Design and Analysis of Algorithms, Assignment 4

As usual, each part of this assignment is submitted as a separate file. As you finish up the parts, you can turn them in at `https://submit.ncsu.edu/`. Before the deadline, you can submit updated parts of any section whenever you want. They'll automatically replace your previous submission.

Be sure you use the right filename for each part of the assignment, including the right capitalization, and don't pack multiple files into an archive for submission. You'll get a small penalty if we have to rename files or unpack them from an archive before we grade them (or just to tell what you've submitted).

For programming problems, you'll need to submit an implementation that compiles and runs using the instructions from the previous assignment. For programs, you'll generally be expected to read input from standard input and write output to standard output. Try to match our expected output format exactly, without printing extra output (like prompts) that the assignment didn't ask for.

1. All-pairs shortest paths (40 points)

   You're going to write a program, `allpairs.c` (or `.cpp`, `.java`, `.py`). This program will use the Floyd-Warshall algorithm to solve the all-pairs shortest paths problem.

   We'll be working with a graph that's defined by a collection of words, each word representing a vertex. Words consist of lower-case letters, and there's an edge between two words if the they have the same length and differ at just one character position. The weight of the edge is how far apart the differing letters are in alphabetic order. So, for example, the words "help" and "yelp" would be connected by an edge of weight 17 (since 'y' is 17 characters away from 'h' in the alphabet). There wouldn't be an edge between "ram" and "arm"; although they contain the same letters, they differ in the first character position and in the second.

   For this problem, you can use a dense graph representation if you want. It won't hurt the storage cost or running time, since the resulting shortest path matrix will be a dense representation.

   Your program will read a list of words and compute the shortest path matrix. To help prove you computed the whole matrix, you'll print a simple statistic based on this matrix. For each word, w_i, compute the number of other words you can reach starting from w_i and traversing edges of the graph (this should require looking at a row of the shortest path matrix). Compute the average number of reachable words, averaged over all words in the list. Print this value rounded to two fractional digits.

   Then, your program will respond to a sequence of queries. Each querry will give a pair of words from the list. In response, your program will print out the length of the shortest path from the first word to the second one, followed by the sequence of words on one such shortest path. If there isn't a path, it will print the two words, followed by " not reachable". See the sample execution below for an example.

   For each query, there may be multiple equally short paths. Your program can print out any one of them.

   If you want, you can use the $O(V^2)$ technique we developed in class for recovering a shortest path. Or, if you'd like 5 points of extra credit, you can implement a linear-time technique instead. To do this, as you fill in the table, keep a record of the value, k that achieves the shortest path between each pair of vertices. Then, when you need to recover a shortest path from some i to j, you can first check to see if there's a direct edge. If not, you can lookup the vertex k that the path goes through. Then, recursively recover a shortest path from i to k and then from k to j.

Input

Input will start with a positive integer n, giving the number of words on the list. This will be followed by n words, one per line. Words will consist only of lower-case letters, but they may be of different lengths.

The word list is followed by a non-negative integer, m, giving the number of queries. This will be followed by m lines, each line containing two words from the list.

For example, the first sample input file, input_ap1.txt contains:

```
9
ran
met
mat
rag
ram
hat
set
tag
rap
2
tag rap
set ran
```

Sample Execution

Once your program is working, you should be able to run it on my sample input files as follows. I'm providing expected output files, but since the shortest paths you compute may not be the same as the ones I compute, I don't expect your output to match these exactly. Here, I'm assuming your implementation is in Java. You'll need to modify the execution slightly if you've used a different language.

```
# Run on the smallest sample input
$ java allpairs < input_ap1.txt
4.56
11 tag rag ran rap
set ran not reachable
# Run on the medium sample input.
$ java allpairs < input_ap2.txt
19.59
33 hand hind find fond food foot
41 work worn torn tort sort soot slot plot plow ploy play
54 zero hero hers hens dens dons cons coos boos boor door dour four
36 plot plop glop goop good food fond find
find door not reachable
```

2. Maximum Flow (40 points)

   This problem will probably be a good bit harder than the previous one. I made it worth the same number of points so it wouldn't hurt too badly if you make some mistakes or if you can't get your solution working. You'll be implementing the Edmonds-Karp algorithm for maximum flow.

Graph Input

Input will come from standard input and will start with a pair of numbers, n and m, giving the number of vertices and a number of **directed** edges in the graph. These two numbers will be followed by m

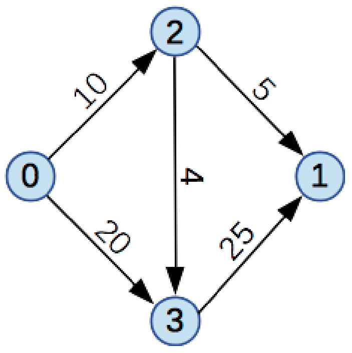
lines, each describing an edge with 3 integer values. The first two values on each line, i j, give the endpoints of the edge (i and j in $0 \ldots n-1$). The third value gives the non-negative integer weight of the edge from i to j. The value of n will always be two or greater. Vertex zero is always the source and vertex one is always the sink. The graph will never contain anti-parallel edges.

Your main programs will be called `maxflow.c`, `maxflow.cpp`, `maxflow.java` or `maxflow.py`.

Sample Input

```
4 5
0 2 10
0 3 20
2 3 4
2 1 5
3 1 25
```

This describes a graph that looks like:



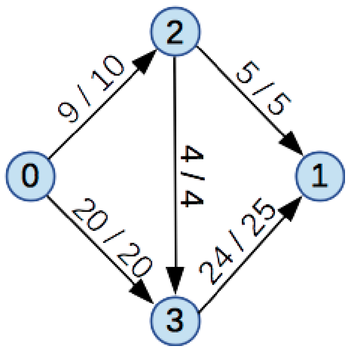
Program Output

The output of your program will describe the maximum flow from the source to the sink. First, you will print a line giving the the total value of the maximum flow (the total amount of commodity flowing from source to sink). Then, you will print the list of edges in the same order as in the input. For each edge, give the flow across that edge rather than its capacity. If there are multiple flows that produce the maximum, you can report any of them, as long as the flow you report is one you could get using BFS to find augmenting paths.

Sample Output

```
29
0 2 9
0 3 20
2 3 4
2 1 5
3 1 24
```

Along with the graph description above, this describes a flow that looks like:



Sample Execution

Once your program is working, you should be able to run it as follows. Here, I'm pretending the solution is implemented in C or C++.

```
# run on the smallest input
$ ./maxflow < input_f1.txt
29
0 2 9
0 3 20
2 3 4
2 1 5
3 1 24
# run on the medium-sized input, the same as the one in
# the sample animation.
$ ./maxflow < input_f2.txt
10
0 2 7
0 5 3
2 3 5
2 4 2
3 1 3
3 7 2
4 3 0
4 7 4
5 2 0
5 4 2
5 6 1
6 4 0
6 7 1
7 1 7
```

When I posted this assignment, I just had two sample inputs. I'll make a third, larger one.