Assignment 2

Algorithm Lab (CS2271)

Introduction

This Report is Made By:

Abhiroop Mukherjee	(510519109)
Hritick Sharma	(510519114)
Sayak Rana	(510519108)

The Source codes for all the problems are in this link

Task Scheduling

The Problem

You are given n unit-time tasks, with their respective deadlines and penalty.

$$d = \{d_1, d_2, \dots, d_n\}$$

$$w = \{w_1, w_2, \dots, w_n\}$$

We need to find a task schedule with minimum late task penalty.

Solution 1: Brute Force

As there are n tasks, we can test all possible cases of schedules and take the one as our answer, which gives the minimum late task penalty.

For n task, there are 2^n possible combinations.

We test each of them, see if that schedule is possible in reality or not, if possible, we calculate the late task penalty.

We then select the combination with minimum late task penalty and report it as our answer.

Implementation Detail

For generating all the combination, we used n digit binary numbers to represent the permutation of a schedule.

“Let $N_t(A)$ denote the number of tasks is schedule A, whose deadline is t or earlier.”

“Clearly, if $N_t(A) > t$, for some t, then there is no way to make a schedule as there are more than t tasks to finish before time t.”

We use the above two ideas to see if a schedule combination is possible in reality or not.

Expected Time Complexity

As we are trying out all the possible combination, time complexity of this algorithm will be at least 2^n.

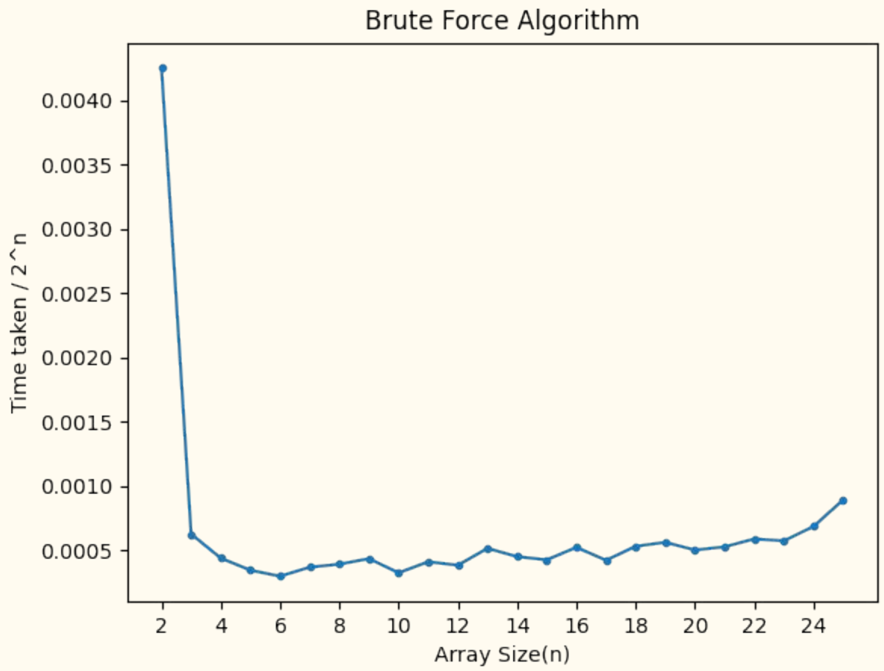
Checking if schedule is possible in reality can be done in O(n)

Hence the expected Time Complexity is $O(2^n)$.

Observed Time Complexity

With the exception of the first observation, we see that as n grows the ratio (time/2^n) remains constant.

This signifies that our Observed Time Complexity is also $O(2^n)$.



Solution 2: Greedy

We can incorporate a greedy strategy to this problem, by linking this problem to well known Graphic Matroid problem.

We first sort the task list in terms of decreasing penalty, then we keep adding tasks from start to finish, while checking if included task makes the schedule possible in real life or not.

In the end we get a schedule with minimum late task penalty.

Implementation Detail

We used inbuilt quicksort method to sort the task set according to penalty.

We used the similar “$N_t(A)$” concept as we used in Brute Force method to check if inclusion of a task makes the schedule possible or not.

Expected Time Complexity

Quicksort will take an average time complexity O(nlgn)

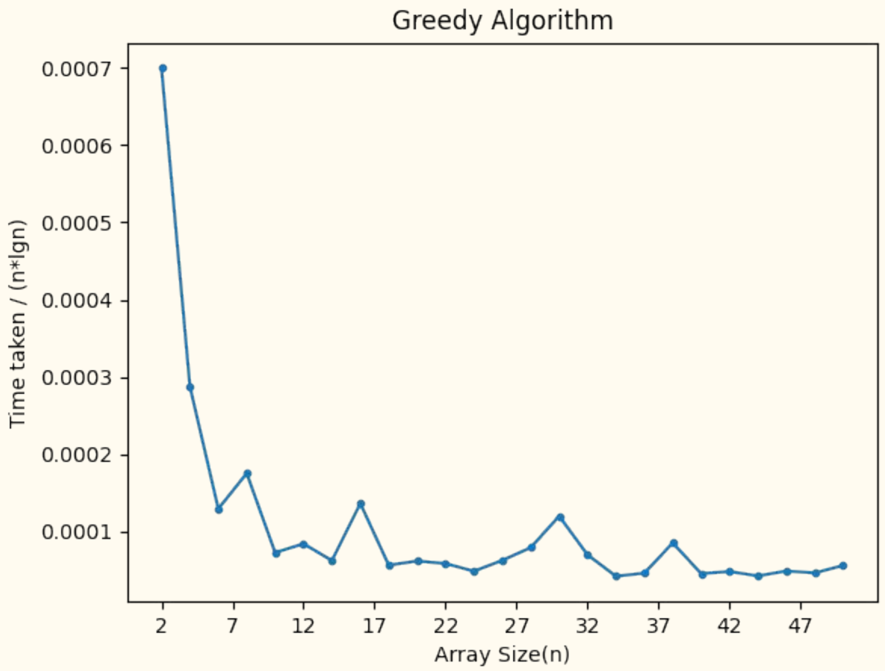
After that only one pass of taskset is required to make a schedule with minimum late task penalty

So Expected Time Complexity is O(nlgn)

Observed Time Complexity

With the exception of the first few observation, we see that as n grows the ratio (time/nlgn) remains constant.

This signifies that our Observed Time Complexity is also O(nlgn).



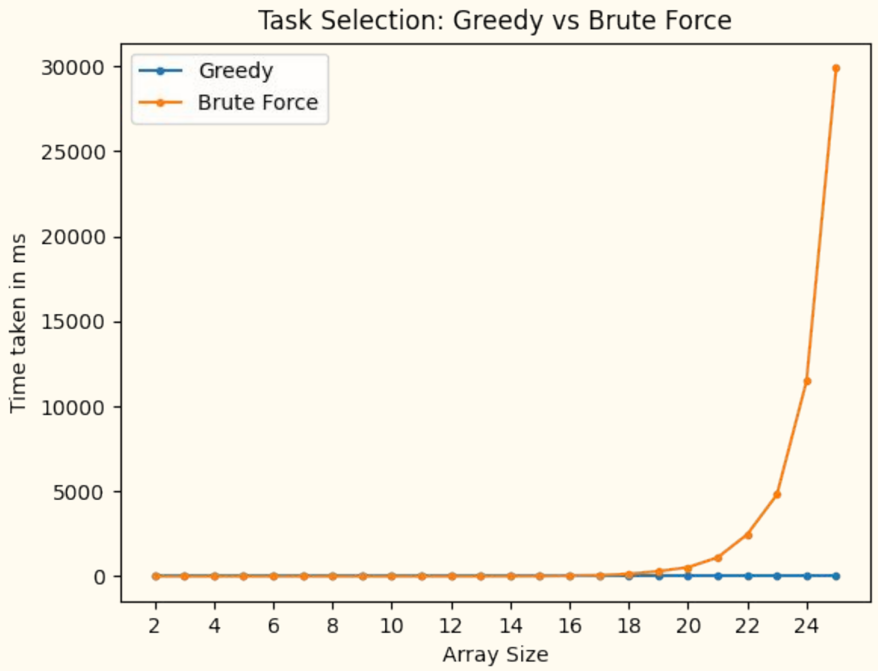
Greedy vs Brute Force

Time Complexity of the Algorithms was as follows

- Brute Force: $O(2^n)$
- Greedy: $O(n\lg n)$

This means that with increasing n, time taken by Brute Force will be significantly higher than Greedy Algorithm.

Exactly same thing is being observed in our experiments.



Matrix Chain Multiplication

The Problem

You are given n matrices which are to be multiplied, along with their dimensions.

Dimensions = {a,b,c,....□,ɣ}

where matrix multiplication is A_{a*b} * B_{b*c} * ... $N_{□*ɣ}$

We need to find a way to multiply this chain with minimum number of steps.

Simple Example

Consider $A_{2*3} * B_{3*5} * C_{5*2}$

Two ways to multiply

1. (A*B)*C : $2*3*5 + 2*5*2 = 50$ operation
2. A*(B*C) : $3*5*2 + 2*3*2 = 42$ operation

We need the output to be the second way

Solution 1: Brute Force

We try out all the way of parenthesizations and report the one with least amount of operations.

The total number of parenthesizations for a n chain matrix comes out to be a special number called Catalan Number, whose value is

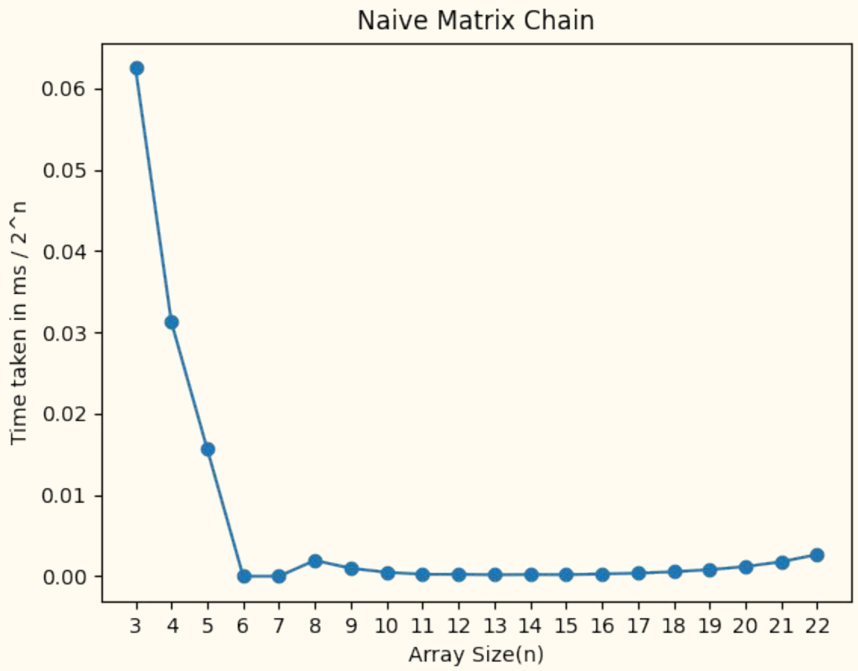
$$C(i) = \frac{1}{i+1} \cdot \binom{2i}{i}$$

Using Stirling Approximation, we get our expected time complexity as O(2^n)

Observed Time Complexity

With the exception of the first few observation, we see that as n grows the ratio (time/2^n) remains constant.

This signifies that our Observed Time Complexity is also $O(2^n)$.



Solution 2: Dynamic Programming

We know from previous method that to get end result, we multiply two of the sub matrices.

Moreover for the end result to have minimum steps, we also need the sub matrices to have minimum steps.

So in this method, we move from bottom to top, saving minimum steps multiplication along the way and using those only to craft the next step multiplication.

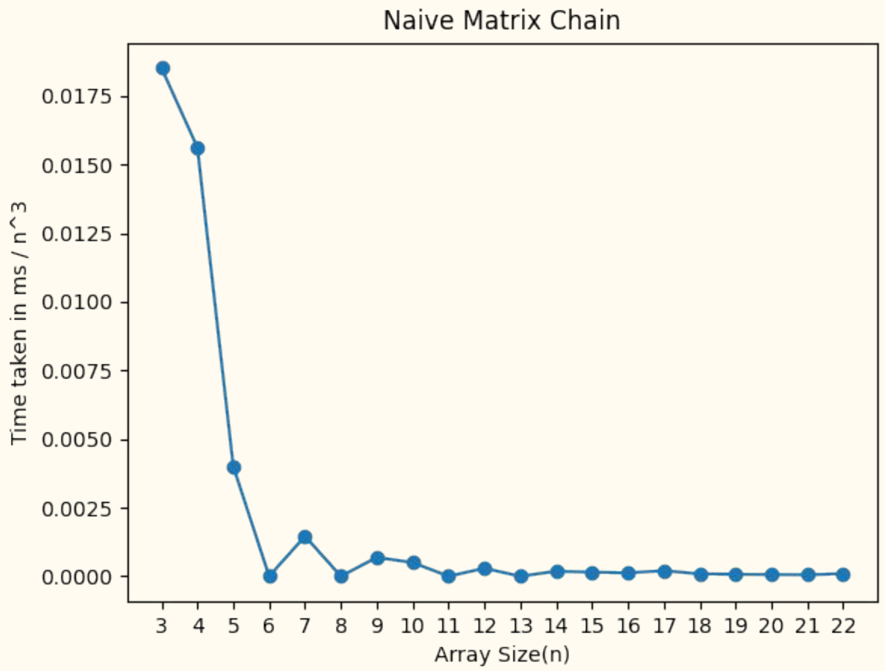
We use the n-table method to facilitate this operation.

This Algorithm has expected Time Complexity of $O(n^3)$.

Observed Time Complexity

With the exception of the first few observation, we see that as n grows the ratio (time/n^3) remains constant.

This signifies that our Observed Time Complexity is also $O(n^3)$.



Huffman Coding

Introduction

A Huffman code is a particular type of optimal prefix code that is commonly used for lossless data compression. The process of finding or using such a code proceeds by means of Huffman coding, an algorithm developed by David A. Huffman while he was a Sc.D . student at MIT, and published in the 1952 paper "A Method for the Construction of Minimum-Redundancy Codes"

Theory

Huffman coding is an efficient method of compressing data without losing information. In computer science, information is encoded as bits—1's and 0's. Strings of bits encode the information that tells a computer which instructions to carry out. Video games, photographs, movies, and more are encoded as strings of bits in a computer. Computers execute billions of instructions per second, and a single video game can be billions of bits of data. It is easy to see why efficient and unambiguous information encoding is a topic of interest in computer science.

Data Compression

Huffman coding Variable length encoding based on frequency of occurrence of the symbols Collapse two least occurring symbols into compound symbol Continue the process until two symbols are left Heap based construction yields the coding tree Minimum overhead on average code word length ensured by collapsing of least probable symbols No other code that uses any other strategy is capable of better compression.

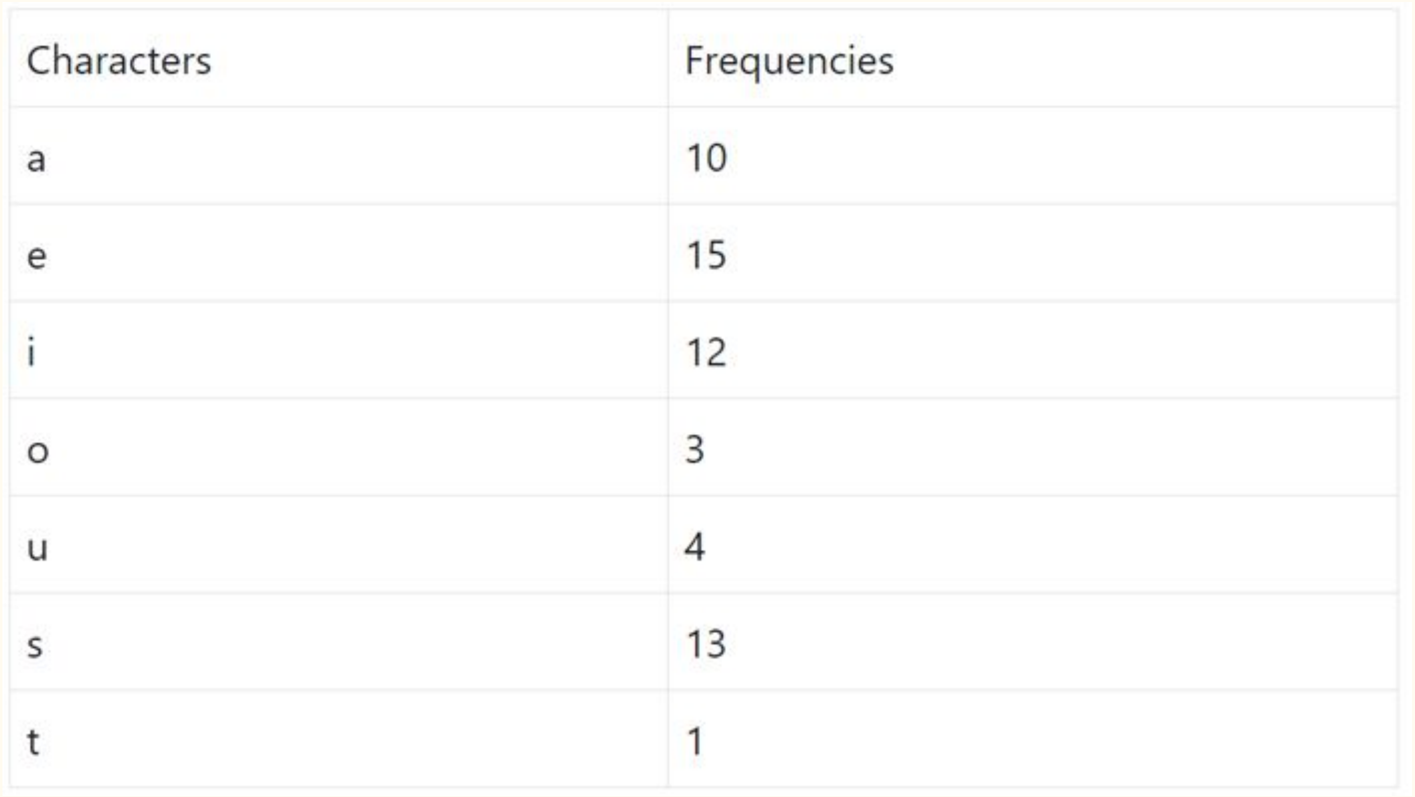
Prefix Rule

Huffman Coding implements a rule known as a prefix rule.

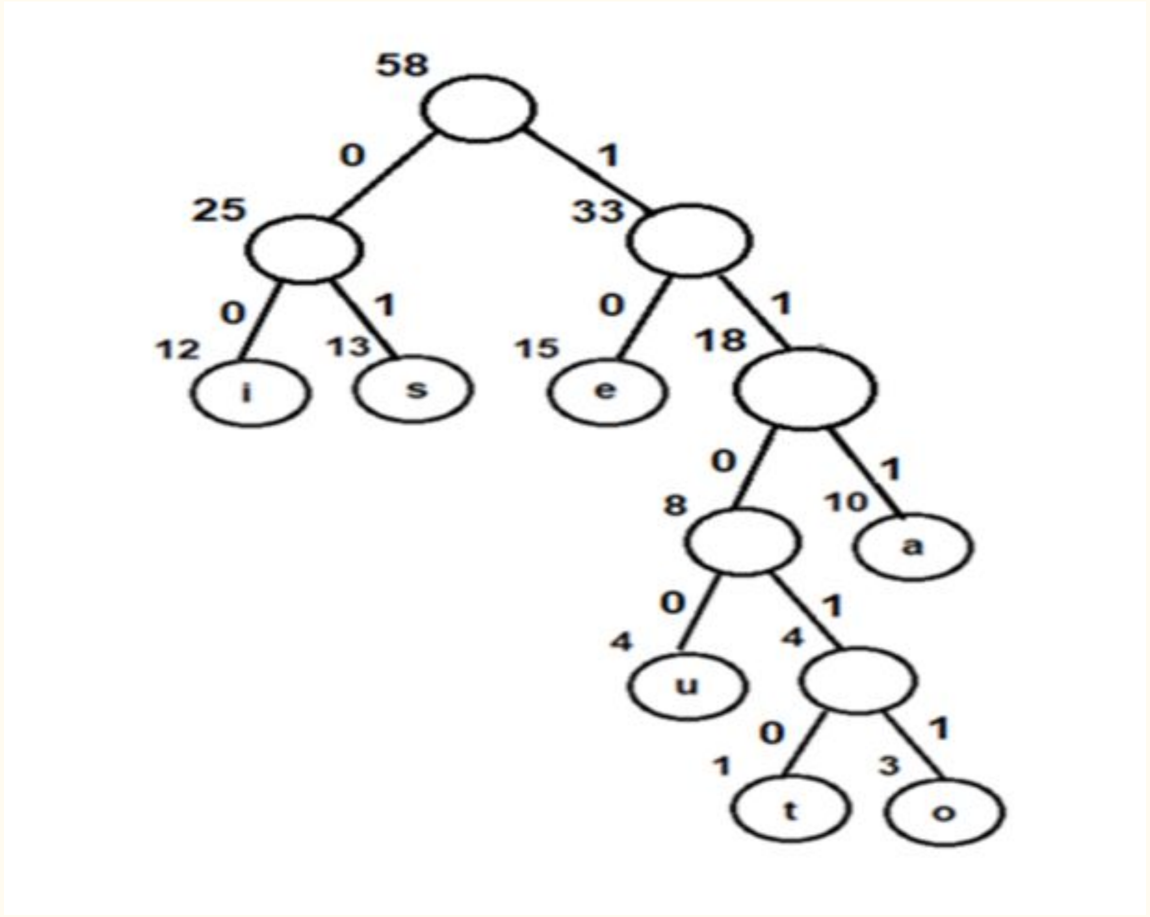
This is to prevent the ambiguities while decoding.

It ensures that the code assigned to any character is not a prefix of the code assigned to any other character.

Example

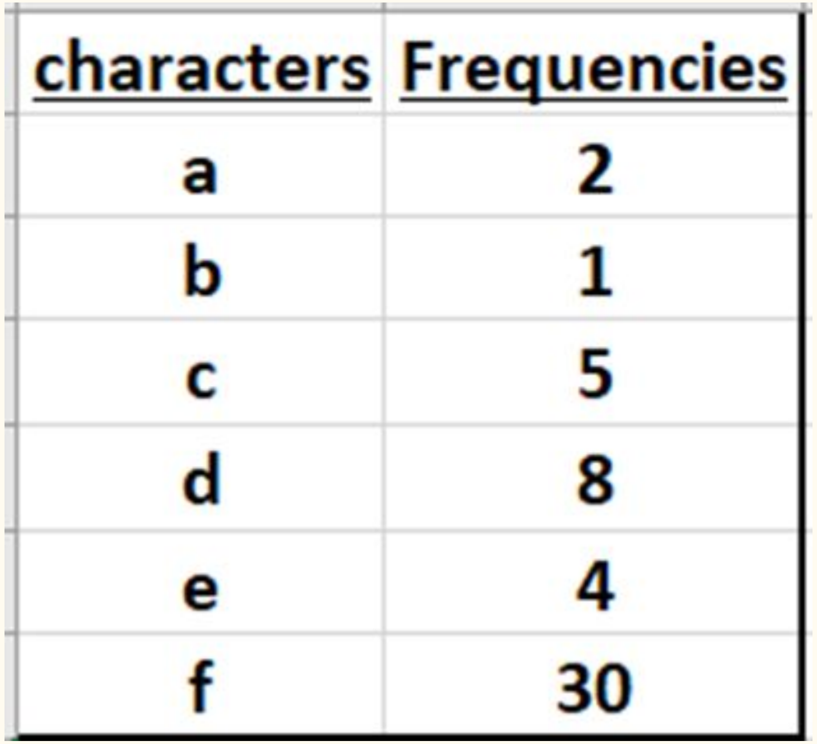


Characters	Frequencies
a	10
e	15
i	12
o	3
u	4
s	13
t	1



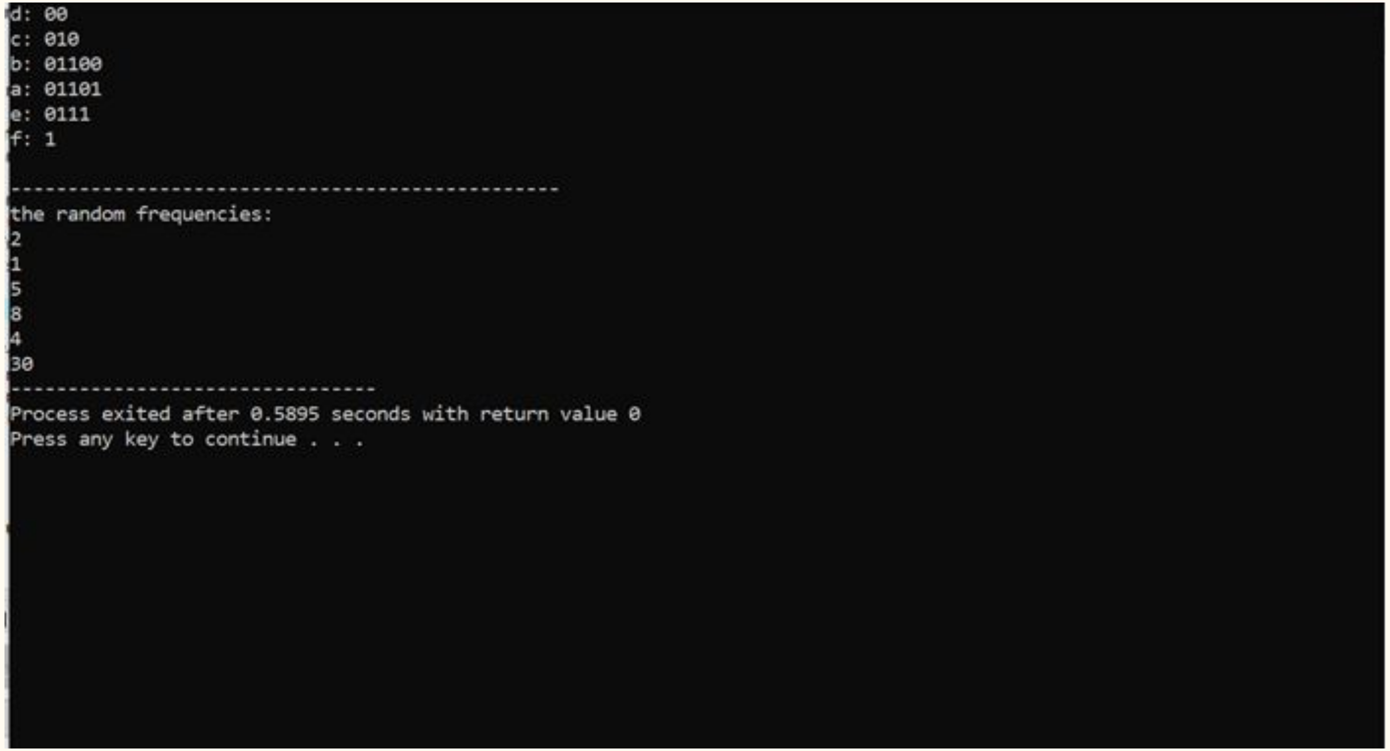
This picture of the Huffman tree is the output for the above character.......

The Example Considered in code



characters	Frequencies
a	2
b	1
c	5
d	8
e	4
f	30

Output



```
d: 00
c: 010
b: 01100
a: 01101
e: 0111
f: 1

------------------------------------------------
the random frequencies:
2
1
5
8
4
30
--------------------------------
Process exited after 0.5895 seconds with return value 0
Press any key to continue . . .
```

Disjoint Set

The Problem

You are given n vertices and m predefined edges in a graph

You have to find a subset of m edges so that the n vertices are divided into disjoint sets.

Simple Example

Consider 7 vertices A,B,..G and given edges (A,B), (B,C), (D,E), (E,F), (A,C), (C,G)

We solve it by following these steps

1. A B C D E F G (all the vertices are initialized as disjoint)
2. Use the edges to keep connecting the sets, unless it makes cycles
   a. (A,B): A-B C D E F G
   b. (B,C): A-B-C D E F G
   c. (D,E): A-B-C D-E F G
   d. (E,F): A-B-C D-E-F G
   e. (A,C) will make a cycle so not taken
   f. (C,G): A-B-C-G D-E-F (final answer)

Implementation Detail

We have implemented this same algorithm using two types of Data Structure

1. Linked List
2. Tree Based

We have also tested this in four datasets:

1. KONECT Western USA Road Map
2. SNAP Facebook Social Circle
3. SNAP Epinion Social Circle
4. SNAP Live Journal Social Network

More About Datasets

All the datasets used have a similar format (all .txt)

- Dataset had commented lines in the start which had number of vertices and edges of the dataset
- After that every line was a set of two numbers, which defined an edge

Using these two properties it was easy to derive a way to extract data from the dataset

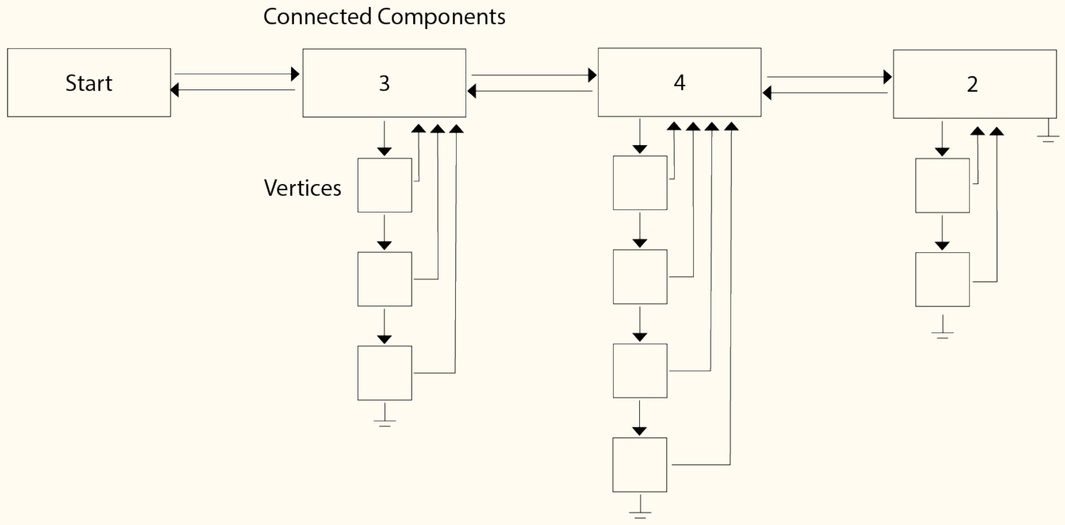
Linked List Implementation

Here we used Two Linked Lists, One for dynamically managing number of Connected Components of the graph (also called Representative Element), and other to maintain the list of vertices inside that connected component

The Connected Component list is doubly linked while the vertices list in singly linked.

Every Representative Node also has a length parameter, which stores the length of vertex linked list inside it.

Every vertex also has a representative pointer which points to its Connected Component Node



Implementation of Make_Set, Find_Set and Union

Make_Set makes a new Representative node with a given vertex (an int) and adds this node to the existing Double Linked List

Find_Set goes through all the Representative Nodes, checking all the node's value and if found, returns it's representative element.

Union takes two representative nodes as input, looks at their length, and shifts the lower length's vertex list to the higher length's vertex list, while updating the length and the representative elements of the higher length vertex list, then it deletes the lower length representative vertex.

Expected Time Complexity: O(lgn) where n is the number of Unions required, i.e more or less equal to number of edges.

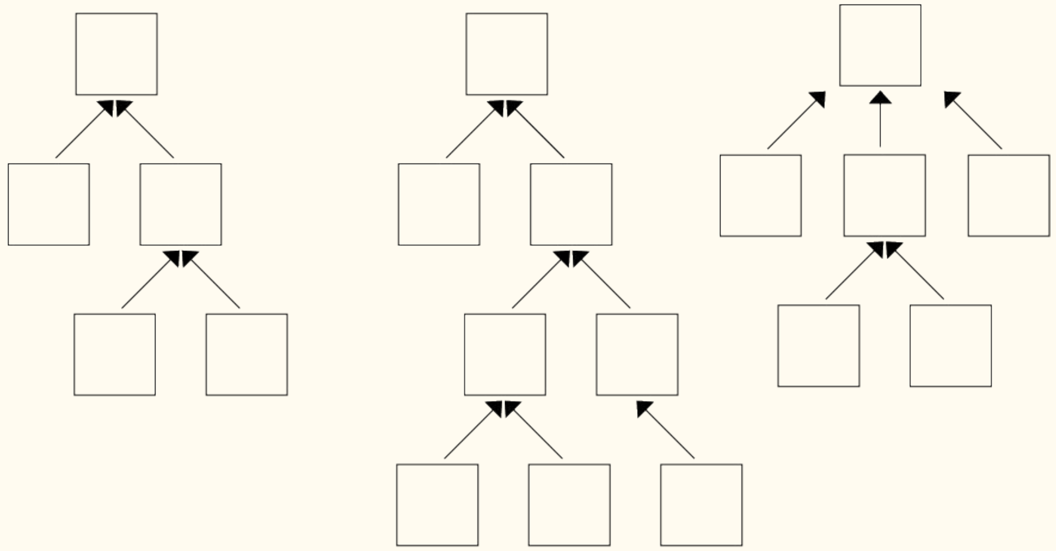
Disjoint Tree Implementation

Here we have only vertex as nodes, which have two extra data other than its value

1. Rank
2. Parent Pointer

Rank of a node signifies how big the subtree with the node as root is.

Parent pointer of a node points to one of the nodes which are the parents of the node. Preferable to the node with highest node in the tree (the main root)



Implementation of Make_Set, Find_Set and Union

Make_Set initializes the vertex node with it's respective data, rank as 1, and its parent pointer pointing to itself

Find_Set of a vertex recursively updates the parent pointers of all the predecessors of the node, and then return the parent of the node, which will be the head root node of the tree.

Union of two nodes will make the lower rank node child of higher rank node. In case the ranks are equal, the parent node will have it's rank incremented by one.

As we know from the datasets the number of nodes, we can directly make a n(no. of vertex) sized array of nodes.

Expected Time Complexity of Tree Based Disjoint Set

Expected Time Complexity: O(lg*n), where lg*n is the number of lg operation required to reduce n to 1.

This is slower than lgn

Eg: $\lg(16) = 4$

$\lg^*(16) = 3$

Eg: $\lg(65536) = 16$

$\lg^*(65536) = 4$

Observations On Datasets

- SNAP Facebook (vertices: 4039, edges: 88234)
    a. Tree Based: 203.307 ms
    b. Linked List Based: 221.124 ms
- SNAP Epinion (vertices: 75888, edges: 508837)
    a. Tree Based: 1257.48 ms (1.25 sec)
    b. Linked List Based: 602489 ms (10 min)
- SNAP Journal (vertices: 4847571, edges: 68993773)
    a. Tree Based: 205968 ms (3 min)
    b. Linked List Based: Worked for approx. 1 hours, to reach 3% completion, expected appx.33 hours
- SNAP USA (vertices: 6262104, edges: 15119284)
    a. Tree Based: 51014 ms (51 sec)
    b. Linked List Based: Worked for approx. 1 hours, to reach 12% completion, expected appx.8 hours

Observation form Observerved Times

1. Tree based Disjoint Set is a lot faster than Linked List based Disjoint Set.
2. Time taken depends number of edges, while space taken during execution depends on the number of vertices.
3. Real Life Datasets are way too big for a measly laptop to handle

Thank You