Design and Analysis of Algorithms, Assignment 3

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).

This assignment includes a couple of programming problems and some problems where you get to write up an argument or an analysis. For the non-programming problems, you'll need to turn in a PDF file for your solution. You can prepare the solution in any software package you like (e.g., Microsoft Word, Open Office, LaTeX), but you'll need to use correct mathematical notation in whatever package you choose, and you'll need to create a PDF version of your work when you turn it in.

For your 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. Matrix Chain Multiplication (25 points)

   You're going to write a program called mult.c (or mult.cpp, mult.java or mult.py, depending on your language choice). This program will solve the matrix chain multiplication problem. It will read in the dimensions of a series of matrices and report the least expensive way to multiply them as a fully parenthesized product of matrices.

   If there's more than one equally good way to parenthesize the matrix chain, either answer is fine (I won't test your program on inputs like this).

Input Format

Input will be given on standard input. It will start with a positive integer, n. This gives the number of matrices that are to be multiplied. This will be followed by $n + 1$ positive integers, one value per line. The first value is the height of the first matrix in the chain. The next value is the height of the second matrix (and also the width of the first matrix). This will continue up to the last value, which gives the width of the last matrix.

The input for the simple example we looked at in class would be given as:

```
7
3
8
4
6
6
9
2
5
```

Output

Print a single line of output (to standard out) giving fully parenthesized expression showing how the product should be computed in order to minimize computation cost. The names of the matrices will be written as `M1 M2` ... Leave one space between matrices, parentheses and multiplication operators, and don't put parentheses around individual matrices or around the entire expression. The output for the example above should look like:

```
( M1 * ( M2 * ( M3 * ( M4 * ( M5 * M6 ) ) ) ) ) * M7
```

Sample Execution

Once you program is working, I should be able to run it as follows. The following shows how it would be run if it was written in python. Shell commands would be similar if it was written your solution in a different language.

```
# Run the program on the smallest input, saving output to
# a file.
$ python ./mult.py < input_m1.txt > output.txt
# Get diff to see if there are any differences in the output.
$ diff output.txt expected_m1.txt
```

2. Look at problem 15-9 on page 410 of your textbook. Describe a dynamic programming algorithm for solving this problem and present a short analysis for the amount of space needed for the algorithm and its running time. This should take about half a page. You don't have to give code for your algorithm. Just describe the table of subproblems it fills in and give a rule for filling it in, like in equations 15.6 or 15.9 in your book. Write up your answer as a file named `break.pdf` (10 pts).

3. Greedy techniques don't always work. Consider the activity selection problem. Our greedy solution always selects the activity with the earliest finish time from among those activities that start after the most recently selected activity. What if we selected the activity with the earliest start time instead? This wouldn't always produce an optimal solution. Prove this by describing a specific example where this technique fails to produce an optimal solution. Write up your explanation in a file called `activity.pdf`. (10 pts)

4. Strongly Connected Components (25 points)

   When you're teaching a new subject, you often have to think about dependency among the topics you need to cover. For example, it's probably necessary to understand what a variable declaration is before you start trying to learn about scope, and it's probably necessary to understand something about types before you can understand a variable declaration. Sometimes, there may be so much dependency that you have some sets of topics that can't be presented in an order that satisfies the dependency. For example, topic A may require that you first understand B. Topic B may require that you first understand C and C may require that you first understand A. In cases like this, the best you can do is to try to introduce all of the topics at once during the same lecture. Your job is to write a program that finds sets of topics like this and reports them. This way, an instructor can think ahead a bit and try to anticipate lectures that will go a little bit long.

   Call your solution `dependency.c`, `dependency.cpp`, `dependency.java` or `dependency.py`, depending on your implementation language.

Input Format

Input will be given on standard input. It will start with a positive integer, n. This gives the number of different topics that need to be covered in the course. The next n lines each give the name of a topic. All topic names are sequences of non-whitespace characters (so they should be easy to parse). The list of topics is followed by a non-negative integer m and a list m of dependencies between pairs of topics. A dependency is given as a pair of course topics where you must understand the first topic before you can understand the second. For example, if a dependency is given as:

```
types declarations
```

it means you have to understand types in order to understand declarations.

Output

Print out all topic sets, S, where $|S| > 1$ where the topics in S must all be covered in the same lecture. List topics in each set on a single line, space separated and sorted in the same order they were given in the topic list at the start of the input. List sets of topics sorted by the order their first topic was given in the input.

Sample Execution

I'm providing a few sample inputs and expected outputs for your dependency program. For C and C++, you should be able to compile your program and run it like the following. As with the earlier examples, I'm using shell comments to explain what I'm doing. If you're programming in Java or Python, you can modify these examples, to try out your solution.

```
# Run the dependency program on a simple input.  The same
# problem as in input_d1.txt.  I just cut-and-pasted it into
# the terminal, instead of redirecting input from a file
# so you could see what the input looks like.  I'm sending the
# output to a file, so it's clear what's input and what's
# output.
$ ./dependency > output.txt
5
variables
types
scope
assignments
declarations
5
types declarations
variables scope
scope declarations
declarations variables
variables assignments
variables scope declarations
# Have a look at the output it produced.
$ cat output.txt
declarations scope variables
# Run on a slightly larger input
$ ./dependency < input_d2.txt
apples foil
boxes detergent eggplant
```

5. Huffman Codes (15 extra credit points)

   Write a program named huffman.c (or huffman.cpp, huffman.java or huffman.py, as appropriate). This program will read a sequence of bytes from standard input until it reaches the end-of-file. Your program will build a Huffman code based on the frequency of byte values seen in the input.

   We want a code that will be able to encode any sequence of bytes (not just the particular byte values that happened to show up in the input), so your program will create codes for every possible byte value, plus one more code to represent the end of the encoding. When you build your Huffman code, assume the following frequencies:

   - If character c didn't occur in the input, it will be given a frequency of 1. So, character c will have a code, but it will probably be a long code, since we're giving it such a low frequency.
   - If character c occurred f times in the input, it will be given a frequency of $f + 1$. So, characters that occur in the input will have a higher frequency than characters that don't occur.
   - An imaginary symbol, EOF will be given a frequency of 1. This will give us a code we can use to mark the end of file in the compressed representation. Remember, since a Huffman code is a variable length code, the last code may end part-way through the last byte of a compressed file. Having a code for EOF gives us a way to mark where the sequence of codewords ends, even if it doesn't fall right at the end of the last byte.

   Recall that the algorithm for generating Huffman codes uses a priority queue of nodes, so it can easily choose the two lowest-frequency nodes. For this, you will use your trinary heap from the previous assignment. You'll need to modify it some; previously it just had to store key values, but now it will have to store the node you're usinging to build your Huffman coding tree (or, more likely, pointers to nodes).

   Once you've determined codewords for every value, your program will print out a table of Huffman codes for each possible byte value, and an extra code for marking the end-of-file. Print your table with the byte value right-justified in a 3-column field to the left, then a space, then the Huffman code as a sequence of bits (a string of 1 and 0 characters). For bytes between 33 and 126 (inclusive), assume the byte represents the ASCII code for a character and print it out as a one-character symbol. For other byte values, just print a 2-character hexadecimal value for this byte. At the end, print the code for marking the end-of-file. For this code, just print EOF in the left-hand column.

Sample Execution

For this problem, you're getting a few sample input files. The first two are text files, but the last one is binary (so it will probably look like garbage if you try to view it in a text editor). Your program should be able to handle any of these. For this problem, depending on how you break ties, you may end you with a different Huffman code than I did (even one with different codeword lengths for some codes). That's fine; that's why the sample output files are called "typical" rather than "expected". When we test your solution, we'll try to see if you got a correct code even if you didn't get exactly the same code we did.

```
# Generate a huffman code for a short text file, where lower-
# case letters and newlines are the only symbols it actually
# contains.
$ ./huffman < input_h1.txt
 00 10010100110
 01 10001100111
 <a few lines omitted>

 09 10010010010
 0A 011011
 0B 111001010100

 <lots of lines omitted>

  ‘ 111001011101
  a 1111
  b 110110
  c 10100
  d 000010
  e 0001
  f 00000
  g 1110011
  h 10101
  i 0011
  j 0110100
  k 1101111
  l 11010
  m 0100
  n 11101
  o 1011
  p 100000
  q 1001111
  r 0101
  s 0111
  t 0010
  u 1100
  v 1000101
  w 111000
  x 000011
  y 01100
  z 1101110
  { 10001111111

 <lots of lines omitted>

 FF 10001100110
EOF 10010100111
```