Test 3

Name: _SAI RAMA KRISHNA TUMMALA____________________

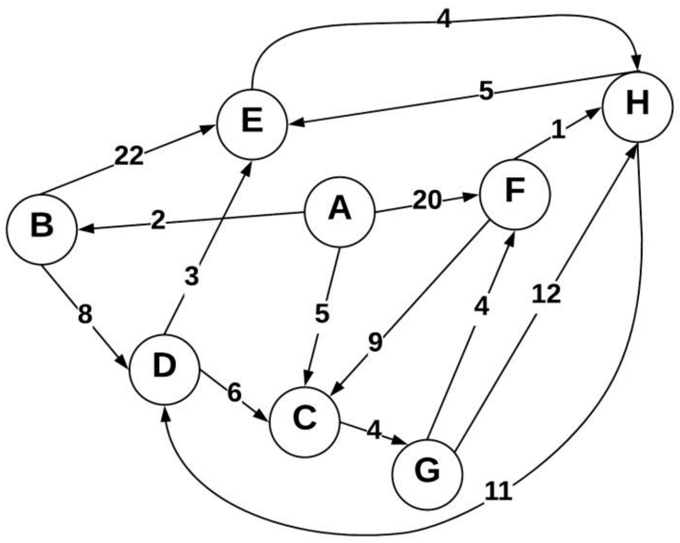
- Everything you turn in must be digitally created.
- No handwriting (except for signature below).
- You must work alone.
- Sharing of answers will result in a 0 on the exam, and possible F in the course.
- Send me your digitally created exam by Friday, May 4th by Midnight on a private slack message.
- Bring your printed signed copy by Monday Morning 10:00 am to my office.

Question	Possible	Score
1	10	
2	10	
3	10	
4	10	
5	10	
6	10	
7	10	
8	10	
9	10	
10	Bonus	
Total:		

By signing this, your saying "I worked alone and did not plagiarize":

____

1) Dijkstra's Algorithm



Use Dijkstra's algorithm to compute the shortest paths from vertex A to every other vertex. Show your work in the space provided below. As the algorithm proceeds, cross out old values and write in new ones, from left to right in each cell. If during your algorithm two unvisited vertices have the same distance, use alphabetical order to determine which one is selected first. Also list the vertices in the order which Dijkstra's algorithm marks them as discovered.

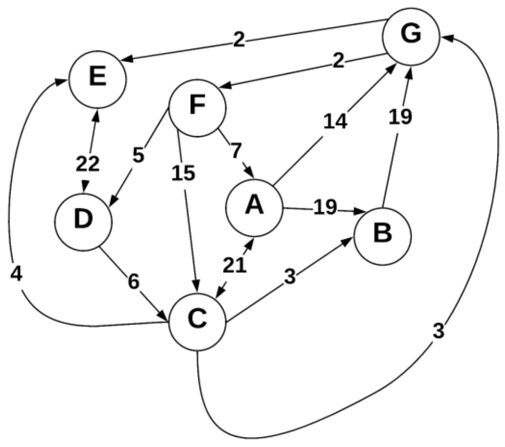
Vertices in Order of Discovery:

A	B	C	G	D	E	F	H	

Vertex	Known	Cost	Previous
A	YES	0	
B	Y	2	A
C	Y	5	A
D	Y	10	B
E	Y	13	D
F	Y	13	G
G	Y	9	C
H	Y	14	F

From vertex F first we have found the path from A and having cost of 20 and then we have found some least cost vertex from G so we updated the cost to 13. At vertex E first we have found cost of 22 from B and then we have changed to D and updated the cost to 13. At vertex H also we have found least cost with 14 so we have updated the previous value to F. The previous cost values at H are 21 and 17 and it updated with 14 and previous visited vertices are G and E.

2) Prims Algorithm



Step through Prim's algorithm to calculate a minimum spanning tree starting from vertex *G.* Show your steps in the table below. As the algorithm proceeds, cross out old values and write in new ones, from left to right in each cell. If during your algorithm two unvisited vertices have the same distance, use alphabetical order to determine which one is selected first. Also list the vertices in the order which Prims algorithm discovers them.

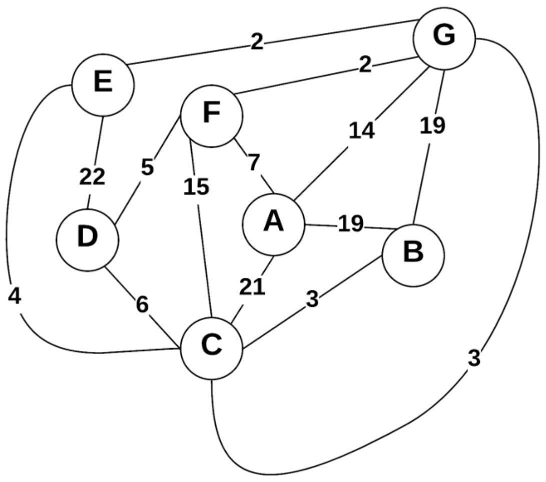
Vertices in Order of Discovery:

G	E	F	D	C	B	A		

- S = Vertices in spanning tree
- U = ! S (vertices not in S)
- Cut = edges going across cut listed alphabetically: (A B) , (C D) , etc.

S (spanning tree)	*U*	Cut (alphabetize)
G	ABCDEF	(GE),(GF)
GE	ABCDF	(ED)
GEF	ABCD	(FA),(FC),(FD)
GEFD	ABC	(DC),(FA)
GEFDC	AB	(CA),(CB),(FA)
GEFDCB	A	(IA)
GEFDCBA	0	(AB),(AG),(CB)

3) Kruskel's Algorithm



Use Kruskal's algorithm to calculate a minimum spanning tree of the graph. Show your steps in the table below, including the disjoint sets at each iteration. If you can select two edges with the same weight, select the edge that would come alphabetically last (e.g., select E—F before B—C. Also, select A—F before A—B).

- Edge Added: put edges added to MST marked as (A B), (E G), etc.
- Edge Cost: weight of edge added
- Running cost is total weight of spanning tree at the point another edge is added.
- Disjoint sets starts as: (A) (B) (C) (D) (E) (F) (G) , and as edges are added => (A) (B C) (D) (E) (F) (G)

Edge Added	Edge Cost	Running Cost	Disjoint Sets
FG	2	2	(A),(B),(C),(D),(E),(FG)
EG	2	4	(A),(B),(C),(D),(EFG)
GC	3	7	(A),(B),(D),(CEFG)
CB	3	10	(A),(D),(BCEFG)
DF	5	15	(A),(BCDEFG)
AF	7	22	(ABCDEFG)

4) Prims Vs Kruskels

Explain why Prim's algorithm is better for dense graphs, while Kruskal's algorithm is better for sparse graphs. What data structures are used to represent each?

Dense graph is a graph in which the number of edges is close to the maximal number of edges.

Sparse graph is a graph in which the number of edges is close to the minimal number of edges.

Prims algorithm has time complexity of O(m+n(logn)) and kruskals has time complexity of O(mlog(n))

Where m is the number of edges and n is number of vertices.

For dense graphs, when m=w(n(log(n)), prims algorithm runs in O(m) which is faster than kruskels which is O(mlog(n)).

For sparse graphs, where m=O(n), we have prims running in O(nlog(n)) while kruskals running in

O(nlog(n)).

Data structure used for kruskals is Disjoint set data structure

Data structure used for prims is Fibonacci heap

5) Greedy Algorithms

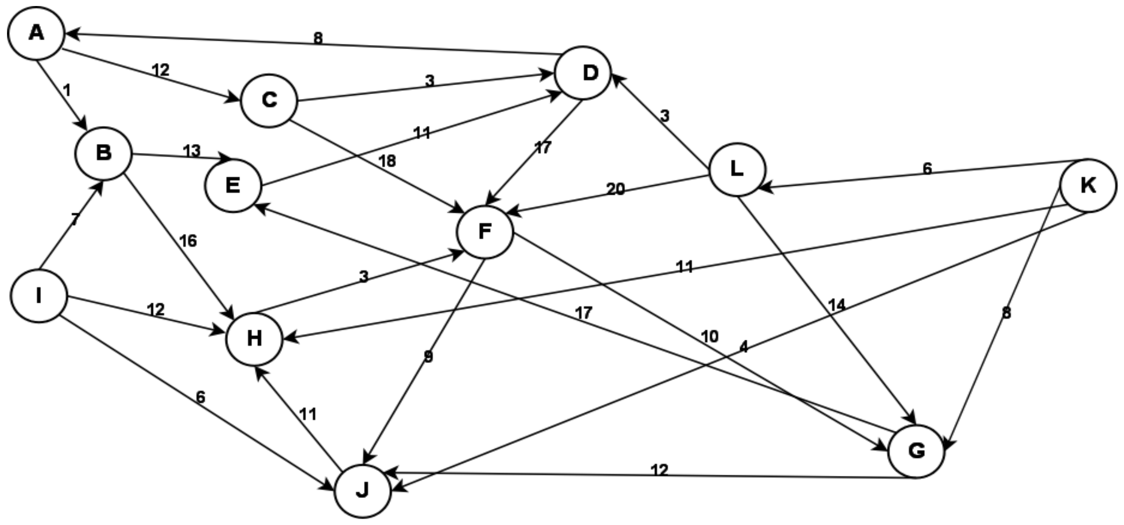
A. Define "Greedy Algorithm"
B. Give an example of a greedy algorithm with explanation of its greediness and performance.
C. Can greedy algorithms produce "optimal" solutions? Short explanation.

A) An algorithm that always takes the best immediate, or local, solution while finding an answer. Greedy algorithms find the overall, or globally, optimal solutions for some optimization problems, but may find less-than-optimal solutions for some instances of other problems.

B) Take the example of coin changing problem we are picking the largest denomination of coin which is not greater than the remaining amount. suppose we are giving having denominations of {3,7,10,20} if we are giving change for 50 in greedy algorithm first we have to take denomination of 20 subtracting from 50 we get 30 and we have to use remaining denominations of 10,10,10 which is not an optimal solution. The optimal solution should be taking two 20's and one 10.

C) NO, greedy algorithms fails to find optimal solution because they usually do not operate exhaustively on all the data. They can make commitments to certain choices too early which prevent them from finding the best overall solution.

6) Graph Traversals



Given the above graph, provide the output of a breadth first and a depth first search. Make choices based on smallest edge weight, then alphabetical to break ties. Start at node A for both.

Depth First:

A	B	E	D	F	J	H	G	C			

Breadth First:

A	B	C	E	H	D	F	J	G			

7) Graph Storage / Manipulation

Given that a weighted directed graph is represented as an adjacency matrix called *adjM*, write a method that reverses all the edges of the graph. That is, for every edge (A , B) in the original graph, there will be an edge (B , A) in the reversed graph with the same weight. Your function should be called *reverse* .

```
/*            In case of Adjacency matrix, we just need to take the transpose of the matrix to reverse
the graph.
              An Adjacency matrix is always a square matrix.
              the number of rows and columns are same in square matrix (adjacency matrix)
*/

#include<iostream>
using namespace std;
void reverse( int b[][10], int n){
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= n; j++)
        {
            cout<<b[i][j]<<"   ";
        }
        cout<<endl;
    }
       }
int main()
{
    int i, j, n;
    int a[10][10] = {0},b[10][10] = {0};
    cout<<"Enter the order of matrix ";
    cin>>n;
    cout<<"Enter the elements\n";
    for (i = 1; i <= n; i++)
    {
        for (j = 1; j <= n; j++)
        {
            cin>>a[i][j];
        }
    }
    for (i = 1; i <= n; i++)
    {
        for (j = 1; j <= n; j++)
        {
            b[j][i] = a[i][j];
        }
    }
    cout<<endl<<"Original Matrix\n";
    for (i = 1; i <= n; i++)
    {
        for (j = 1; j <= n; j++)
        {
            cout<<a[i][j]<<"    ";
        }
        cout<<endl;
    }
    cout<<endl<<"Reverse of adjacency matrix\n";
       reverse(b, n);
}
```

8) Graph Traversal

Write a method that returns whether a graph is a tree. Your method takes a graph *G=(V,E)* as the input and outputs a boolean value. Your method should be called *isTree*().

```
// A C++ Program to check whether a graph is tree or not
#include<iostream>
#include <list>
#include <limits.h>
using namespace std;

// Class for an undirected graph
class Graph
{
	int V; // No. of vertices
	list<int> *adj; // Pointer to an array for adjacency lists
	bool isCyclicUtil(int v, bool visited[], int parent);
public:
	Graph(int V); // Constructor
	void addEdge(int v, int w); // to add an edge to graph
	bool isTree(); // returns true if graph is tree
};

Graph::Graph(int V)
{
	this->V = V;
	adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
	adj[v].push_back(w); // Add w to v’s list.
	adj[w].push_back(v); // Add v to w’s list.
}

// A recursive function that uses visited[] and parent to
// detect cycle in subgraph reachable from vertex v.
bool Graph::isCyclicUtil(int v, bool visited[], int parent)
{
	// Mark the current node as visited
	visited[v] = true;

	// Recur for all the vertices adjacent to this vertex
	list<int>::iterator i;
	for (i = adj[v].begin(); i != adj[v].end(); ++i)
	{
		// If an adjacent is not visited, then recur for
		// that adjacent
		if (!visited[*i])
		{
		if (isCyclicUtil(*i, visited, v))
			return true;
		}

		// If an adjacent is visited and not parent of current
		// vertex, then there is a cycle.
		else if (*i != parent)
		return true;
	}
```

```
        return false;
}

// Returns true if the graph is a tree, else false.
bool Graph::isTree()
{
        // Mark all the vertices as not visited and not part of
        // recursion stack
        bool *visited = new bool[V];
        for (int i = 0; i < V; i++)
              visited[i] = false;

        // The call to isCyclicUtil serves multiple purposes.
        // It returns true if graph reachable from vertex 0
        // is cyclcic. It also marks all vertices reachable
        // from 0.
        if (isCyclicUtil(0, visited, -1))
                     return false;

        // If we find a vertex which is not reachable from 0
        // (not marked by isCyclicUtil(), then we return false
        for (int u = 0; u < V; u++)
              if (!visited[u])
              return false;

        return true;
}

// Driver program to test above functions
int main()
{
        Graph g1(5);
        g1.addEdge(1, 0);
        g1.addEdge(0, 2);
        g1.addEdge(0, 3);
        g1.addEdge(3, 4);
        g1.isTree()? cout << "Graph is Tree\n":
                     cout << "Graph is not Tree\n";

        Graph g2(5);
        g2.addEdge(1, 0);
        g2.addEdge(0, 2);
        g2.addEdge(2, 1);
        g2.addEdge(0, 3);
        g2.addEdge(3, 4);
        g2.isTree()? cout << "Graph is Tree\n":
                     cout << "Graph is not Tree\n";

        return 0;
}
```

9) Huffman Coding

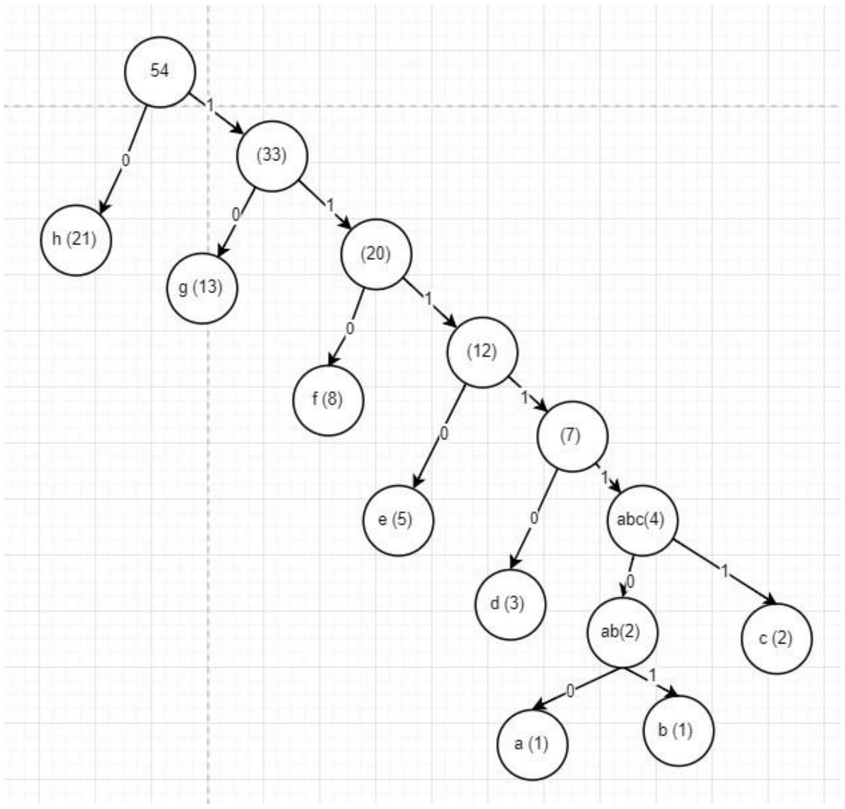
(A) What is an optimal Huffman code for the following set of frequencies, based on the first 8 Fibonacci numbers:

a:1 b:1 c:2 d:3 e:5 f:8 g:13 h:21.

Show your answer as a tree. *Note:* assume that the ordering on the nodes is first by the frequency, and then by the alphabetic order of the node label, so that ab:2 precedes c:2; the node labels are alphabetized too, so that we have a node ab:2 but not ba:2.

(B) Use the code from part (a) to decode the string 1111111111001111101. (As a check: the result should be the name of something that is often yellow.)

A)

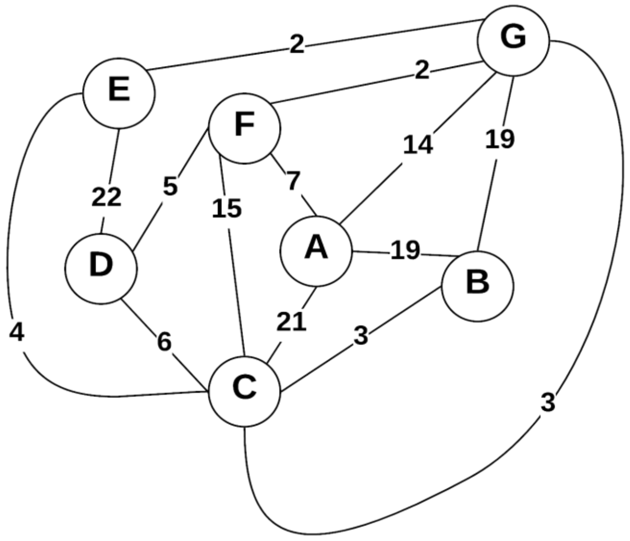


The value of a will be 1111100, b=1111101, c=111111,d=11110,e=1110,f=110,g=10,h=0

B) decoding the string 1111111111001111101

111111 1111100 1111101
c **a** **b**
So the result would be "cab"

Using the graph from question 3, show a Bellman Ford solution.



I have taken A as starting point as there are 7 vertices we can find number of iterations by doing number of vertices – 1.
So there should be 6 iterations.

PI(V) is parent predecessor array.we are starting A and have to find the vertices that will go from A.
If we find any least edge we can replace with edge which is there before.

1st iteration:-

	A	B	C	D	E	F	G
D(V)	**0**	**19**	**12**	**12**	**11**	**7**	**9**
PI(v)		**A**	**G**	**F**	**G**	**A**	**F**

2nd iteration:-

	A	**B**	**C**	**D**	**E**	**F**	**G**
D(V)	**0**	**15**	**12**	**12**	**11**	**7**	**9**
PI(V)		**C**	**G**	**F**	**G**	**A**	**F**

The values will be same for the next 4 iterations also.

Representing them in matrix form

	A	B	C	D	E	F	G
0	0	INF	INF	INF	INF	INF	INF
1	0	19	12	12	11	7	9
2	0	15	12	12	11	7	9
3	0	15	12	12	11	7	9
4	0	15	12	12	11	7	9
5	0	15	12	12	11	7	9
6	0	15	12	12	11	7	9