Algorithms
Computer Science 140 & Mathematics 168
Instructor: B. Thom
Fall 2004
Homework 5a
Due on Friday, 10/01/04 (5pm in my office or under my door)

1. **[5 Points] Reinforcing LCS** Show what values would be stored in the Dynamic Programming Table for the LCS problem if the inputs were

$$X_m =< 1, 0, 0, 1, 0, 1, 0, 1 >$$
$$Y_n =< 0, 1, 0, 1, 1, 0, 1, 1, 0 >$$

Also show a corresponding optimal subsequence.

2. **[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. Your 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? (FYI: a back of the envelop calculation suffices here.)

(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 $\texttt{mono}(i)$ 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 $\texttt{mono}(j)$. Now, do something with these $\texttt{mono}(j)$ values to compute the desired $\texttt{mono}(i)$. Write pseudo-code for $\texttt{mono}(i)$. (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 precise 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 $\texttt{mono}(i)$. 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.

(e) Now, assume that the constant in front of your algorithm's asymptotic run-time is 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 solve the client's problem (again, for an input of length 100) using this algorithm on the Granny Smith?

(f) Show the table constructed by your dynamic programming algorithm for the array $4, 5, 2, 7, 3, 8$.

(g) Explain how your algorithm can be modified slightly to find the longest monotonically increasing sequence anywhere in A (in other words, it is no longer required that the last element be $A[n]$).

(h) Finally, suppose we wanted to know an actual longest sequence (as opposed to just the maximal length). Describe how you'd modify your algorithm to report this information.

3. **[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. Steel, 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. (FYI: a back of the envelop calculation suffices here.)

(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) Explain why the worst-case running time of `partition`(n,C) can take exponential time. Use a precise lower bound argument similar to the ones we saw in class.

(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 approach should compute an actual answer (an optimal division of jobs to processors). Also, explain (in one or two sentences) why your approach is correct and briefly analyze its runtime.

4. **[25 Points] Ski Optimization!**

Your job at Snapple is pleasant but in the winter you've decided to become a ski bum. You've hooked up with the Mount Baldy Ski Resort (they actually happen to have some snow this year!). They'll let you ski all winter for free (and drink as much free Snapple as you want!) in exchange for helping their ski rental shop with an algorithm to assign skis to skiers.

Ideally, each skier should obtain a pair of skis whose height matches his or her own height exactly. Unfortunately, this is generally not possible. We define the *disparity* between a skier and his or her skis to be the absolute value of the difference between the height of the skier and the pair of skis. Our objective is to find an assignment of skis to skiers that minimizes the sum of the disparities. We'll solve some interesting subproblems along the way to finding an efficient algorithm for the ski assignment problem.

(a) First, let's assume that there are n skiers and n skis. The previous computer scientist at the ski resort, Professor R. U. Kitting, proposed the following algorithm for this case. Consider all possible assignments of the n skis to the n skiers. For each one, compute the sum of the disparities. Finally, select the assignment that minimizes the sum of the disparities. Explain why Professor Kitting was fired. How much time would this algorithm take on a computer that performs 1 billion operations per second if there were 50 skiers and 50 pairs of skis?

(b) **While you need to understand the concept discussed in this item to complete the problem, the proof asked for here is optional (Extra Credit!)** The next computer scientist at the resort, Dr. G. I. M. Fast, spent most of her time on the slopes. She did however make an interesting discovery one day on the chair lift. She observed that if we have a short person and a tall person, it would never be better to give the shorter person a taller pair of skis than were given to the tall person. Show that this is always true. (It would be a good idea to split your argument into cases and consider each separately.)

(c) **While you need to understand the concept discussed in this item to complete the problem, the proof asked for here is optional (Extra Credit!)** After Dr. Fast's brief stint at the ski resort, you were hired. Your first job is to find a "greedy" algorithm that minimizes the sum of the disparities assuming that there are n people and n pairs of skis. You—being the brilliant engineer that you are— notice that if you first sort people by increasing height and then also sort the skis by increasing height, and then pair each i-th largest skis with the i-th largest person, an optimal solution is indeed found. Argue carefully why this algorithm is correct (you'll want to utilize part b here). In other words, show that no better solution is possible. What is the time complexity of your algorithm?

(d) Your next task is to design an efficient dynamic programming algorithm for the more general case that there are m skiers and n pairs of skis and $m \leq n$. Again, start by sorting the skiers and skis by increasing height. Let h_i denote the height of the i^{th} skier in sorted order and let s_j denote the height of the j^{th} pair of skis in sorted order. Let $A[i, j]$ be the optimal cost (sum of absolute differences of heights) for matching the first i skiers with skis from the set $\{1, 2, \ldots, j\}$. The solution we seek is then simply $A[m, n]$. Fill in the blanks below to define A recursively:

$$A[i, j] = \begin{cases} \underline{\hspace{3cm}} & \text{if } i = 0 \\ \underline{\hspace{3cm}} & \text{if } i = j \\ \underline{\hspace{3cm}} & \text{if } i < j \end{cases}$$

Now give pseudocode (with explanatory comments) or a concise and complete

English description of a dynamic programming algorithm that solves the optimal assignment problem of skis to skiers using a table for $A[i, j]$.

(e) What is the running time of your program? Explain.

(f) Briefly describe how your algorithm can be modified to allow you to find an actual optimal assignment (rather than just the cost) of skis to skiers. How does this affect the running time of your algorithm?

(g) Illustrate your algorithm by explicitly filling out the $A[i, j]$ table for the following sample data:

- Ski heights: $1, 2, 5, 7, 13, 21$.
- Skier heights: $3, 4, 7, 11, 18$.