Algorithms Section 2
Computer Science 140 & Mathematics 168
Spring 2003
Homework 4a
Due Friday, February 14

This is a challenging and conceptually important homework set. Please start early and come to office hours with any questions.

1. **[20 Points] More Fun at the Firm of Weil, Proffet, and Howe!** You have been hired as Vice President for Algorithm Design at the Wall Street firm of Weil, Proffet, and Howe. Consider an array A of values of a share for a particular company over the last n days. Assume that A is indexed from 1 to n and $A[1]$ corresponds to the value of a share on day 1, $A[2]$ corresponds to the value of a share on the next day, and so forth.

   A *monotonically increasing subsequence* of the array is a sequence of array elements $A[i_1], A[i_2], \ldots, A[i_k]$ such that $A[i_1] < A[i_2] < \ldots < A[i_k]$ and such that $i_1 < i_2 < \ldots < i_k$. For example, a monotonically increasing subsequence of the array $5, 12, 7, 9, 6$ is $(5, 7, 9)$.

   Assume that we are given an array A of length n. Our goal is to find the length of the longest monotonically increasing subsequence *which includes the element* $A[n]$. That is, $A[n]$ must be the last element in the subsequence. For example, for the array $5, 2, 7, 3, 6$, we have $n = 5$ and the length of the longest monotonically increasing subsequence which includes $A[n]$ is 3. (The subsequence $(2, 3, 6)$.)

   (a) The previous Vice President for Algorithm Design, Dr. Juss Dooit, proposed a brute-force algorithm for finding the length of the longest monotonically increasing subsequence which includes $A[n]$. This algorithm simply enumerated all possible subsequences, one-by-one, tested to see if each enumerated subsequence was monotonically increasing, and if so, compared the length of that subsequence to the longest monotonically increasing subsequence seen so far. Consider the **Crayfish 2000** computer that can enumerate and test 10^{12} subsequences per second. How long would it take to execute Dr. Dooit's algorithm on an array of length 100 on the Crayfish? Why was Dr. Dooit fired?

   (b) Professor Rhea Cursive wants to find a recursive algorithm for solving this problem. The algorithm `mono`(i) will return the *length* of the longest monotonically increasing subsequence in $A[1], \ldots, A[i]$ which includes $A[i]$. Clearly, if she can write such a recursive algorithm, we can run it with $i = n$ to solve our problem. (Notice that `mono` doesn't take the array A as input. We're assuming that A is implicit. That is, it's a global variable that `mono` can access.) Professor Cursive has a vague notion that `mono`(i) can be computed as follows: Look at each j from 1 to $i - 1$ and check to see if $A[j] < A[i]$. If so, make a recursive call to compute mono(j). Now, do something with these mono(j) values to compute the desired mono(i). Write pseudo-code for mono(i). (It should be comprehensible, but need not be in any particular programming language.)

(c) Explain why the worst-case running time of the algorithm is at least exponential. Use a lower bound argument similar to the ones we saw in class.

(d) Now convert this algorithm into a dynamic programming algorithm as follows: Let M be an array indexed from 1 to n. $M[i]$ will store the value of mono(i). However, we will compute $M[1]$ first, then $M[2]$, and so forth all the way up to $M[n]$. Carefully describe the procedure for computing $M[i]$. Then, derive the running time of the algorithm to fill in the entire array M. The running time should be polynomial in n. That is $\Theta(n^c)$ for some fixed positive integer c (like 2 or 3).

(e) Now, assume that the constant in front of your polynomial is something bad, like 50. Assume also that your dynamic programming algorithm is implemented and executed on a Granny Smith Personal Computer which is only able to execute 10^4 instructions per second. How long does it take to find the length of the longest monotonically increasing subsequence using this algorithm on the Granny Smith?

(f) Show the table constructed by your dynamic programming algorithm for the array $5, 2, 7, 3, 6$.

(g) Explain how your algorithm can be modified slightly to allow you to find the actual longest monotonically increasing subsequence. What does this modification do to the running time of your algorithm?

2. **[25 Points] Load Balancing on the Snapple Computer!** Snapple Computers is an emerging manufacturer of parallel computers. You've been hired to design algorithms for the operating system of Snapple's new personal computer, the Snapple Mango Melon Madness (referred to hereafter as the "M^3"). The M^3 has **two** identical UltraSuperSparkingMangoPower processors. When run in batch mode, the operating system receives an array S of the integer running times of n tasks. That is $S[1], \ldots, S[n]$ are integer running times for n tasks. The objective of the operating system is to find an optimal **load balance** for these tasks on the two processors. That is, we wish to find a partition of the tasks in S into two sets (such that the tasks in one set will be executed on one of the processors and the set of tasks in the other set will be executed on the second processor in parallel) that minimizes the total elapsed time required to complete all the tasks.

   For example, if S contains five tasks with running times 2, 1, 3, 5, 7 then we can assign the tasks with running times 2 and 7 to one processor and the tasks with running times 1, 3, and 5 to the other, resulting in an elapsed execution time of 9. On the other hand, if the five running times had instead been 2, 1, 3, 5, 8 the best we could do is have a partition of 2, 3, and 5 on one processor and 1 and 8 on the other, with an elapsed running time of 10.

   We will solve this problem in a sequence of steps. Be patient.

(a) Your boss, Dr. I. Stee, has proposed a brute-force algorithm that involves simply enumerating all possible partitions of the task set S and finding the best one. If the M^3 can evaluate one million partitions per second and the set S contains just 50 tasks, how long does it take the M^3 to find the optimal load balance? Explain how you found your answer.

(b) Now consider a variant of this problem called the *Partition Problem*. In this problem we are given the set S of n running times and we are also given a positive integer C. Our objective is return the boolean **true** if there exists a subset of S which adds up to exactly C and otherwise return the boolean **false**. To this end, give pseudo-code for a recursive function `partition`(i, C) which returns the boolean **true** if there is a subset of $S[1], \ldots, S[i]$ which adds up to exactly C and returns **false** otherwise.

(c) Give a lower bound argument to explain why `partition`(n, C) can take exponential time.

(d) Next, describe a dynamic programming algorithm that takes as input the set S and a number C and determines if there exists a subset of S that which adds up to exactly C. Describe your algorithm carefully and explain why it works.

(e) What is the asymptotic running time of your algorithm as a function of n and C?

(f) Now let's reconsider Snapple's load balancing problem. Show how your dynamic programming algorithm can be used to determine an optimal partition of S into two subsets such that the total elapsed running time of the resulting partition is minimized. Your algorithm must determine the actual partition. What is the running time of this algorithm?