

CS381 Homework 3

Out: Sep. 28, Thursday
Due: Oct. 12, Thursday, in (or before) class
Submissions will not be accepted afterwards.

Instructions:

Read the course policy (including academic dishonesty) in the course webpage at <http://www.cs.purdue.edu/homes/gopal/cs381>. Text refers to the Introduction to Algorithms (second edition) book. **Justify your answers. Show appropriate work.**

Reading: Chapters 8.1, 15, and 16 of Text.

Problem 1

You are given two different sets of cards — A and B . Set A consists of n different cards, each card has a distinct positive integer number written on it. Similarly set B consists of n different cards, each card has a distinct positive number written on it. Every card in set A has a matching card in set B — i.e., they have the same numbers written on both of them. Your goal is to match every card in A to its corresponding card in B . The only tool you have is a machine that answers the following type of query: given a card in A and a card in B it will tell which of the following three cases is true — the numbers on the both cards are the same, or A card has smaller number than B card, or A card has larger number than B card. Show that you need at least $\Omega(n \log n)$ queries to solve the problem. (10 points) (Hint: Use the decision tree method that you learnt to show the lower bound on sorting. How many distinct possibilities are there for the leaves of the tree?)

Problem 2

You are given two DNA sequences (strings) $s[1 \dots m]$ and $t[1 \dots n]$ and the goal is to find the “best” **alignment** between the two sequences. (DNA

strings are formed by only four types of nucleotides, represented with four alphabet letters: adenine (abbreviated A), cytosine (C), guanine (G) and thymine (T).) An alignment is an insertion of spaces in arbitrary locations along the sequences so that they end up with the same size. A space in one sequence should not align with a space in the other. But spaces can be inserted at the beginning or at the end.

For example, if $s = GACGGATTAG$ and $t = GATCGGAATAG$, then an alignment of s and t is:

$$\begin{aligned} s &= GA - CGGATTAG \\ t &= GATCGGAATAG \end{aligned}$$

That is by inserting a gap after the second symbol in s , we can make s and t to be of the same size. Of course, one can do this, by inserting a gap at any other position as well. Thus, there is more than one way one can form an alignment given two sequences.

A very important problem for researchers is to find the “best” alignment (among possibly many alignments). A best alignment can be defined as follows. First, an alignment can be **scored** by a scoring scheme. We assume a scoring matrix *score* where the entry $score[x, y]$ gives the alignment score for x and y , where x and y can be any one of the four alphabets (i.e. A,T,G,C) and the gap (i.e. “-”). Note that scoring matrix is a 5×5 matrix. The score for an alignment is the sum of the scores of its aligned characters. A best alignment is one which receives the maximum score called the **similarity score**, denoted by $sim(s, t)$. For example, the score matrix can be: $s[x, y] = 1$ if $x = y$, and $s[“-”, x] = s[x, “-”] = -1$ for any $x \in \{A, C, G, T\}$, and $s[x, y] = -2$ if $x \neq y$. Then the score for the alignment of the above example will be $1 + 1 + (-1) + 1 + 1 + 1 + 1 + (-2) + 1 + 1 + 1 = 6$. This can be verified to be the best score possible for this example and hence the alignment shown is the best alignment possible for this score matrix.

Your task is the following: given two DNA sequences (strings) $s[1 \dots m]$ and $t[1 \dots n]$, of length m and n respectively (m need not be same as n) and the score matrix as input, design a dynamic programming algorithm for finding the similarity score (i.e., $sim(s, t)$) and also output an alignment (i.e., the best alignment) that has this similarity score. Note that you should first give a recursive formulation of the dynamic programming solution, clearly explaining your notation (in particular, what the subproblems are). Then describe the algorithm and analyze its running time. You will get 10 points for recursive formulation and 10 points for algorithm description and time

analysis. (Hint: This problem has a similar approach to the longest common subsequence problem discussed in class.)

Problem 3

As discussed in class, the Knapsack problem is as follows. We are given a set $S = \{a_1, a_2, \dots, a_n\}$ of n objects, with specified sizes and profits (all values are positive integers i.e., $size(a_i)$ and $profit(a_i)$ are both positive integers for all i), and a knapsack capacity m , a positive integer. We have to find a subset of objects whose total size is bounded by m and the total profit is *maximized*. We gave a dynamic programming algorithm with running time $O(nm)$ in class. Here your task is to give an alternate dynamic programming algorithm whose running time depends on n and the value of the most profitable object.

Let P be the profit of the most profitable object, i.e., $P = \max_{a \in S} profit(a)$. (Then nP is a trivial upper bound that can be achieved by any solution — why?) For each $i \in \{1, \dots, n\}$ and $p \in \{1, \dots, nP\}$, let S_{ip} denote a subset of $\{a_1, \dots, a_i\}$ whose total profit is exactly p and whose total size is minimized. Let $A(i, p)$ denote the sum of the sizes of the elements in the set S_{ip} . (Note that $S_{i,p}$ can be empty, in which case we will set $A(i, p) = \infty$.)

(a) Design a dynamic programming algorithm to compute $A(i, p)$ for all values i and p . Give the recursive formulation. Note that you should first give a recursive formulation of the dynamic programming solution, clearly explaining your notation (in particular, what the subproblems are). Then describe the algorithm and analyze its running time. 8 points for recursive formulation and 7 points for algorithm description and time analysis.

(b) How will you find the optimal value and the optimal solution from the $A(i, p)$ values? (5 points)

Problem 4

You are given a sequence A of n distinct integers — a_1, a_2, \dots, a_n . Your goal is to find a longest increasing subsequence of A . For example, if A is 10, 4, 5, 20, 7, 30, 9, then a longest increasing subsequence of A is 4, 5, 7, 9. Note that 4, 5, 20, 30 is also a longest increasing subsequence.

(a) Consider the following greedy algorithm:

for $i = 1$ to n do

```

    Start a (sub)sequence  $S_i$  that has  $a_i$  as the first element
    for  $j = i + 1$  to  $n$  do
        Add  $a_j$  to  $S_i$  if it is greater than the current last element in  $S_i$ 
    endfor
endfor
Output the subsequence  $S_i$  that has the maximum length.

```

What is the running time of the above algorithm? Is the above algorithm correct? In other words, does it always output a longest increasing subsequence? (5 points) (Hint: You may want to try running the algorithm on a few examples.)

(b) We studied the longest common subsequence problem in class and gave a dynamic programming algorithm for it. Using this algorithm as a subroutine, give an $O(n^2)$ time algorithm to find the longest increasing subsequence of a given sequence of n numbers. (10 points) (Hint: To use the longest common subsequence as a subroutine you need two input sequences, right? You have only one input sequence here. You have to find one more “suitable” sequence.)

(c) Give a dynamic programming based algorithm that finds the longest increasing subsequence of a given sequence of n numbers in $O(n \log n)$ time. (Hint: First, note that this will be a different approach than the one used for the longest common subsequence. As usual, it is better to first think of just finding the *length* of the longest increasing subsequence. What is a good way to decompose this problem into subproblems? How about the subproblem of finding the longest increasing subsequence of the sequence a_i, a_{i+1}, \dots, a_n . There are n different such subproblems. Since you want to get a total runtime of $O(n \log n)$, you may want to solve each subproblem in $O(\log n)$ time.) (7 points for the dynamic programming formulation and 8 points for the algorithm description and analysis. Note that your algorithm has to find a longest increasing subsequence and not just its length.)

Problem 5

Suppose you are the “algorithmatician” of your company and the manager comes to you with the following problem. The company has to buy n different software products. Due to various constraints, the company can only buy these software products at the rate of *at most one per month*. Each software is currently selling for a price of \$100. However, they are all becoming more expensive according to exponential growth curves: in particular, the

cost of software j increases by a factor $r_j > 1$ each month, where r_j is a known parameter. This means that if software j is purchased t months from now, it will cost $100(r_j)^t$. It is given that all price growth rates are distinct: that is, $r_i \neq r_j$ for software $i \neq j$ (even though at the start they all have the same price of \$100.)

The question the manager poses for you is this: Given that the company can only buy at most one software product a month, in which order should it buy the products so that the *total amount of money spent is as small as possible*? In particular he has the following questions:

(a) (5 points) Consider the following example to get started. Suppose $r_1 = 2$, $r_2 = 3$, and $r_3 = 4$. What is the best order to buy the three products and why?

(b) (5 points) Give a greedy algorithm to find the optimal order assuming that we have n products to buy and the growth rate of product i is r_i , $1 \leq i \leq n$. What is the time complexity of your algorithm? (Hint: The answer to example (a) can suggest a strategy.)

(c) (10 points) Prove that your greedy algorithm always gives the optimal ordering. (Hint: Use the technique discussed in class: show that the ordering output by greedy is no worse compared to the optimal ordering. Your proof can go as follows. Suppose the greedy ordering G is different from the optimal ordering O . Let the k th element ($k \geq 1$) be the first element that is different in the two orderings. Then show that there is another ordering in which the first k elements agree with G whose cost is no worse than O .)