

CS375 Weekly Assignment 6

Due April 7th 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. The following few questions concern the rod-cutting problem from class:

Rod Cutting

Input: A length n rod and an array $p[0, \dots, n]$ where $p[i]$ denotes the sell-price of a length i rod.

Output: A collection S of positive integers such that $\sum_{s \in S} s = n$ and $\sum_{s \in S} p[s]$ is maximized.

- (a) First, an algorithm that was proposed by a student last semester:

```
def OPT(n):
    if n <= 0:
        return []

    output = []

    i = index less than or equal to n that maximizes p[i] / i
    # so i maximizes price-per-unit-length

    remaining = n

    while remaining >= i:
        add i to output
        remaining -= i

    # here + is using list concatenation
    return output + OPT(remaining)
```

Show that this algorithm is wrong.

- (b) Let's change the problem a little: suppose we are only allowed to make at most k cuts for some integer $k \geq 0$. Develop a $\Theta(n^2k)$ algorithm to determine the best way to cut a length n rod using **at most** k cuts.

Trace through your algorithm on the input $p = [0, 8, 18, 26, 26, 36, 41, 49, 55, 63, 68, 77, 79, 84]$ and $k = 3$.

2. Pick 3 of the following and do them. You'll pick a few more next week.

- i) (Erickson 3.1) In a previous life, you worked as a cashier in the lost Antarctic 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 $A[1 \dots n]$ 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 $A[1 \dots n]$ and $B[1 \dots n]$ of characters, decide whether A and B can be partitioned into words at the same indices. For example, the strings BOTHEARTHANDSATURNPIN 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 $A[1 \dots n]$ and $B[1 \dots n]$ 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 $h_{i+1} > \ell_i + \ell_{i+1}$ 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  $\ell_1, \dots, \ell_n$  and  $h_1, \dots, h_n$  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,  $W = w_1, w_2, \dots, w_n$ , 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  $w_j, w_{j+1}, \dots, w_k$  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  $i$ th second,  $x_i$  robots arrive. Based on remote sensing data, you know this sequence  $x_1, x_2, \dots, x_n$  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  $k$ th 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  $j$ th second, then it is capable of destroying up to  $f(j)$  robots.

The problem. Given the data on robot arrivals  $x_1, x_2, \dots, x_n$ , 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.

- 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( $x_1, \dots, x_n$ ):

Let  $j$  be the smallest number for which  $f(j) \geq x_n$   
 (If no such  $j$  exists then set  $j = n$ )

Activate the EMP in the  $n$ th second

If  $n - j \geq 1$  then

Continue recursively on the input  $x_1, \dots, x_{n-j}$   
 (i.e., invoke Schedule-EMP( $x_1, \dots, x_{n-j}$ ))

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

- (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, YAIIOAI, 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.]
  - Let  $A[1 \dots m]$  and  $B[1 \dots n]$  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  $B$ .
  - Let  $A[1 \dots m]$  and  $B[1 \dots n]$  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  $B$ .
  - Call a sequence  $X[1 \dots n]$  of numbers bitonic if there is an index  $i$  with  $1 < i < n$ , such that the prefix  $X[1 \dots i]$  is increasing and the suffix  $X[i \dots n]$  is decreasing. Describe an efficient algorithm to compute the longest bitonic subsequence of an arbitrary array  $A$  of integers.
  - Call a sequence  $X[1 \dots n]$  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.
  - Describe an efficient algorithm to compute the shortest oscillating supersequence of an arbitrary array  $A$  of integers.
  - Call a sequence  $X[1 \dots n]$  of numbers convex if  $2X[i] < X[i-1] + X[i+1]$  for all  $i$ . Describe an efficient algorithm to compute the longest convex subsequence of an arbitrary array  $A$  of integers.
  - Call a sequence  $X[1 \dots n]$  of numbers weakly increasing if each element is larger than the average of the two previous elements; that is,  $2X[i] > X[i-1] + X[i-2]$  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  $X[1 \dots n]$  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.