CS 170 Algorithms
Spring 2014 Elchanan Mossel
HW 7

Due April 4 , 6:00pm

1. (15 pts.) Cutting cloth

You are given a rectangular piece of cloth with dimensions $X \times Y$, where X and Y are positive integers, and a list of n products that can be made using the cloth. For each product $i \in [1,n]$ you know that a rectangle of cloth of dimensions $a_i \times b_i$ is needed and that the final selling price of the product is c_i. Assume the a_i, b_i and c_i are all positive integers. You have a machine that can cut any rectangular piece of cloth into two pieces wither horizontally or vertically. Design an algorithm that determines the best return on the $X \times Y$ piece of cloth, that is, a strategy for cutting the cloth so that the products made from the resulting piece give the maximum piece of selling prices. You are free to make as many copies of a given product as you wish, or none if desired.

2. (15 pts.) Optimal binary search trees

Suppose we know the frequency with which keywords occur in programs of a certain language, for instance:

begin	5%
do	40%
else	8%
end	4%
if	10%
then	10%
while	23%

We want to organize them in a binary search tree, so that the keyword in the root is alphabetically bigger than all keywords in the left subtree and smaller than all keywords in the right subtree (and this holds for all nodes).

Figure 1 has a nicely-balanced example on the left. In this case, when a keyword is being looked up (for compilation perhaps), the number of comparisons needed is at most three; for instance, in finding "while", only the three nodes "end", "then", and "while" get examined. But since we know the frequency with which keywords are accessed, we can use an even more fine-tuned cost function, the *average number of comparisons* to look up a word, For the search tree on the left, it is

$$cost = 1(0.04) + 2(0.40 + 0.10) + 3(0.05 + 0.08 + 0.10 + 0.23) = 2.42$$

By this measure, the best search tree is the one on the right, which has a cost of 2.18.

Give an efficient algorithm for the following task.

Input: n words (in sorted order); frequencies of these words: $p_1, p_2, \ldots, p_n$.

Output: The binary search tree of lowest cost (defined above as the expected number of comparisons in looking up a word).

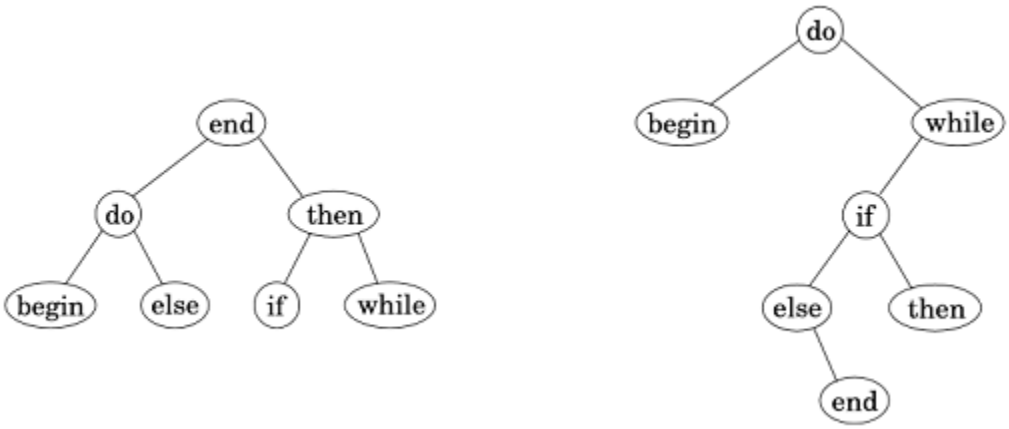

Figure 1: Two binary search trees for the keywords of a programming language

3. (20 pts.) Timesheets Part 2

Recall problem 4 from homework 6.

Suppose we have N jobs labelled $1,\ldots,N$. For each job, you have determined the bonus of completing the job, $V_i \geq 0$, a penalty per day that you accumulate for not doing the job, $P_i \geq 0$, and the days required for you to successfully complete the job $R_i > 0$.

Every day, we choose one unfinished job to work on. A job i has been finished if we have spent R_i days working on it. This doesn't necessarily mean you have to spend R_i contiguous sequence of days working on job i. We start on day 1, and we want to complete all our jobs and finish with maximum reward. If we finish job i at the end of day t, we will get reward $V_i - t \cdot P_i$. Note, this value can be negative if you choose to delay a job for too long.

Now, what we did not tell you last time is that we have a time limit of T days, in which we can choose to work on some of these jobs in only those T days. Given this information, what is the optimal job scheduling policy with a time limit of T days? Notice that 0 is a lower bound since we can choose to do no jobs at all if all of them happen to have negative value, or all of them take more than time T.

4. (15 pts.) Sorting with errors Recall the *sorting with errors* problem from class. We want to sort where comparisons may be faulty that is for elements i, j we may have the wrong answer to the query $a[i] < a[j]$?. The goal is to find an order $a[1], a[2], \ldots, a[n]$ that minimizes the number of $i < j$ with $a[i] > a[j]$. We saw in class that if we have an *almost optimal* ordering to start with where locations of elements are at most k positions away from locations in the optimal ordering, we can solve this in $O(2^{6k}n^2)$ time. Give an algorithm for this problem that runs in $O(2^n n^2)$ time when we don't have such an *almost optimal* ordering to use as a seed.

5. (15 pts.) Exon chaining Each gene corresponds to a subregion of the overall genome (the DNA sequence). Frequently, a gene consists of several pieces called exons, which are separated by fragments called introns. This complicates the process of identifying genes in a newly sequenced genome. Suppose we have a new DNA sequence and we want to check whether a certain gene (a string) is present in it. Because we cannot hope that the gene will be a contiguous subsequence, we look for partial matches – fragments of the DNA that are also present in the gene (actually, even these partial matches will be approximate, not perfect). We then attempt to assemble these fragments. Let $x[1\ldots n]$ denote the DNA sequence. Each partial match can be represented by a triple (l_i, r_i, w_i), where $x[l_i \ldots r_i]$ is the fragment and w_i is a weight representing the strength of the match (it might be a local alignment score or some other statistical quantity). Many of these potential

matches could be false, so the goal is to find a subset of the triples that are consistent (nonoverlapping) and have a maximum total weight. Show how to do this efficiently.

6. **(20 pts.) Time and space complexity of dynamic programming** Our dynamic programming algorithm for computing the edit distance between strings of length m and n creates a table of size $n \times m$ and therefore needs $O(mn)$ time and space. In practice, it will run out of space long before it runs out of time. How can this space requirement be reduced?

   (a) Show that if we just want to compute the value of the edit distance (rather than the optimal sequence of edits), then only $O(n)$ space is needed, because only a small portion of the table needs to be maintained at any given time.

   (b) Now suppose that we also want the optimal sequence of edits. As we saw earlier, this problem can be recast in terms of a corresponding grid-shaped dag, in which the goal is to find the optimal path from node $(0,0)$ to node (n,m). It will be convenient to work with this formulation, and while we're talking about convenience, we might as well also assume that m is a power of 2. Let's start with a small addition to the edit distance algorithm that will turn out to be very useful. The optimal path in the dag must pass through an intermediate node $(k,m/2)$ for some k; show how the algorithm can be modified to also return this value k.

   (c) Now consider a recursive scheme:

```
procedure find-path((0,0) to (n.m))
compute the value of k above
find-path ((0,0) to (k,m/2))
find-path ((k,m/2) to (n,m))
concatenate these two paths, with k in the middle
```

   Show that this scheme can be made to run in $O(mn)$ time and $O(n)$ space.