Below are the detailed steps used in Dijkstra’s algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.

1) Create a set sptSet (shortest path tree set) that keeps track of vertices included in the shortest-path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.

2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.

Dijkstra’s algorithm

Below are the detailed steps used in Dijkstra’s algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.

1) Create a set sptSet (shortest path tree set) that keeps track of vertices included in the shortest-path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.

2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.

3) While sptSet doesn’t include all vertices

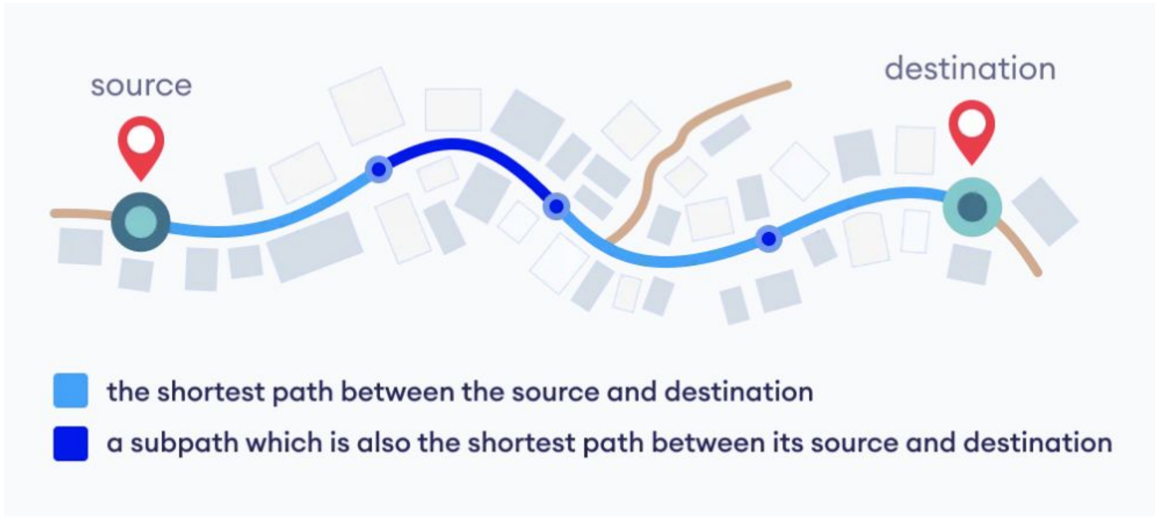
….a) Pick a vertex u which is not there in sptSet and has a minimum distance value.

….b) Include u to sptSet.

….c) Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if the sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

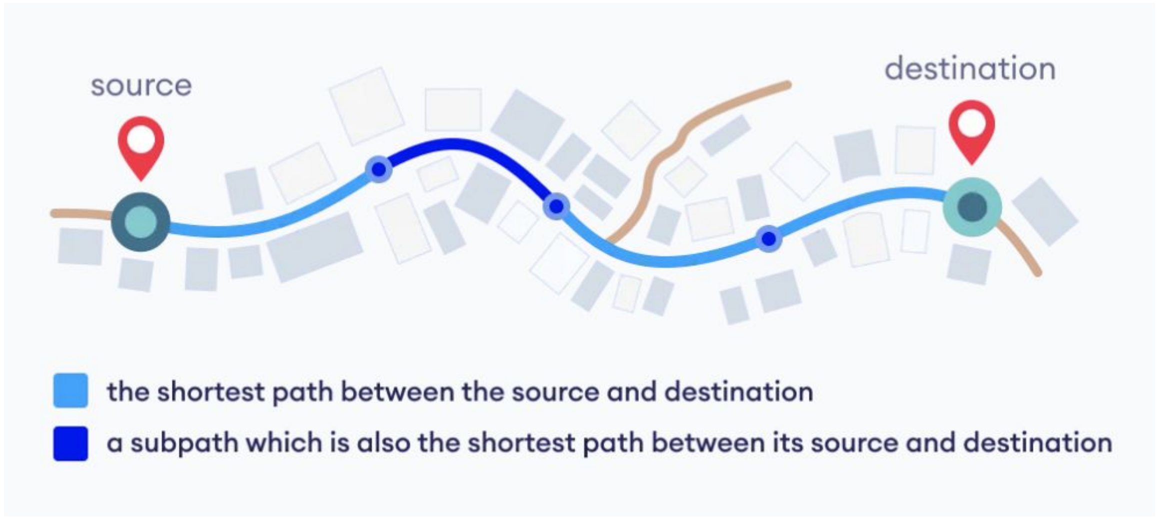
Dijkstra’s algorithm

Dijkstra's Algorithm works on the basis that any subpath B -> D of the shortest path A -> D between vertices A and D is also the shortest path between vertices B and D.


source
destination
🟦 the shortest path between the source and destination
🟦 a subpath which is also the shortest path between its source and destination

Dijkstra’s algorithm

Dijkstra's Algorithm works on the basis that any subpath B -> D of the shortest path A -> D between vertices A and D is also the shortest path between vertices B and D.


source
destination
the shortest path between the source and destination
a subpath which is also the shortest path between its source and destination

Each subpath is the shortest path

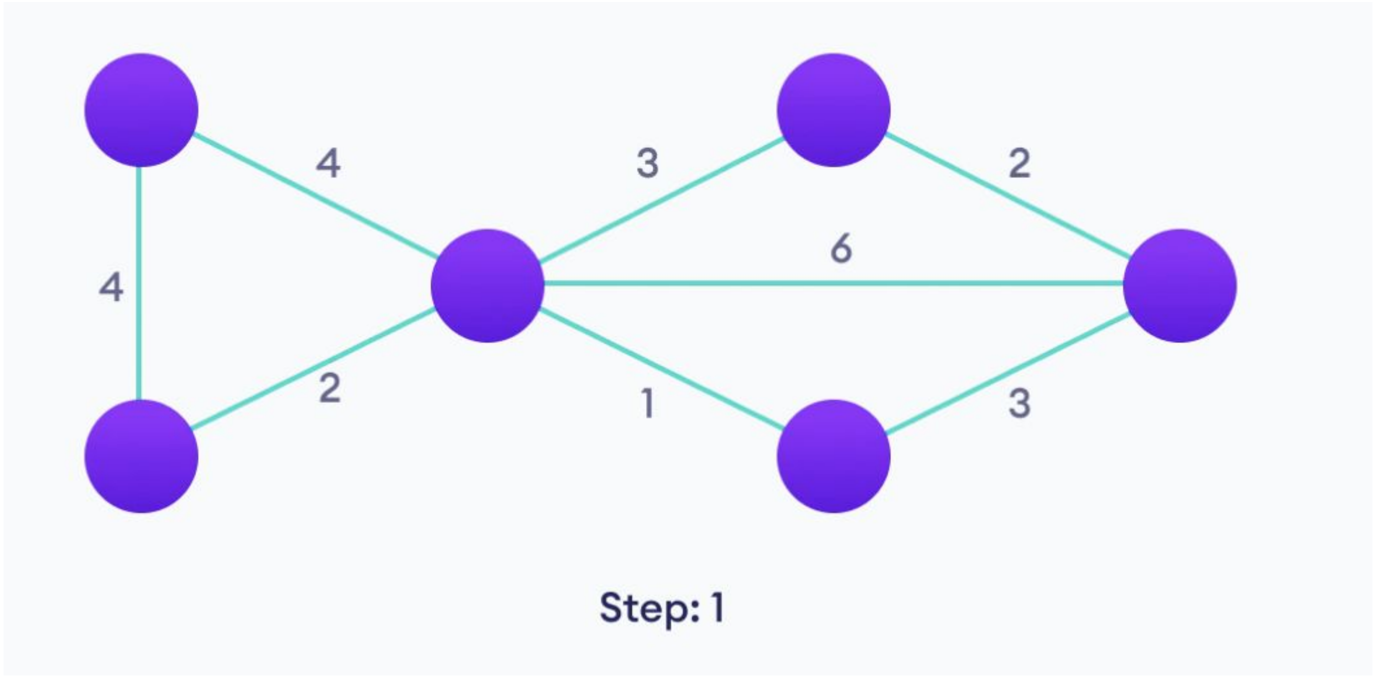
Dijkstra’s algorithm

Djikstra used this property in the opposite direction i.e we overestimate the distance of each vertex from the starting vertex. Then we visit each node and its neighbors to find the shortest subpath to those neighbors.

The algorithm uses a greedy approach in the sense that we find the next best solution hoping that the end result is the best solution for the whole problem.

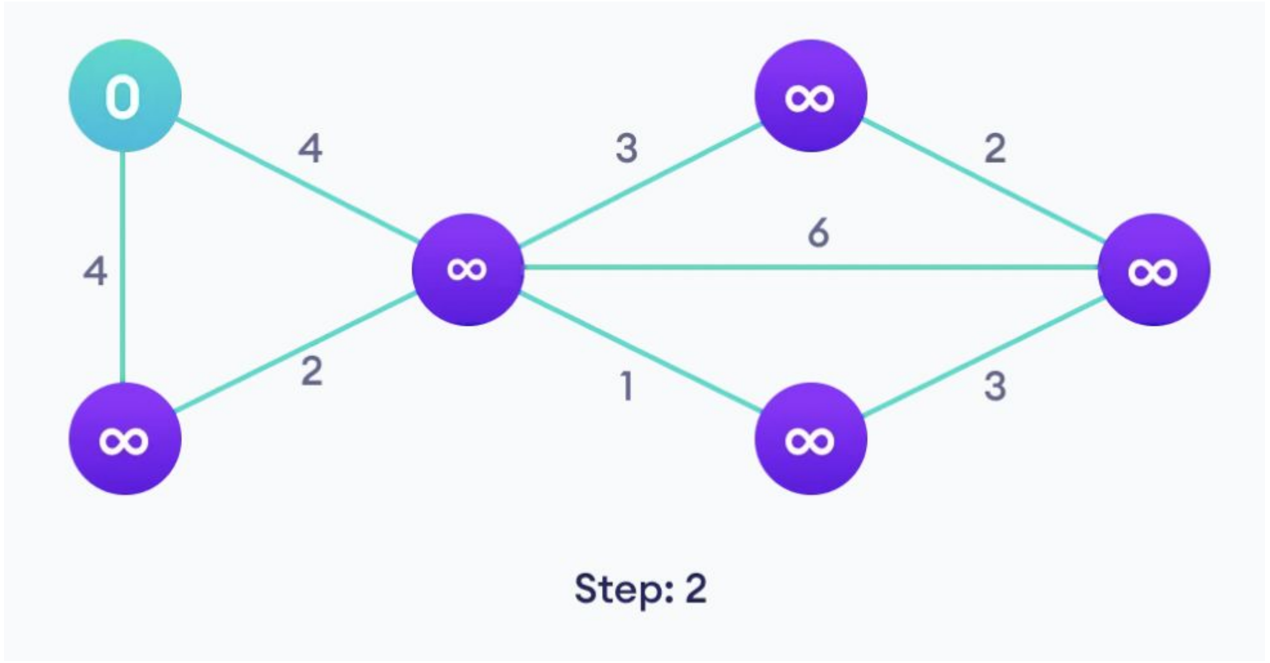
Dijkstra’s algorithm- example

Start with a weighted graph



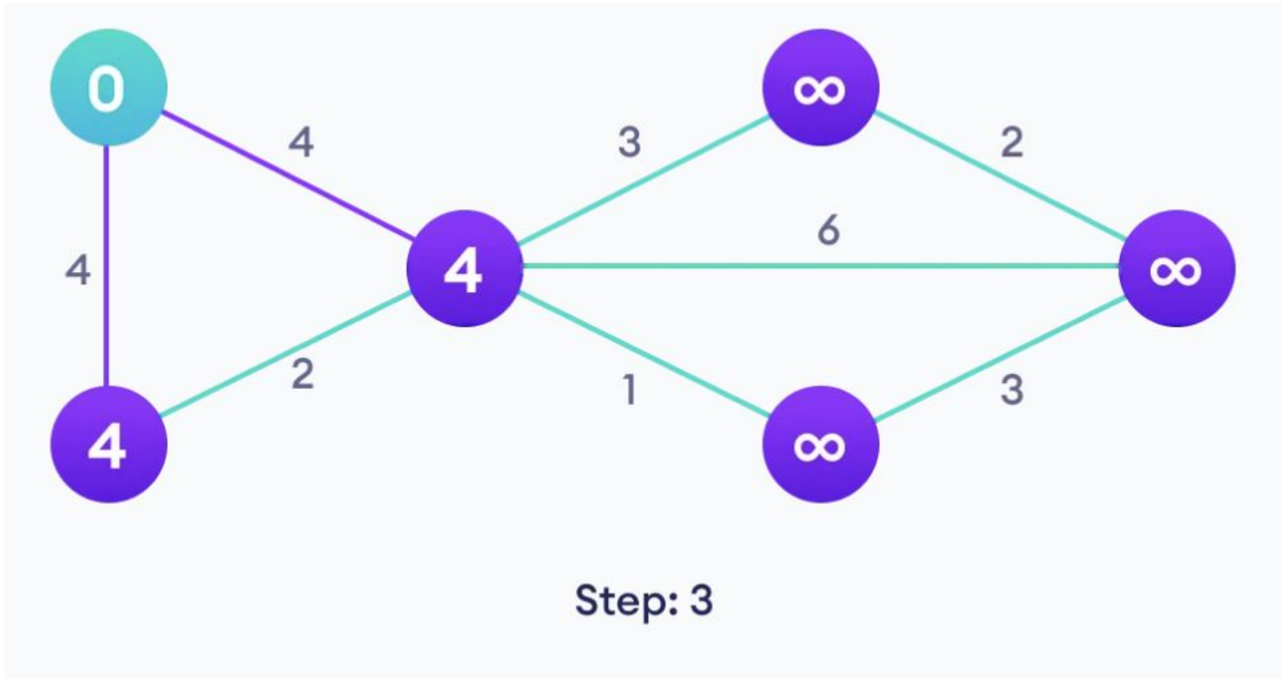
Dijkstra’s algorithm- example

Choose a starting vertex and assign infinity path values to all other devices



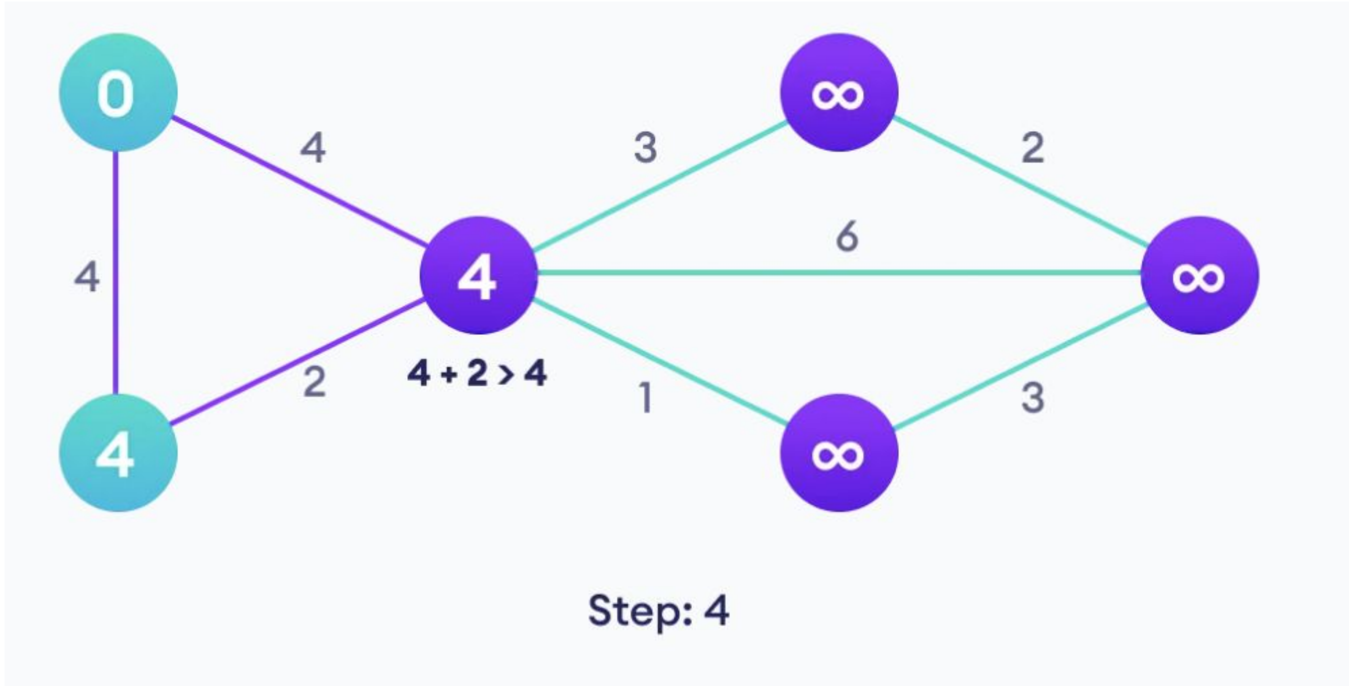
Dijkstra’s algorithm- example

Go to each vertex and update its path length



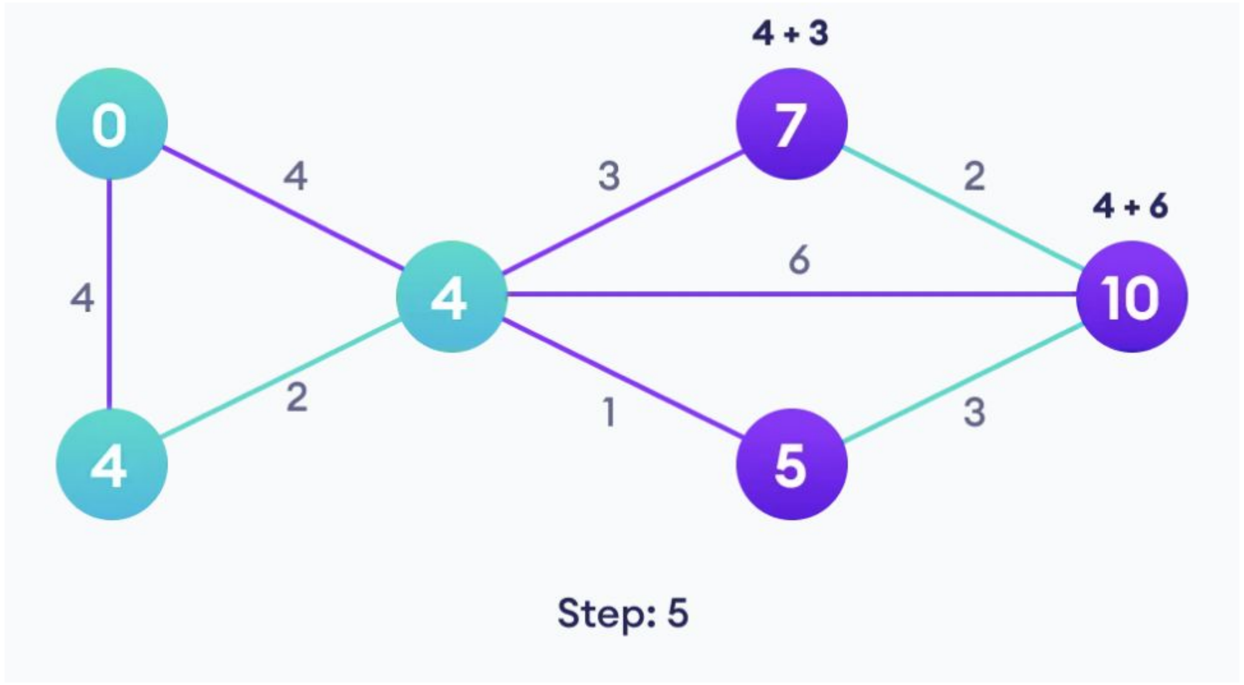
Dijkstra’s algorithm- example

If the path length of the adjacent vertex is lesser than new path length, don't update it



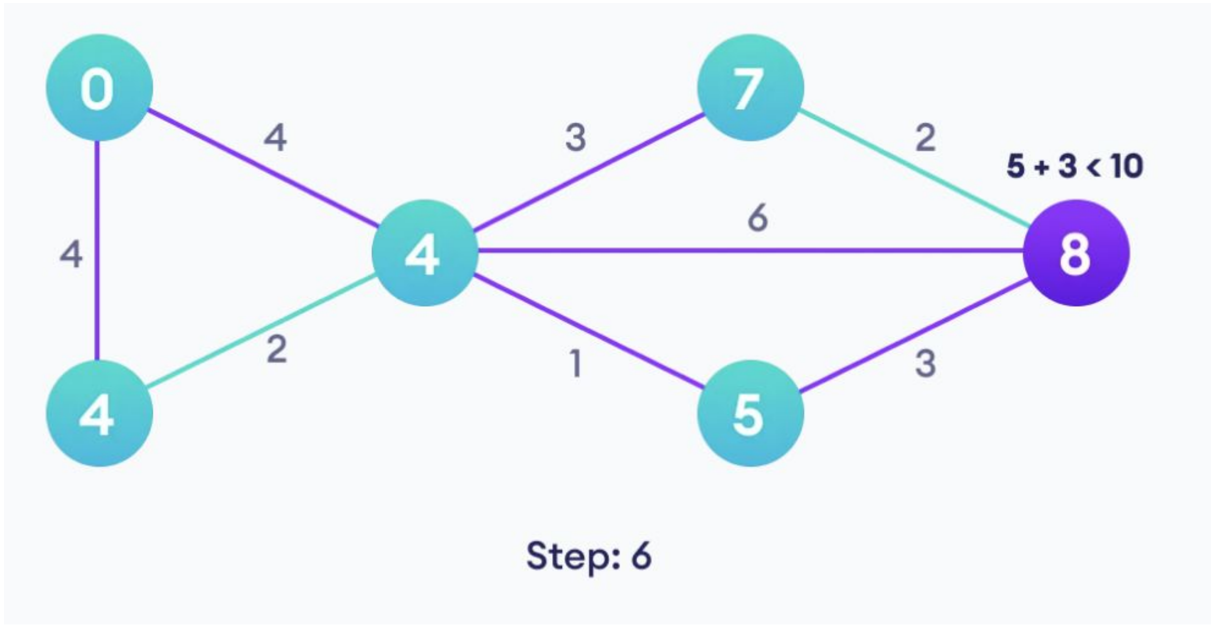
Dijkstra’s algorithm- example

Avoid updating path lengths of already visited vertices



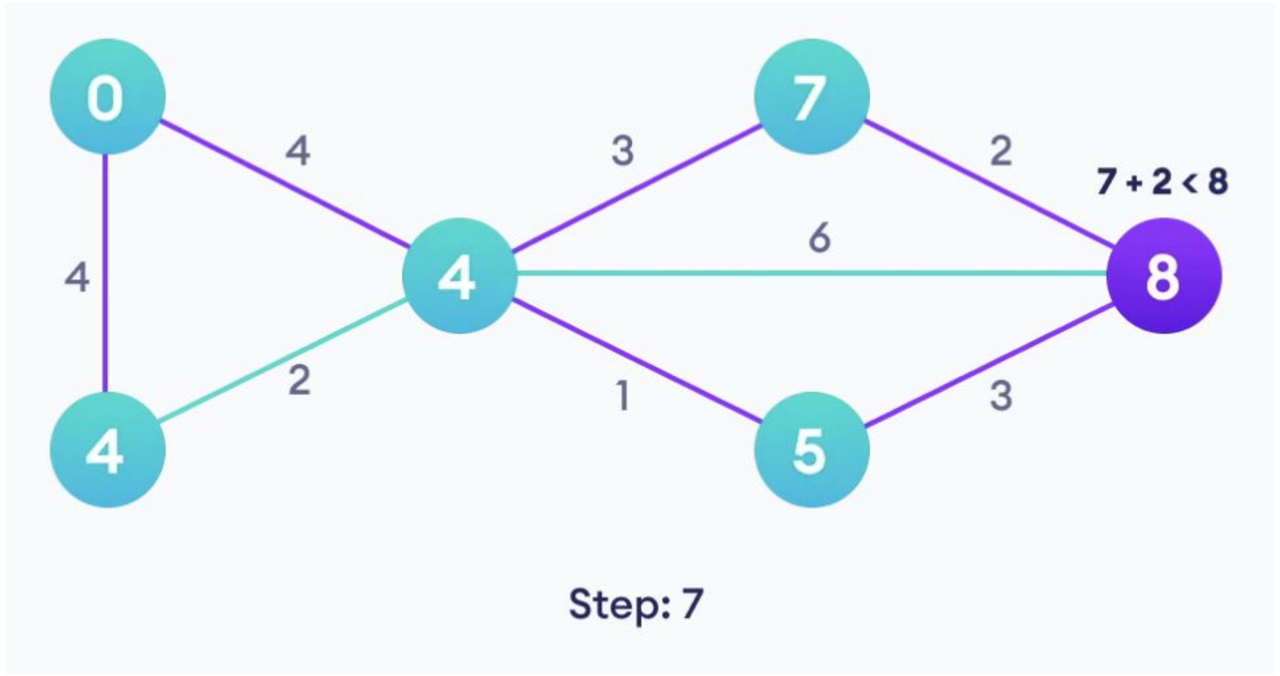
Dijkstra's algorithm- example

After each iteration, we pick the unvisited vertex with the least path length. So we choose 5 before 7



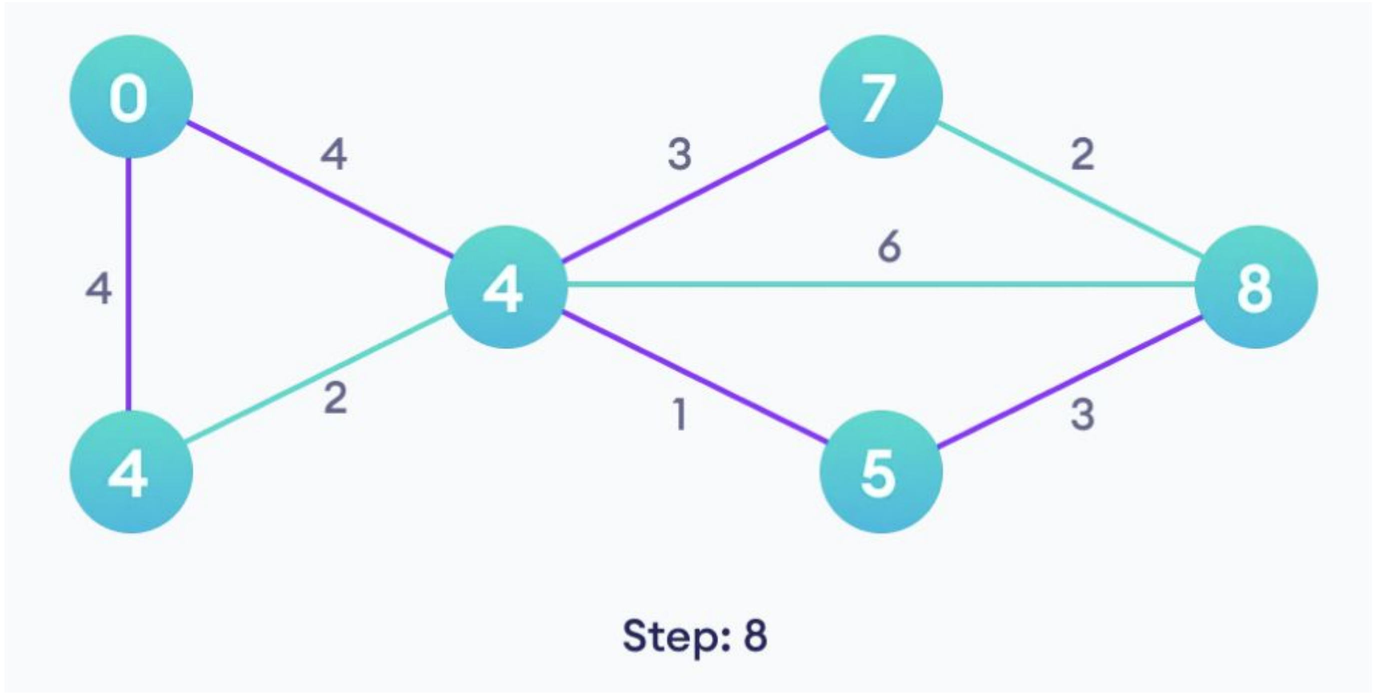
Dijkstra’s algorithm- example

Notice how the rightmost vertex has its path length updated twice



Dijkstra’s algorithm- example

Repeat until all the vertices have been visited



Dijkstra’s algorithm- pseudocode

We need to maintain the path distance of every vertex. We can store that in an array of size v, where v is the number of vertices.

We also want to be able to get the shortest path, not only know the length of the shortest path. For this, we map each vertex to the vertex that last updated its path length.

Dijkstra’s algorithm- pseudocode

Once the algorithm is over, we can backtrack from the destination vertex to the source vertex to find the path.

A minimum priority queue can be used to efficiently receive the vertex with least path distance.

Dijkstra’s algorithm- pseudocode

```
function dijkstra(G, S)
    for each vertex V in G
        distance[V] <- infinite
        previous[V] <- NULL
        If V != S, add V to Priority Queue Q
    distance[S] <- 0
    while Q IS NOT EMPTY
        U <- Extract MIN from Q
        for each unvisited neighbour V of U
            tempDistance <- distance[U] + edge_weight(U, V)
            if tempDistance < distance[V]
                distance[V] <- tempDistance
                previous[V] <- U
      return distance[], previous[]
```

Dijkstra’s algorithm- pseudocode

1. Initialization of all nodes with distance "infinite"; initialization of the starting node with 0
2. Marking of the distance of the starting node as permanent, all other distances as temporarily.
3. Setting of starting node as active.
4. Calculation of the temporary distances of all neighbour nodes of the active node by summing up its distance with the weights of the edges.
5. If such a calculated distance of a node is smaller as the current one, update the distance and set the current node as antecessor. This step is also called update and is Dijkstra's central idea.
6. Setting of the node with the minimal temporary distance as active. Mark its distance as permanent.
7. Repeating of steps 4 to 7 until there aren't any nodes left with a permanent distance, which neighbours still have temporary distances.

Notes on the code

1) The code calculates the shortest distance but doesn’t calculate the path information. We can create a parent array, update the parent array when distance is updated (like prim’s implementation) and use it to show the shortest path from source to different vertices.
2) The code is for undirected graphs, the same Dijkstra function can be used for directed graphs also.
3) The code finds the shortest distances from the source to all vertices. If we are interested only in the shortest distance from the source to a single target, we can break the for loop when the picked minimum distance vertex is equal to the target (Step 3.a of the algorithm).

Dijkstra’s algorithm- video

https://youtu.be/ba4YGd7S-TY

Dijkstra’s algorithm-time complexity

Time Complexity of the code I will send out is O(V^2).

If the input graph is represented using adjacency list, then the time complexity of Prim’s algorithm can be reduced to O(E log V) with the help of binary heap.

Dijkstra’s algorithm- applications

- To find the shortest path
- In social networking applications
- In a telephone network
- To find the locations in the map

Huffman coding

Huffman Coding is a technique of compressing data to reduce its size without losing any of the details. It was first developed by David Huffman.

Huffman Coding is generally useful to compress the data in which there are frequently occurring characters.

Huffman coding

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the largest code.

Huffman coding

The variable-length codes assigned to input characters are Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bitstream.

Huffman coding

Let us understand prefix codes with a counter example. Let there be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1.

This coding leads to ambiguity because code assigned to c is the prefix of codes assigned to a and b. If the compressed bit stream is 0001, the de-compressed output may be “cccd” or “ccb” or “acd” or “ab”.

Huffman coding

There are mainly two major parts in Huffman Coding

1. Build a Huffman Tree from input characters.
2. Traverse the Huffman Tree and assign codes to characters.

Steps to build a Huffman tree

Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.

1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)

Steps to build a Huffman tree

Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.

1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)
2. Extract two nodes with the minimum frequency from the min heap.

Steps to build a Huffman tree

Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.

1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)
2. Extract two nodes with the minimum frequency from the min heap.
3. Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.

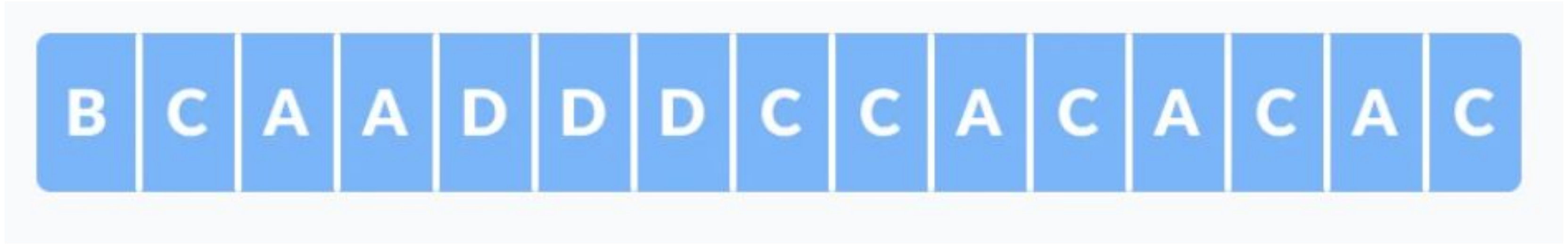
Steps to build a Huffman tree

Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.

1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)
2. Extract two nodes with the minimum frequency from the min heap.
3. Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

How does Huffman coding work?

Suppose the string below is to be sent over a network.



How does Huffman coding work?

Each character occupies 8 bits. There are a total of 15 characters in the above string. Thus, a total of 8 * 15 = 120 bits are required to send this string.

Using the Huffman Coding technique, we can compress the string to a smaller size.

Huffman coding first creates a tree using the frequencies of the character and then generates code for each character.

Once the data is encoded, it has to be decoded. Decoding is done using the same tree.

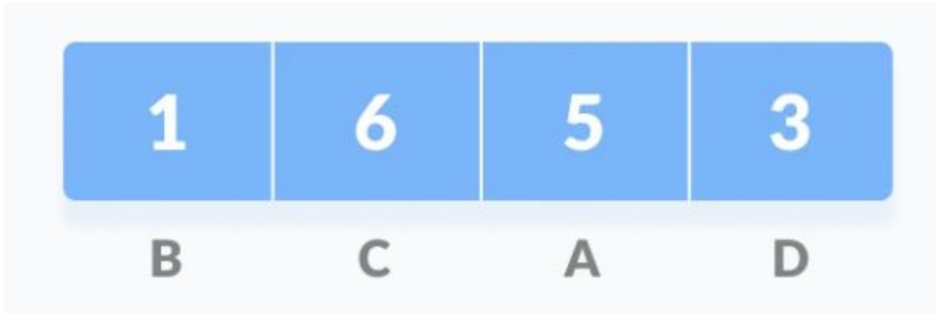
How does Huffman coding work?

Huffman Coding prevents any ambiguity in the decoding process using the concept of prefix code ie. a code associated with a character should not be present in the prefix of any other code.

How does Huffman coding work?

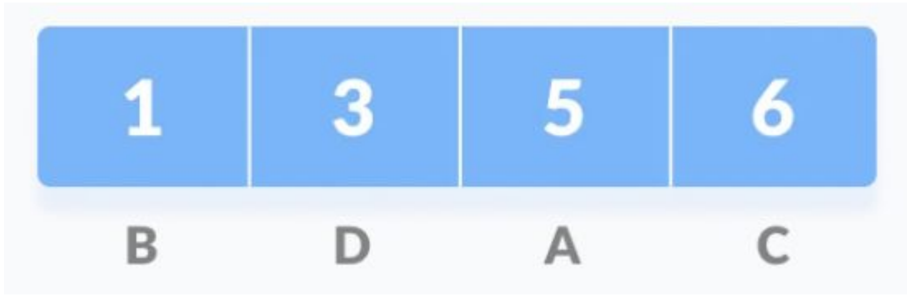
Huffman coding is done with the help of the following steps.

1. Calculate the frequency of each character in the string.



How does Huffman coding work?

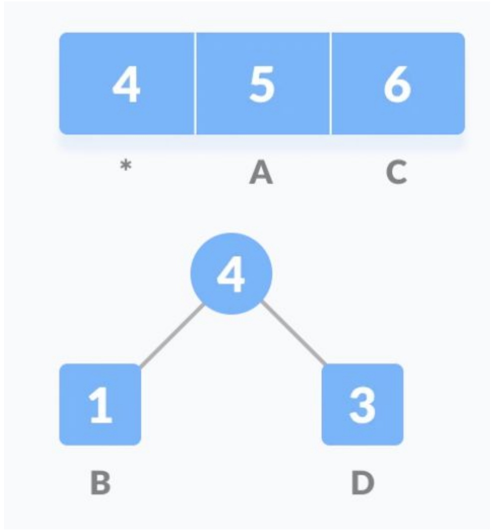
2. Sort the characters in increasing order of the frequency. These are stored in a priority queue Q.



How does Huffman coding work?

3. Make each unique character as a leaf node.

4. Create an empty node z. Assign the minimum frequency to the left child of z and assign the second minimum frequency to the right child of z. Set the value of the z as the sum of the above two minimum frequencies.



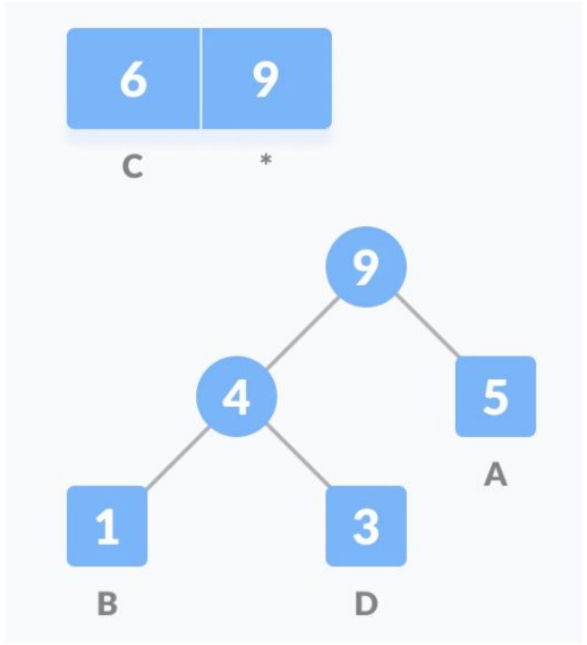
Getting the sum of the least numbers

How does Huffman coding work?

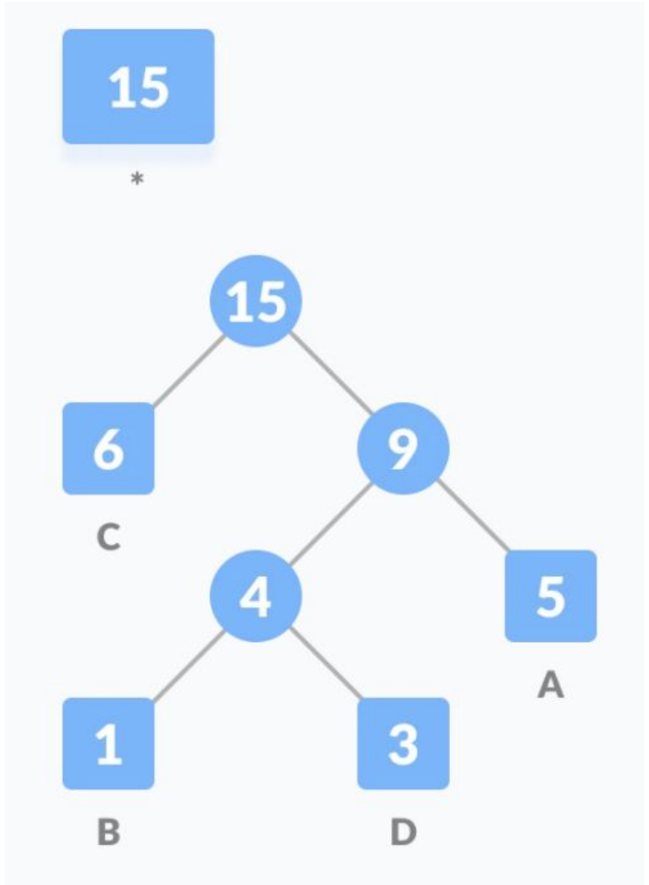
5. Remove these two minimum frequencies from Q and add the sum into the list of frequencies (* denote the internal nodes in the figure above).

6. Insert node z into the tree.

7. Repeat steps 3 to 5 for all the characters.

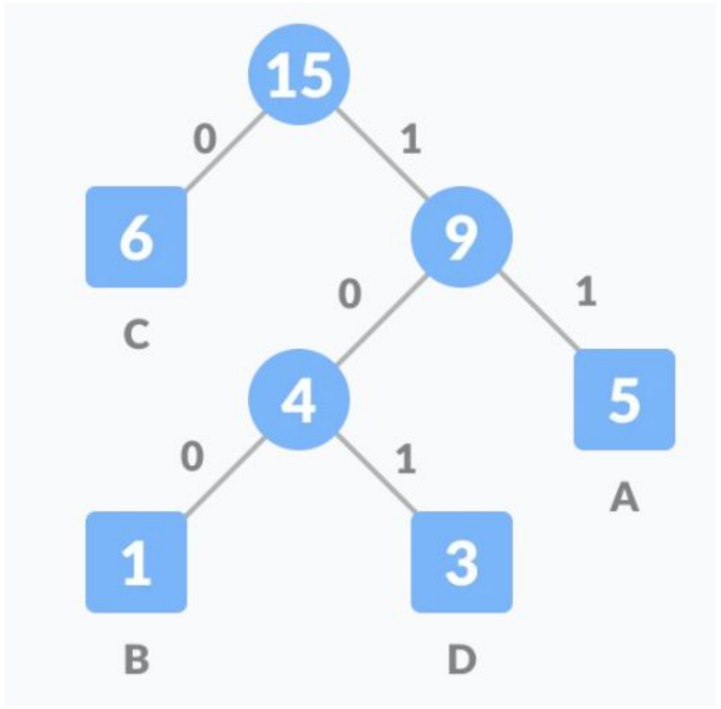


How does Huffman coding work?



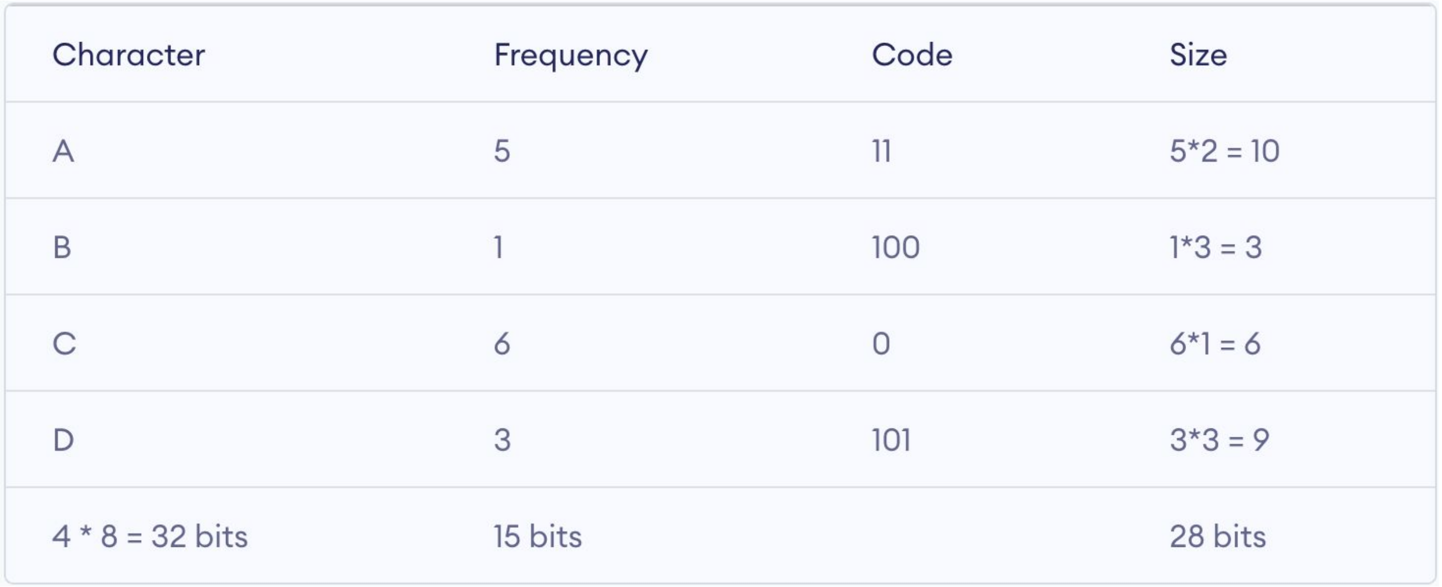
How does Huffman coding work?

8. For each non-leaf node, assign 0 to the left edge and 1 to the right edge.



How does Huffman coding work?

For sending the above string over a network, we have to send the tree as well as the above compressed-code. The total size is given by the table below.



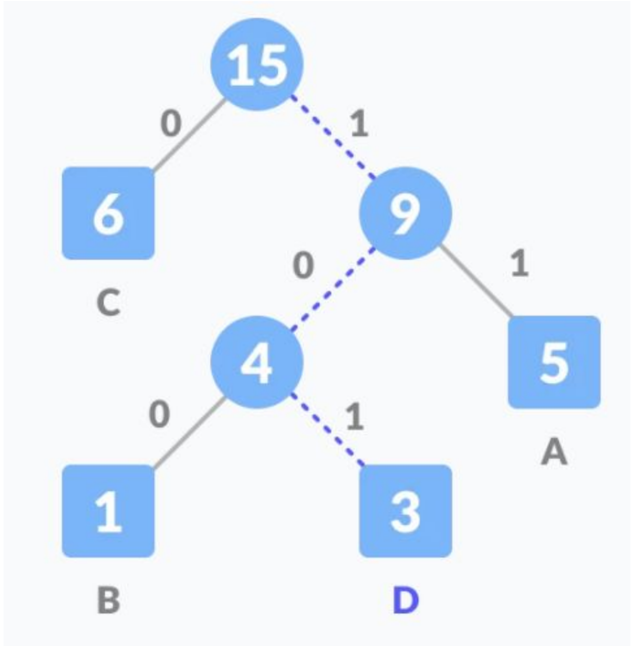
Character	Frequency	Code	Size
A	5	11	5*2 = 10
B	1	100	1*3 = 3
C	6	0	6*1 = 6
D	3	101	3*3 = 9
4 * 8 = 32 bits	15 bits		28 bits

How does Huffman coding work?

Without encoding, the total size of the string was 120 bits. After encoding the size is reduced to 32 + 15 + 28 = 75.

How does Huffman coding work? - decoding the code

For decoding the code, we can take the code and traverse through the tree to find the character. Let 101 is to be decoded, we can traverse from the root as in the figure below.



Huffman coding algorithm

```
create a priority queue Q consisting of each unique character.
sort then in ascending order of their frequencies.
for all the unique characters:
    create a newNode
    extract minimum value from Q and assign it to leftChild of newNode
    extract minimum value from Q and assign it to rightChild of newNode
    calculate the sum of these two minimum values and assign it to the value of newNode
    insert this newNode into the tree
    return rootNode
```

Huffman coding- video

https://www.youtube.com/watch?v=dM6us854Jk0

Huffman algorithm- time complexity

The time complexity for encoding each unique character based on its frequency is O(n log n).

Extracting minimum frequency from the priority queue takes place 2*(n-1) times and its complexity is O(log n). Thus the overall complexity is O(n log n).

Huffman algorithm- applications

- Huffman coding is used in conventional compression formats like GZIP, BZIP2, PKZIP, etc.
- For text and fax transmissions.

Summary

- Dijkstra’s algorithm solves the single-source shortest-path problem of finding shortest paths from a given vertex (the source) to all the other vertices of a weighted graph or digraph. It works as Prim’s algorithm but compares path lengths rather than edge lengths. Dijkstra’s algorithm always yields a correct solution for a graph with nonnegative weights.
- A Huffman tree is a binary tree that minimizes the weighted path length from the root to the leaves of predefined weights. The most important application of Huffman trees is Huffman codes.
- A Huffman code is an optimal prefix-free variable-length encoding scheme that assigns bit strings to symbols based on their frequencies in a given text. This is accomplished by a greedy construction of a binary tree whose leaves represent the alphabet symbols and whose edges are labeled with 0’s and 1’s.

Homework problems!!

Please complete exercise 9.3: #2

Please complete exercises 9.4: #1