HW 5 (due Tuesday, at noon, October 14, 2014)

CS 473: Fundamental Algorithms, Fall 2014 Version: **1.1**

Make sure that you write the solutions for the problems on separate sheets of paper. Write your name and netid on each sheet.

Collaboration Policy: The homework can be worked in groups of up to 3 students each.

1. (40 PTS.) Simultaneous Climbs.

One day, Kris (whom you all know and love from past climbing competitions that took place during your first midterm) got tired of climbing in a gym and decided to take a very large group of climber friends (after, all he is a popular guy) outside to climb. The climbing area where they went, had a huge wide boulder, not very tall, with various marked hand and foot holds. Kris took a look and quickly figured out an "allowed" set of moves that his group of friends would do so that they get from one hold to another. He also figured out the difficulty of each individual move and assigned a grade (weight) to it. The higher the weight, the harder the move. Let $G = (V, E)$ be the (undirected) graph with a vertex for each hold and an edge between two holds (u, v) if v can be reached from u (and vice versa) by one of the moves that Kris decided. For an edge $(u, v) \in E$, we have a weight $w(uv)$ associated with (u, v) which represents the difficulty that he assigned to that particular move (it is always positive, negative weights would mean that climbers can defy gravity).

A ***k-climb*** is a sequence where a climber performs k moves in sequence. In graph G it is represented by a simple path with exactly k edges in it. Two k-climbs are disjoint if they do not share any vertex. A collection M of k-climbs is a ***k-climb packing*** if all pairs of climbs of M are disjoint (for $k = 1$ the set M is a matching in the graph). The total weight of a k-climb packing is the total weight of the edges used by the climbers.

Kris and his friends decided to play a game (they are all very good climbers), where as many climbers as possible are simultaneously on the wall and each climber needs to perform a set of k moves in sequence. In other words, they are interested in the problem of computing the maximum weight k-climb packing in G. In general, this problem seems hard.

Describe an efficient algorithm (i.e., provide pseudo-code, etc), as fast as possible, for computing the maximum weight k-climb packing when G is a rooted tree (fortunately for the tree case this is much easier) or a forest (collection of rooted trees).

Your algorithm should be recursive and use memoization to achieve efficiency. (You can not assume G is a binary tree - a node might have arbitrary number of children.) What is the running time of your algorithm as function of $n = |V(G)|$ and k?

2. (30 PTS.) Process these words.

In a word processor the goal of "pretty-printing" is to take text with a ragged right margin, like this

```
I guess it takes two
to progress
from an apprentice to a
legitimate surfer. Two digits
in front of the
size in feet of the
wave one needs to take, two double overhead waves that holds
one under after
the wipeout, and two equally sized pieces that
one's previously intact board comes up
on the surface as.
```

and turn it into text whose right margin is as "even" as possible, like this

```
I guess it takes two to progress
from an apprentice to a legitimate
surfer. Two digits in front of
the size in feet of the wave one
needs to take, two double overhead
waves that holds one under after
the wipeout, and two equally sized
pieces that one's previously intact
board comes up on the surface as.
```

To make this precise enough for us to start thinking about how to write a pretty-printer for text, we need to figure out what it means for the right margins to be "even". So suppose our text consists of a sequence of *words*, $W = \{w_1, w_2, \ldots, w_n\}$, where w_i consists of c_i characters. We have a maximum line length of L. We will assume we have a fixed-width font and ignore issues of punctuation or hyphenation.

A *formatting* of W consists of an ordered partition of the words in W into *lines*. In the words assigned to a single line, there should be a space after each word except the last; and so if $w_j, w_{j+1}, \ldots, w_k$ are assigned to one line, then we should have

$$[\sum_{i=j}^{k-1}(c_i+1)] + c_k \leq L.$$

We will call an assignment of words to a line *valid* if it satisfies this inequality. The difference between the left-hand side and the right-hand side will be called the *slack* of the line-that is, the number of spaces left at the right margin.

Given a partition of a set of words W, the *penalty* of the formatting is the sum of the *squares* of the slacks of all lines (including the last line). Give an efficient algorithm to find a partition of a set of words W into valid lines, so that the penalty of the formatting becomes minimized.

3. (30 PTS.) Spanglish

The wildly popular Spanish-language search engine El Goog needs to do a serious amount of computation every time it recompiles its index. Fortunately, the company has at its disposal a single large supercomputer, together with an essentially unlimited supply of high-end PCs. They have broken the overall computation into n distinct jobs, labeled $J_1, J_2, \ldots J_n$, which can be performed completely independently of one another. Each job consists of two stages: first it needs to be preprocessed on the supercomputer, and then it needs to be finished on one of the PCs. Let us say that job J_i needs p_i seconds of time on the supercomputer, followed by f_i seconds of time on a PC.

Since there are at least n PCs available on the premises, the finishing of the jobs can be performed fully in parallel - all the jobs can be processed at the same time. However, the supercomputer can only work on a single job at a time, so the system managers need to work out an order in which to feed the jobs to the supercomputer. As soon as the first job in order is done on the supercomputer, it can be handed off to a PC for finishing; at that point in time a second job can be fed to the supercomputer; when the second job is done on the supercomputer, it can proceed to a PC regardless of whether or not the first job is done (since the PCs work in parallel); an so on.

Let us say that a schedule is an ordering of the jobs for the supercomputer, and the completion time of the schedule is the earliest time at which all jobs will have finished processing on the PCs. This is an important quantity to minimize, since it determines how rapidly El Goog can generate a new index.

Give a polynomial-time algorithm that finds a schedule which as small a completion time as possible.