Final Examination

CS 4104 (Fall 2009)

Assigned: December 2, 2009.
Due: at Torgerson 2160B by 5pm on December 14, 2009.
DO NOT EMAIL YOUR SOLUTIONS TO ME!

Name: _____

9-digit PID: _____

Instructions

1. For every algorithm you describe, prove its correctness and analyse its running time and space used. I am looking for clear descriptions of algorithms and for the most efficient algorithms you can come up with.

2. You may consult the textbook, your notes, or the course web site to solve the problems in the examination. You **may not** work on the exam with anyone else, ask anyone questions, or consult other textbooks or sites on the Web for answers. **Do not use** concepts from chapters in the textbook that we have not covered.

3. You must prepare your solutions digitally and submit a hard-copy.

4. I prefer that you use LaTeX to prepare your solutions. However, I will not penalise you if you use a different system. To use LaTeX, you may find it convenient to download the LaTeX source file for this document from the link on the course web site. At the end of each problem are three commented lines that look like this:

   ```
   % \solution{
   %
   % }
   ```

   You can uncomment these lines and type in your solution within the curly braces.

5. Do not forget to staple the hard copy you hand in.

Good luck!

Problem 1 (10 points) Let us start with some quickies. For each statement below, except for the last one, say whether it is true or false. You do not need to provide a rationale for your solution.

1. For the problem of computing the largest matching in a bipartite graph, the Ford-Fulkerson algorithm runs asymptotically faster than the scaling algorithm.
2. $\sum_{i=1}^{n} i = \Theta(n^2)$.
3. In a directed graph, $G = (V, E)$ let $\pi(u, v)$ denote the length of the shortest path that starts at u and ends at v. If there is no such path, you can assume that $\pi(u, v) = \infty$. Since G is directed, $\pi(u, v) \neq \pi(v, u)$, in general. Then, for any three nodes $u, v, w \in V$, $\pi(u, v) + \pi(v, w) \geq \pi(u, w)$.
4. Suppose we have an algorithm that when given a graph G with n nodes and an integer k, checks if G has a vertex cover of size k or less in $O(2^k n)$ time. Note that this time is polynomial in n (but exponential in k). Since we reduced Independent Set to Vertex Cover in class, we have an algorithm that can check if a graph G has an independent size of size at least l (for some integer $l > 0$) in time polynomial in n.
5. Estimate the probability that Homer Simpson will resolve the $\mathcal{P} = \mathcal{NP}$ conundrum in the series finale of the *The Simpsons*. Please provide a probability. Do not answer "True" or "False".

Problem 2 (30 points) Many object-oriented programming languages implement a class for manipulating strings. A primitive operation supported by such languages is to split a string into two pieces. This operation usually involves copying the original string. Hence, a split operation takes n units of time for a string of length n, *regardless of the location of the split*. However, if we want to split a string into many pieces, the order in which we make the breaks can affect the total running time of all the splits.

For example, suppose we want to split a 20-character string at positions 3 and 8. If we make the first cut at position 3, the cost of the first cut is the length of the string, which is 20. Now the cut at position 8 falls within the second string, whose length is 17, so the cost of the second cut is 17. Therefore, the total cost is 20 + 17 = 37. Instead, if we make the first cut at position 8, the cost of this cut is still 20. However, the second cut at position 3 falls within the first string, which has length 8. Therefore, the cost of the second cut is 8, implying a total cost of 20 + 8 = 28. Since 28 < 37, it is better to make the first cut at position 8 and the second cut at position 3. However, if we want to split the same string at positions 3 and 18, you can convince yourself that the smallest total cost is achieved by first making the cut at position 3 and then at position 18.

Design an algorithm that, given the locations of m cuts in a string of length n, finds the minimum cost of breaking the string into $m+1$ pieces at the given locations. Your algorithm must minimise the total cost of breaking the string at all the cut locations. You can assume that the cut locations are distinct and that no cut is at position 1 or at position n. *Hint:* A greedy approach will not work. Neither will divide and conquer.

Problem 3 (40 points) You are given an undirected graph $G = (V, E)$. Each vertex $v \in V$ has a label $l_v \in \{-1, 0, 1\}$. Each edge $e \in E$ has a weight $w_e > 0$. Consider the set V_0 of nodes in V whose label is 0. A *labelling* of V_0 is an assignment of 1 or -1 as the label of every node in V_0. Note that there are $2^{|V_0|}$ possible labellings of V_0. Your goal is to compute a labelling that takes the edge structure of G into account. For example, if a node v in V_0 has many more neighbours with label 1 than -1, you may want to change v's label to 1. Therefore, you decide to compute a labelling that maximises the *consistency* $c(G)$ of G, defined as

$$c(G) = \sum_{e=(u,v) \in E} w_e l_u l_v.$$

Devise a polynomial time algorithm to maximise $c(G)$. Note that your algorithm must produce a label of 1 or -1 for *every* node in V_0. Therefore, when your algorithm completes, every node in V should have a label of either 1 or -1. *Hint:* A greedy approach will not work. Neither will divide and conquer.

Problem 4 (20 points) This problem deals with cost-effective purchase of candy, a resource-allocation problem of particular importance. You enter a candy store. In front of you are glass jars, each filled with a different mouth-watering type of candy. You are on a strict budget: you have at most $\$b$ with you. There are n types of candy, each in a unique jar. For each $1 \leq i \leq n$, the entire jar containing the ith type of candy costs $\$c_i$ and weighs w_i kilogrammes. You are allowed to purchase as much candy as you want from each jar. You can purchase as tiny a portion of any piece of candy as you want. Put another way, you can think of the candy as being powdered[1] and you can scoop up whatever weight you want from each jar. Devise an algorithm that maximises the total weight of the candy you purchase, as long as the total cost of the candy you buy is at most $\$b$.

[1] I am not suggesting that anyone will actually want to *eat* powdered candy.