CS375 Assignment 14

Due Nov 11th 10PM ET

A general note for CS375: When writing up your homework, please write neatly and explain your answers clearly, giving all details needed to make your answers easy to understand. Graders may not award full credit to incomplete or illegible solutions.

You are encouraged to discuss the problems with your classmates, **but your submission should be your own.** When writing your solutions up, you should not be looking at any other student's submission.

Clear communication **is the point**, on every assignment. In general in CS375, unless explicitly specified otherwise, answers should be accompanied by explanations. Answers without explanations may not receive full credit. Please feel free to ask me any questions about explanations that might come up!

Necessary acknowledgements: many of these questions are reproductions or adaptations of questions of Eric Aaron's devising. Others are from our course texts as listed on the syllabus.

1. Consider the following problem:

> Given a graph $G = (V, E)$ (vertices V and edges E) with a weight w_v for every vertex v, find a simple path (a sequence of adjacent vertices which doesn't revisit the same vertex twice) in V of maximum total weight.

(a) First, suppose that G is itself a simple path: so $V = \{v_1, ..., v_n\}$ where each v_i is adjacent to v_{i+1}. Use dynamic programming to find the maximal weight simple path.

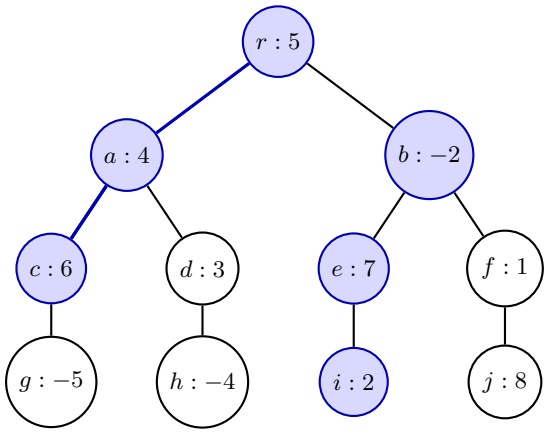
Example: Suppose the vertex weights, in order, are

$$3,\ -2,\ 5,\ 4,\ -6,\ 5.$$

The path consisting of v_3, v_4 has total weight 9, while the path consisting of v_1, v_2, v_3, v_4 has total weight 10.

(b) Next, suppose that G is a tree. Use dynamic programming to find the maximal weight simple path.

Example:



(c) Next, if G is a general graph, explain why your previous algorithm will not work.

(d) ~~Console yourself with a description of an obviously correct $n \times n!$ time algorithm for this problem.~~

(e) ~~Rejoice by using dynamic programming to find an algorithm for general graphs with a runtime of $\tilde{O}(2^n)$ (here $\tilde{O}(2^n)$ means I'm ignoring polynomial factors of n, so for example $n^3 2^n$ would be fine here).~~

2. Pick another of the following.

i) (Erickson 3.1) In a previous life, you worked as a cashier in the lost Antarctican colony of Nadiria, spending the better part of your day giving change to your customers. Because paper is a very rare and valuable resource in Antarctica, cashiers were required by law to use the fewest bills possible whenever they gave change. Thanks to the numerological predilections of one of its founders, the currency of Nadiria, called Dream-Dollars, was available in the following denominations: \$1, \$4, \$7, \$13, \$28, \$52, \$91, and \$365.

i. The greedy change algorithm repeatedly takes the largest bill that does not exceed the target amount. For example, to make \$122 using the greedy algorithm, we first take a \$91 bill, then a \$28 bill, and finally three \$1 bills. Give an example where this greedy algorithm uses more Dream-Dollar bills than the minimum possible. [Hint: It may be easier to write a small program than to work this out by hand.]

ii. Describe and analyze a recursive algorithm that computes, given an integer k, the minimum number of bills needed to make k DreamDollars. (Don't worry about making your algorithm fast; just make sure it's correct.)

iii. Describe a dynamic programming algorithm that computes, given an integer k, the minimum number of bills needed to make k Dream-Dollars. (This one needs to be fast.)

ii) (Erickson 3.3) Describe efficient algorithms for the following variants of the text segmentation problem. Assume that you have a subroutine IsWord that takes an array of characters as input and returns **True** if and only if that string is a "word". Analyze your algorithms by bounding the number of calls to IsWord.

i. Given an array `A[1 .. n]` of characters, compute the number of partitions of `A` into words. For example, given the string `ARTISTOIL`, your algorithm should return 2, for the partitions `ARTIST OIL` and `ART IS TOIL`.

ii. Given two arrays `A[1 .. n]` and `B[1 .. n]` of characters, decide whether `A` and `B` can be partitioned into words at the same indices. For example, the strings `BOTHEARTHANDSATURNSPIN` and `PINSTARTRAPSANDRAGSLAP` can be partitioned into words at the same indices as follows:

```
BOT HEART HAND SAT URNS PIN
PIN START RAPS AND RAGS LAP
```

iii. Given two arrays `A[1 .. n]` and `B[1 .. n]` of characters, compute the number of different ways that `A` and `B` can be partitioned into words at the same indices.

iii) (AD 6.2) Suppose you're managing a consulting team of expert computer hackers, and each week you have to choose a job for them to undertake. Now, as you can well imagine, the set of possible jobs is divided into those that are low-stress (e.g., setting up a Web site for a class at the local elementary school) and those that are high-stress (e.g., protecting the nation's most valuable secrets, or helping a desperate group of Cornell students finish a project that has something to do with compilers). The basic question, each week, is whether to take on a low-stress job or a high-stress job. If you select a low-stress job for your team in week i, then you get a revenue of $\ell_i > 0$ dollars; if you select a high-stress job, you get a revenue of $h_i > 0$ dollars. The catch, however, is that in order for the team to take on a high-stress job in week i, it's required that they do no job (of either type) in week $i-1$; they need a full week of prep time to get ready for the crushing stress level. On the other hand, it's okay for them to take a low-stress job in week i even if they have done a job (of either type) in week $i-1$. So, given a sequence of n weeks, a plan is

specified by a choice of low-stress, high-stress, or none for each of the n weeks, with the property that if high-stress is chosen for week $i > 1$, then none has to be chosen for week $i - 1$. (It's okay to choose a high-stress job in week 1.) The value of the plan is determined in the natural way: for each i, you add ℓ_i to the value if you choose low-stress in week i, and you add h_i to the value if you choose high-stress in week i. (You add 0 if you choose none in week i.)

i. Show that the following algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer.

```
for i in range(1, n + 1):
    if h_{i+1} > ℓ_i + ℓ_{i+1} then
        Output "Choose no job in week i"
        Output "Choose the high-stress job in week i+1
        Skip ahead to iteration i+2
    else:
        Output "Choose a low-stress job in week i
        Continue with iteration i+1
```

(to avoid overflow problems, we say $h_i = \ell_i = 0$ for $i > n$)

ii. Give an efficient algorithm that takes values for $\ell_1, ..., \ell_n$ and $h_1, ..., h_n$ and returns the *value* of an optimal plan.

iv) (AD 6.6) In a word processor, the goal of pretty-printing is to take text with a ragged right margin, like this:

```
Call me Ishmael.
Some years ago,
never mind how long precisely,
having little or no money in my purse,
and nothing particular to interest me on shore,
I thought I would sail about a little
and see the watery part of the world.
```

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

```
Call me Ishmael. Some years ago, never
mind how long precisely, having little
or no money in my purse, and nothing
particular to interest me on shore, I
thought I would sail about a little
and see the watery part of the world.
```

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, $W = w_1, w_2, ..., w_n$, 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 a 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 $w_j, w_{j+1}, ..., w_k$ 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.

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

v) (AD 6.8) The residents of the underground city of Zion defend themselves through a combination of kung fu, heavy artillery, and efficient algorithms. Recently they have become interested in automated methods that can help fend off attacks by swarms of robots. Here's what one of these robot attacks looks like.

- A swarm of robots arrives over the course of n seconds; in the ith second, x_i robots arrive. Based on remote sensing data, you know this sequence $x_1, x_2, ..., x_n$ in advance.
- You have at your disposal an electromagnetic pulse (EMP), which can destroy some of the robots as they arrive; the EMP's power depends on how long it's been allowed to charge up. To make this precise, there is a function f so that if j seconds have passed since the EMP was last used, then it is capable of destroying up to $f(j)$ robots.
- So specifically, if it is used in the kth second, and it has been j seconds since it was previously used, then it will destroy $\min(x_k, f(j))$ robots. (After this use, it will be completely drained.)
- We will also assume that the EMP starts off completely drained, so if it is used for the first time in the jth second, then it is capable of destroying up to $f(j)$ robots.

The problem. Given the data on robot arrivals $x_1, x_2, ..., x_n$, and given the recharging function f, choose the points in time at which you're going to activate the EMP so as to destroy as many robots as possible.

i. Show that the following algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer.

```
Schedule-EMP(x_1, ..., x_n):
    Let j be the smallest number for which f(j) ≥ x_n
        (If no such j exists then set j = n)
    Activate the EMP in the nth second
    If n − j ≥ 1 then
        Continue recursively on the input x_1, ..., x_{n−j}
            (i.e., invoke Schedule-EMP(x_1, ..., x_{n−j}))
```

Give an efficient algorithm that takes the data on robot arrivals $x_1, ..., x_n$ and the recharging function f and returns the maximum number of robots that can be destroyed by a sequence of EMP activations.

vi) (Erickson 3.5) This set of exercises asks you to develop efficient algorithms to find optimal subsequences of various kinds. A subsequence is anything obtained from a sequence by extracting a subset of elements, but keeping them in the same order; the elements of the subsequence need not be contiguous in the original sequence. For example, the strings `C`, `DAMN`, `YAIOAI`, and `DYNAMICPROGRAMMING` are all subsequences of the string `DYNAMICPROGRAMMING`. **Each of these counts as one problem** [Hint: Exactly one of these problems can be solved in O(n) time using a greedy algorithm.]

i. Let `A[1 .. m]` and `B[1 .. n]` be two arbitrary arrays. A common subsequence of `A` and `B` is another sequence that is a subsequence of both `A` and `B`. Describe an efficient algorithm to compute the longest common subsequence of A and B.

ii. Let `A[1 .. m]` and `B[1 .. n]` be two arbitrary arrays. A common supersequence of `A` and `B` is another sequence that contains both `A` and `B` as subsequences. Describe an efficient algorithm to compute the shortest common supersequence of `A` and `B`.

iii. Call a sequence `X[1 .. n]` of numbers bitonic if there is an index `i` with $1 < i < n$, such that the prefix `X[1 .. i]` is increasing and the suffix `X[i .. n]` is decreasing. Describe an efficient algorithm to compute the longest bitonic subsequence of an arbitrary array `A` of integers.

iv. Call a sequence `X[1 .. n]` of numbers oscillating if `X[i] < X[i + 1]` for all even `i`, and `X[i] > X[i + 1]` for all odd `i`. Describe an efficient algorithm to compute the longest oscillating subsequence of an arbitrary array `A` of integers.

v. Describe an efficient algorithm to compute the shortest oscillating supersequence of an arbitrary array `A` of integers.

vi. Call a sequence `X[1 .. n]` of numbers convex if 2`X[i]` $<$ `X[i-1] + X[i+1]` for all `i`. Describe an efficient algorithm to compute the longest convex subsequence of an arbitrary array `A` of integers.

vii. Call a sequence `X[1 .. n]` of numbers weakly increasing if each element is larger than the average of the two previous elements; that is, 2`X[i]` $>$ `X[i-1] + X[i-2]` for all `i` > 2. Describe an efficient algorithm to compute the longest weakly increasing subsequence of an arbitrary array `A` of integers.

viii. Recall that a sequence `X[1 .. n]` of numbers is increasing if `X[i]` $<$ `X[i+1]` for all i. Describe an efficient algorithm to compute the longest common increasing subsequence of two given arrays of integers.

vii) (Erickson 3.6) A *shuffle* of two strings X and Y is formed by interspersing the characters into a new string, keeping the characters of X and Y in order. For example, the string BANANAANANAS is a shuffle of the strings BANANA and ANANAS in several different ways:

BANANAANANAS BANANAANANAS BANANAANANAS

i. Given three string A, B, and C of lengths m, n, and $m+n$ respectively, describe and analyze an efficient algorithm to determine if C is a shuffle of A and B.

ii. A **smooth** shuffle of X and Y is a shuffle of X and Y that never uses more than two consecutive symbols of either string. Describe and analyze an efficient algorithm to decide, given 3 strings A, B, and C as above, whether C is a smooth shuffle of A and B.

viii) (Erickson 3.8) Describe and analyze an efficient algorithm to find the length of the longest contiguous substring that appears both forward and backward in an input string T of length n. The forward and backward strings must not overlap. As some examples:

- Given the string ``ALGORITHM'', your output should be 0.
- Given the string ``RECURSION'', your output should be 1.
- Given the string ``REDIVIDE'', your output should be 2.

ix) (Erickson 3.9, sorta) A palindrome is any string that is exactly the same as its reversal, like ``I'', or ``RACECAR''.

i. Describe and analyze an efficient algorithm to determine the *longest* subsequence of a given string that is a palindrome.

ii. Describe and analyze an efficient algorithm to determine the *shortest* supersequence of a given string that is a palindrome.

x) (Algorithm Design 6.11) Suppose you're consulting for a company that manufactures PC equipment and ships it to distributors all over the country. For each of the next n weeks, they have projected *supply* s_i of equiment (measured in pounds), which has to be shipped by an air freight carrier.

Each week's supply can be carried by one of two air freight companies, A or B. Company A charges a fixed rate r per pound (so it costs $r \cdot s_i$ to ship a week's supply s_i). Company B makes contracts for a fixed amount of c per week, independent of the weight. However, contracts for company B must be must be made in blocks of 4 consecutive weeks at a time.

A schedule for the PC company is a choice of air freight company (A or B) for each of the n weeks, with the restriction that company B, whenever it is chosen, must be chosen for 4 consecutive weeks at a time. The *cost* of the schedule is the total amount paid to companies A and B.

Describe and analyze an efficient algorithm to determine the schedule of minimal cost.

xi) (Algorithm Design 6.20) Suppose it's nearing the end of the semester and you're taking n courses, each with a final project that still has to be done. Each project will be graded on the following scale: It will be assigned an integer number on a scale of 1 to $g > 1$, higher numbers being better grades. Your goal, of course, is to maximize your average grade on the n projects.

You have a total of $H > n$ hours in which to work on the n projects cumulatively, and you want to decide how to divide up this time. For simplicity, assume H is a positive integer, and you'll spend an integer number of hours on each project.

To figure out how best to divide up your time, you've come up with a set of functions $\{f_i : i = 1, 2, ..., n\}$ for each of your n courses; if you spend $h \leq H$ hours on the project for course i, you'll get a grade of $f_i(h)$. (You may assume that the functions f_i are nondecreasing: if $h < h'$, then $f_i(h) \leq f_i(h')$.)

So the problem is: Given these functions $\{f_i\}$, decide how many hours to spend on each project (in integer values only) so that your average grade, as computed according to the f_i, is as large as possible. In order to be efficient, the running time of your algorithm should be polynomial in n, g, and H; none of these quantities should appear as an exponent in your running time. You can assume that each f_i takes constant time to evaluate.

xii) Due to Dynamic Programming importance as a tool, there are millions of good questions out there for practice. I'm linking Leetcode's here, but if you want more practice the best ones can be found in the textbooks I usually pull from. **If you copy over the problem statement and do a full solution, any leetcode question can be counted as an additional solution here**, though I can't promise that their difficulty is above or below the caliber of difficulty on an exam.