National Institute of Technology, Calicut
Department of Computer Science and Engineering
CS2094 – Data Structures Lab
Assignment-5

Submission deadline (on or before):
28th March 2016, 10:00:00 PM (for both Main and Advanced batches)

Policies for Submission and Evaluation

You must submit your assignment in the moodle (Eduserver) course page, on or before the submission deadline. Also, ensure that your programs in the assignment must compile and execute without errors in Athena server. During evaluation your uploaded programs will be checked in Athena server only. Failure to execute programs in the assignment without compilation errors may lead to zero marks for that program.

Your submission will also be tested for plagiarism, by automated tools. In case your code fails to pass the test, you will be straight away awarded zero marks for this assignment and considered by the examiner for awarding F grade in the course. Detection of ANY malpractice regarding the lab course will also lead to awarding an F grade.

Naming Conventions for Submission

Submit a single ZIP (.zip) file (do not submit in any other archived formats like .rar or .tar.gz). The name of this file must be ASSG<NUMBER>_<ROLLNO>_<FIRST-NAME>.zip (For example: ASSG5_BxxyyyyCS_LAXMAN.zip). DO NOT add any other files (like temporary files, input files, etc.) except your source code, into the zip archive.

The source codes must be named as ASSG<NUMBER>_<ROLLNO>_<FIRST-NAME>_ <PROGRAM-NUMBER>.<extension> (For example: ASSG5_BxxyyyyCS_ LAXMAN_1.c). If there is a part *a* and a part *b* for a particular question, then name the source files for each part separately as in ASSG5_BxxyyyyCS_LAXMAN_1b.c.

If you do not conform to the above naming conventions, your submission might not be recognized by some automated tools, and hence will lead to a score of 0 for the submission. So, make sure that you follow the naming conventions.

Standard of Conduct

Violations of academic integrity will be severely penalized.

Each student is expected to adhere to high standards of ethical conduct, especially those related to cheating and plagiarism. Any submitted work MUST BE an individual effort. Any academic dishonesty will result in zero marks in the corresponding exam or evaluation and will be reported to the department council for record keeping and for permission to assign F grade in the course. The department policy on academic integrity can be found at:
http://cse.nitc.ac.in/sites/default/files/Academic-Integrity.pdf.

Assignment Questions

General Instructions for all the questions:

- Invalid input should be detected and suitable error messages should be generated.
- Sample inputs are just indicative.

1. **DISJOINT-SET**

   Write a program that implements the disjoint-set data structure using rooted forests. Also, write functions to implement the *ranked union* and *path compression* heuristics on your data structure, and compute the efficiency of the disjoint set data structure **find** operation by applying neither, either or both of the heuristics, by counting the total number of data accesses performed over the course of the program.

   Your program must support the following functions:

   - **makeset(x)** creates a singleton set with element **x**.
   - **find(x)** finds the representative of the set containing the element **x**.
   - **union(x,y)** merges the sets containing elements **x** and **y** into a single set. The representative of the resultant set is assigned with **find(x)**, unless the ranked union heuristic is used and the ranks of both **find(x)** and **find(y)** are different. Otherwise, the representative is assigned in accordance with the ranked union heuristic.

   Note that looking up an element in the data structure must be done in O(1) time.

   Input format:

   The input consists of multiple lines, each one containing a character from {'m', 'f', 'u', 's'} followed by zero, one or two integers. The integer(s), if given, is in the range 0 to 10000.

   - Call the function **makeset(x)** if the input line contains the character 'm' followed by an integer **x**. Print “PRESENT” if **x** is already present in some set, and the value of **x**, otherwise.
   - Call the function **find(x)** if the input line contains the character 'f' followed by an integer **x**. Output the value of **find(x)** if **x** is found, and “NOT FOUND”, otherwise.
   - Call the function **union(x,y)** if the input line contains the character 'u' followed by space separated integers **x** and **y**. Print “ERROR”, without terminating, if either **x** or **y** isn’t present in the disjoint set. Print **find(x)** itself if **find(x)=find(y)**. Otherwise, print the representative of the resultant set. The representative of the resultant set is assigned with **find(x)**, unless the ranked union heuristic is used and the ranks of both **find(x)** and **find(y)** are different. Otherwise, the representative is assigned in accordance with the ranked union heuristic.
   - If the input line contains the character 's', print the number of data accesses performed by the **find** commands by each of the data structures over the course of the program and terminate.

   Output format:

   The output consists of multiple lines of space-separated columns. The columns correspond to the following disjoint-set data structures:

   a. with neither ranked union nor path compression applied.
   b. with only ranked union applied.
   c. with only path compression applied.
   d. with both ranked union and path compression applied.

   Each line in the output contains the output of the corresponding line in the input, after applying to the respective data structures. The final line of the output contains the number of data accesses performed by the **find** commands by each of the data structures over the course of the program.

```
Sample Input       Sample Output
m 1                1
m 2                2
m 3                3
m 4                4
m 5                5
m 6                6
m 7                7
m 8                8
m 9                9
u 1 2              1 1 1 1
u 3 4              3 3 3 3
u 5 6              5 5 5 5
u 7 8              7 7 7 7
u 9 8              9 7 9 7
u 6 8              5 5 5 5
u 4 8              3 5 3 5
u 2 8              1 5 1 5
f 9                1 5 1 5
m 10               10
u 10 9             10 5 10 5
s                  38 32 33 30
```

GRAPH ALGORITHMS

Data structure conventions:

In a graph with **n** vertices, the vertices are labeled from **0** to **n-1**. Use adjacency lists to store the graphs, with the vertices sorted in ascending order. The adjacency list of each node is a singly linked list that contains its adjacent nodes sorted in ascending order from left to right. The nodes in this list contain two fields, namely, the label of the adjacent node and the weight of the edge, if provided. Unless specified otherwise, the adjacency lists must be processed iteratively from left to right.

2. GRAPH TRAVERSALS

Write a program to perform depth-first and breadth-first searching on a directed graph.

Input/output format:

The first line of the input contains a positive integer **n**, the number of vertices in the graph, in the range 1 to 10000.

The subsequent **n** lines contain the labels of the nodes adjacent to the respective nodes, sorted in ascending order from left to right. If a node has no adjacent nodes, then the line corresponding to its adjacency list will be empty.

The rest of the input consists of multiple lines, each one containing a three-letter string followed by zero or two integers. The integers, if given, will be in the range **0** to **n-1**.

- The string “dfs” will be followed by two integers, say **start** and **key**. Perform a depth first search for **key** in the graph by starting a traversal from **start**. Output the nodes visited by the traversal either till the node labeled **key** is found, or all the nodes have been traversed without finding a node with label **key**.
- The string “bfs” will be followed by two integers, say **start** and **key**. Perform a breadth first search for **key** in the graph by starting a traversal from **start**. Output the

nodes visited by the traversal either till the node labeled **key** is found, or all the nodes have been traversed without finding a node with label **key**.

- The string "stp" means terminate the program.

The output, if any, of each command should be printed on a separate line, and the nodes visited during a traversal should be space separated.

Sample Input

```
11

3
6 10
0 7
3 8

5
8
0 4 9
8
6
bfs 1 4
bfs 9 2
dfs 4 8
dfs 1 4
bfs 4 8
dfs 9 2
stp
```

Sample Output

```
1 3 0 7 8 4
9 8 0 4 3 7
4 8
1 3 7 8 9 4
4 3 8
9 8 4 3 7 0
```

3. **SHORTEST PATH**

Write a program that implements Dijkstra's algorithm for computing shortest paths in a directed graph with positive edge weights.

Input/output format:

The first line of the input contains a positive integer **n**, the number of vertices in the graph, in the range 1 to 1000.

The subsequent **n** lines contain the labels of the nodes adjacent to the respective nodes, sorted in ascending order from left to right. If a node has no adjacent nodes, then the line corresponding to its adjacency list will be empty.

The subsequent **n** lines contain the weights of the edges corresponding to the adjacency list. The edge weights are positive real numbers in the range (0, 10000]. If a node has no adjacent nodes, then the line corresponding to its adjacent edge weights will be empty.

The rest of the input consists of multiple lines, each one containing a four-letter string followed by zero, one or two integers. The integers, if given, will be in the range **0** to **n-1**.

- The string “apsp” will be followed by a single integer, the label of the source vertex. Output the shortest path distance from the source vertex to all the **n** vertices in the graph, sorted in the order of their labels, in a space separated format. Print “INF” for nodes that are unreachable from the source vertex.
- The string “sssp” will be followed by two integers, respectively, labels of the source and destination nodes. Output the shortest path from the source node to the destination node, if such a path exists. Print “UNREACHABLE”, otherwise.
- The string “stop” means terminate the program.

The output, if any, of each command should be printed on a separate line.

Sample Input

```
9
1 4
5
3
6

2 7 8
2
4
5 7
2 20
3
7
5

1 6 4
0
2
2 1
apsp 0
sssp 0 6
sssp 0 7
sssp 5 6
sssp 8 7
stop
```

Sample Output

```
0 2 6 13 12 5 18 10 9
18
10
13
1
```

4. **MINIMUM SPANNING TREE**

Write programs that compute the minimum spanning tree of a connected undirected graph using the following algorithms:

a. Kruskal’s algorithm
b. Prim’s algorithm

Input format:

The first line of the input contains a positive integer **n**, the number of vertices in the graph, in the range 1 to 1000.

The subsequent **n** lines contain the labels of the nodes adjacent to the respective nodes, sorted in ascending order from left to right.

The subsequent **n** lines contain the weights of the edges corresponding to the adjacency list. The edge weights are real numbers in the range [-10000, 10000]. Further, no two edges have the same weight.

Output format:

Print the sum of the edge weights of the minimum spanning tree as the output.

Sample Input

```
12
8 9
2 3 4
1 3 5 6
1 2 4
1 3 5 7 8 9
2 4 6
2 5 7 10 11
4 6 8
0 4 7 9
0 4 8
6 11
6 10
27 41
10 11 17
10 7 33 44
11 7 26
17 26 5 8 15 16
33 5 21
44 21 31 18 29
8 31 20
27 15 20 13
41 16 13
18 23
29 23
```

Sample Output

```
164
```

****