Design and Analysis of Algorithms E
FAST-NU, Lahore, Spring 2019

Homework 4

Dynamic Programming

Due Monday April 15 before class — Marked out of 100 points.

Note: For the coding problems you must submit print outs of your C++ codes. You must also submit the actual code files on SLATE, before 11:55 PM on the deadline.

Problem 1

Recall the Shortest Path in DAGs algorithm. For your convenience, the pseudo-code is reproduced below:

ShortestPathDAG (G=(V, E), s)

```
TopologicalSort(G=(V,E))
For each v ∈ V before s in the linearized order
        dist(v) ← ∞
        parent(v) ← '-'
dist(s) ← 0
parent(s) ← '-'
For each v ∈ V after s in the linearized order
        For each u ∈ V such that (u, v) ∈ E (i.e. u is predecessor of v)
                IF(dist(v) > dist(u) + w(u,v))
                        dist(v) ← dist(u) + w(u,v)
                        parent(v) ← u
```

(a) Implement this algorithm in C++. The graph should be passed to your function in the *adjacency list* format. Use the following structure for this purpose:

```
vector< list<neighbor> > G;
```

where neighbor is defined as:

```
struct neighbor{
  int vnum;//vertex number
  int weight; //edge weight
};
```

Be particularly careful about the green portion in the code. It must take overall O(|V| + |E|) time in your implementation. Be careful how to travers all the predecessors u of v. You are free to change the looping as long it has the same effect as in the code above.

Read the graph from a text file in the following format:

```
3
0 1
1 2
2 0 1
```

This is a graph with three nodes, namely 0, 1 and 2, with edges (0, 1), (1,2), (2, 0) and (2, 1) respectively.

(b) Sometimes it is necessary in network communication that no path consists of more than a certain number of hops, say *m* hops, where hops are simply edges, even though some path with more than *m* hops may exist between two vertices that is shorter in terms of total weight of its edges. This constraint makes sense because more hops mean more channels of communication and potential packet loss etc. Modify the ShortestPathDAG DP algorithm given above so that it finds the shortest path from s to any other node with no more than *m* hops. If no path exists between *s* and *v* with less than or equal to *m* hops, then dist(v) should be infinity.

Problem 2

In this problem, we ask you to implement the edit distance DP algorithm studied in class, but in the top down fashion, using *memoization*, as explained in the context of the Rod Cutting problem. Once again, write C++ code, use the STL vectors to make the DP matrices:

```
vector<vector<int>> E, S; //E for values, S to reconstruct solution
```

Your program should ask the user to input two strings, and then output their optimal alignment and its cost.

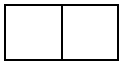
Your program should allow the user to set the penalties *i*, *d* and s for insert, delete and substitute in a config.txt file. The algorithm should work according to these penalties. A match should always get the penalty 0.

Problem 3

You have a strange room in your house and a strange set of mats to cover its floor with. The room has dimensions $2 \times k$ sq. feet, and each mat is a rectangle of exactly 2 sq. feet. Following pictures explain the situation:

A 2 sq. feet mat may be placed horizontally or vertically as shown below. Note the colors in the following pictures are only there to show you the orientations of the mats. All mats are identical in reality, having the same color.

A mat placed horizontally



A mat placed vertically

a 2x1 floor

There is only one way to mat this floor, and that is:

a 2x2 floor

There are two ways to mat this floor, and they are:

Can you see that there are 3 ways to cover a 2x3 floor? How many ways are there to cover a 2x4 floor?

Write a DP algorithm to count the number of ways there are for you to mat the $2 \times k$ sq. feet floor of your room. Give a recurrence, and write a bottom-up pseudo-code and implement it in C++. Your program should ask the user for the number k, and then output the corresponding number of ways. ***For this problem, you must prove the correctness of your recurrence to get credit.***

Problem 4

You were born during the 5th great war between the planets Earth and Zopita. Growing up, you despised the atrocities of interplanetary-warfare and dreamt of universal peace. After a long, long struggle – years of conflict (and a betrayal in love by a Zopita sweetheart) – it all boiled down to you solving a computational problem to put the wretched war to an end! This was what you had to do: Given an unbroken stream of characters, transmitted from the Zopita Headquarters, you had to determine whether the stream was a sentence of valid English words (the grammar does not matter, neither does the fact that the sentence makes any sense). For example, the stream "biguydhellfi" does not give a string of valid words, but the stream, "unleashdeadlyturtleheadsonearthplease", is just the sentence: "unleash deadly turtleheads on earth please". To your aide, you have a lexicon of 40th century English, prepared by the elusive inter-galactic hermit, Barig Saim. Given a string w, it returns true if w is a valid English word and false otherwise in $O(1)$. Given a stream of length n, describe a DP algorithm that will detect whether or not it is a valid English sentence in $\theta(n^2)$. Your algorithm should also print the valid string. Beware: the future of Earth is in your hands!

(a) The following is required:

- Define an optimal subproblem, E[k]
- Define a recurrence to compute subproblem E[k] using smaller subproblems.
- Define an array S[k] that can be used to retrieve an optimal splitting of the war message, if one exists.

(b) Implement your algorithm in C++. Your program should read the war message from a text file, msg.txt, into a C++ string; break it into valid words using your DP algorithm, if possible, and print them. Following is the code for a class called *lex*. Simply replace the file's path with its path on your machine. The dictionary file is also supplied with this statement. The method *lex*.isword is an $O(1)$ method which accepts a string and reports if it's a valid word in the dictionary. Note: you might find some string methods useful, e.g. the method s.substr(i,j) which returns the substring of s between indices i and j, etc.

```
#include <set>
#include <string>
#include <fstream>
using namespace std;

class lex{
        set <string> base;
public:
        lex(string filepath){
                ifstream ifile(filepath);
                        char buff[1000];
                        while(ifile.getline(buff,999)){
```

```
                        string b=buff;
                        b.resize(b.size()-1);
                        base.insert(b);
                    }

                ifile.close();
        }
        bool isword(string s){
                return (base.count(s)>0);
        }
};
//sample, how to use
int main(){
    lex l("/Users/sarimbaig/Documents/temp/lex.txt");

    if(l.isword("pineapple")){
        //...
    }
}
```

Problem 5

The following table defines the * operator between three characters: x, y and z.

*	x	y	z
x	y	y	x
y	z	y	x
z	x	z	z

Your algorithm receives a string of the characters x, y and z, for example yyyyxz. It then decides whether it is possible to bracket the sequence such that the result of the overall operation comes out to be x. If yes, the algorithm outputs the bracketing. For example, for the sequence given above, there exists a bracketing: $\big(y * (y * y) * (y * x)\big) * z = x$. The algorithm should be as efficient as possible.

(a) How many bracketings exist for a string of n characters? This would be the number of combinations an exhaustive algorithm would need to check.
(b) Define a sub-problem to be solved by the DP procedure.
(c) Give a recurrence to compute subproblems.
(d) Implement the algorithm in C++. It should ask the user to enter a string of x, y, and z characters, and show a bracketing that produces x, or report that none exists.

THE END