Problem Set 7

This problem set is due in lecture on **Monday, November 13, 2006**. The homework lab for this problem set will be held 2–4 P.M. on Sunday, November 12, 2006, in room 32-124.

Reading: Chapter 15, 16.1–16.3, 22.1, and 23.

**Problem 7-1 is mandatory. Failure to turn in a solution will result in a serious and negative impact on your term grade!**

Both exercises and problems should be solved, but only the problems should be turned in. Exercises are intended to help you master the course material. Even though you should not turn in the exercise solutions, you are responsible for material covered in the exercises.

Mark the top of each sheet with your name, the course number, the problem number, your recitation section, the date and the names of any students with whom you collaborated.

You will often be called upon to “give an algorithm” to solve a certain problem. Your write-up should take the form of a short essay. A topic paragraph should summarize the problem you are solving and what your results are. The body of the essay should provide the following:

1. A description of the algorithm in English and, if helpful, pseudo-code.
2. At least one worked example or diagram to show more precisely how your algorithm works.
3. A proof (or indication) of the correctness of the algorithm.
4. An analysis of the running time of the algorithm.

Remember, your goal is to communicate. Full credit will be given only to correct solutions which are described clearly. Convoluted and obtuse descriptions will receive low marks.

---

**Exercise 7-1.** Do Exercise 15.4-5 on page 356 of CLRS.

**Exercise 7-2.** Do Exercise 16.1-3 on page 379 of CLRS.

**Exercise 7-3.** Do Exercise 22.1-5 on page 530 of CLRS.

**Exercise 7-4.** Do Exercise 23.1-5 on page 566 of CLRS.

**Exercise 7-5.** Do Exercise 23.2-4 on page 574 of CLRS.

**Exercise 7-6.** Do Exercise 23.2-5 on page 574 of CLRS.
Problem 7-1. Typesetting

In this problem you will write a program (real code that runs!) to solve the following typesetting scenario. **Because of the trouble you may encounter while programming, we advise you to START THIS PROBLEM AS SOON AS POSSIBLE.**

You are given an input text consisting of a sequence of \( n \) words of lengths \( \ell_1, \ell_2, \ldots, \ell_n \), where the length of a word is the number of characters it contains. Your printer can only print with its built-in Courier 10-point fixed-width font set that allows a maximum of \( M \) characters per line. (Assume that \( \ell_i \leq M \) for all \( i = 1, 2, \ldots, n \).) When printing words \( i \) and \( i + 1 \) on the same line, exactly one space character (blank) must be printed between the two words. In addition, any remaining space at the end of the line is padded with blanks. Thus, if words \( i \) through \( j \) are printed on a line, the number of extra space characters at the end of the line (after word \( j \)) is

\[
M - j + i - \sum_{k=i}^{j} \ell_k.
\]

There are many ways to divide a paragraph into multiple lines. To produce nice-looking output, we want a division that fills each line as much as possible. A heuristic that has empirically shown itself to be effective is to charge a cost of the cube of the number of extra space characters at the end of each line. To avoid the unnecessary penalty for extra spaces on the last line, however, the cost of the last line is 0. In other words, the cost \( \text{LINECOST}(i, j) \) for printing words \( i \) through \( j \) on a line is given by

\[
\text{LINECOST}(i, j) = \begin{cases} 
\infty & \text{if words } i \text{ through } j \text{ do not fit on a line,} \\
0 & \text{if } j = n \text{ (i.e., last line),} \\
(M - j + i - \sum_{k=i}^{j} \ell_k)^3 & \text{otherwise.}
\end{cases}
\]

The total cost for typesetting a paragraph is the sum of the costs of all lines in the paragraph. An optimal solution is a division of the \( n \) words into lines in such a way that the total cost is minimized.

(a) Argue that this problem exhibits optimal substructure.

(b) Recursively define the value of an optimal solution.

(c) Give an efficient algorithm to compute the cost of an optimal solution. Analyze the running time and space requirements of your algorithm.

(d) Write code (in any programming language you wish\(^1\)) to print an optimal division of the words into lines. The input to the program is a text file and the parameter \( M \). For simplicity, assume that a word in the text file is a maximal sequence of characters that are not spaces. Thus, a word is a substring strictly between two space characters or bounded by the beginning or end of the input.

We define a space according to the “POSIX locale”, i.e., the characters which are spaces are white space (‘ ‘), form-feed (‘ \f’), newline (‘ \n’), carriage return

---

\(^1\)Solutions will be written in C and Python.
Problem Set 7

Problem 7-1. Splitting a string

A character may be a space (\'\ \'), horizontal tab (\'\t\'), and vertical tab (\'\v\'). This definition is convenient in most programming languages:

- In Python, you can split a string \(s\) into words using \(words = s.split()\), and you can test whether a character \(c\) is a space using \(c.isspace()\).
- In Perl, you can split a string \($s\) into words using \(@words = split \/'\s+/'\, \$s\), and you can test whether a character \($c\) is a space using \($c = \~/'\s/'\).
- In Ruby, you can split a string \(s\) into words using \(words = s.split\), and you can test whether a character \(c\) is a space using \(c = \~/'\s/'\).
- In Java, you can split a string \(s\) into words using \(String[] words = s.split("\\s+")\) and you can test whether a character \(c\) is a space using \(java.lang.Character.isWhitespace(c)\).
- In C, you can iterate over words in \(s\) using
  
  ```c
  word = strtok(s, " \f\n\r\t\v");
  while ((word = strtok(NULL, " \f\n\r\t\v"))) {
    ...
  }
  ```
  (after \#include <string.h>), and you can test whether a character \(c\) is a space using \(isspace(c)\) (after \#include <ctype.h>),
- In Haskell, you can split a string \(s\) into words using \(words s\), and you can test whether a character \(c\) is a space using \(isSpace c\).
- In Emacs LISP, you can split a string \(s\) into words using \((split-string s)\), and you can test whether a character \(c\) is a space using \((string-match "\\s-" c)\).
- ...you get the idea.

You should hand in your source code and the output of sample executions of your program. Sample input text files are provided on the course website. For each of the four sample input files, you should hand in program output for \(M = 48\) and \(M = 75\). To help you debug your program, we have supplied the correct output with \(M = 60\) on the first sample input.

Problem 7-2. Minimizing waiting times

You’ve just landed an internship at the trendy new Algofrenzy restaurant that is looking for ways to improve customer satisfaction. Customers are able to view the Algofrenzy menu online prior to their visit, which allows them to place their order immediately upon arriving at the restaurant. The Algofrenzy management currently faces the following problem, which they would like you to help them solve. Every day, promptly at noon, several customers arrive simultaneously, all armed with their lunch orders. Unfortunately, the restaurant has only one chef at hand, and the chef can process only one customer’s order at a time. In what order should the customers be processed so as to minimize waiting times?
Specifically, the problem you are asked to solve is the following. Suppose that $n$ customers arrive simultaneously and start waiting to be served. (Assume that the restaurant was empty prior to their arrival.) Further, suppose that the preparation time required to fulfill the order placed by customer $i$ is $p_i > 0$ minutes. The waiting time $w_i$ of customer $i$ is the preparation time for this customer, $p_i$, plus the sum of the preparation times for the customers served before customer $i$. (For example, if there are three customers with preparation times $p_1 = 2$, $p_2 = 4$, $p_3 = 3$, and the customers are served in the order $(3, 1, 2)$, then the waiting times are $w_3 = 3$, $w_1 = 5$, $w_2 = 9$.) The goal is to find an order to process the customers that minimizes the average waiting time,

$$\frac{1}{n} \sum_{i=1}^{n} w_i.$$

(a) Argue that this problem exhibits optimal substructure.

(b) Argue that this problem exhibits the greedy-choice property.

(c) Give an efficient algorithm to compute an optimal order to serve the customers. Analyze the running time and space requirements of your algorithm.

(d) **Bonus Part:** *This part is optional. However, solving this part does not exempt you from solving the other parts of this question.*

Suppose now that not all $n$ customers know what they want to order when they arrive. Thus, although they arrive simultaneously, they do not all order immediately. Specifically, each customer takes some time, $d_i \geq 0$ minutes, to decide what to order; his/her order cannot be processed before that. Suppose also that, at any time, the chef may decide to suspend an order she is currently working on to work on another order, and then resume working on the first order at a later time; an order can be suspended any number of times. Let the waiting time $w_i$ be the total time customer $i$ has spent in the restaurant until receiving his/her order. Give an algorithm that schedules the orders (preparation tasks for the chef) so as to minimize the average waiting time in this new scenario.

You may assume that $p_i, d_i$ are all integers, and that the chef’s decisions to suspend/continue an order take place only at the beginning of each minute. Also assume that, between noon and the time at which all $n$ orders have been fulfilled, not a single minute is free, i.e. at every minute in between, there is at least one order waiting to be processed. Thus, if the total preparation time for the $n$ orders is $T = \sum_{i=1}^{n} p_i$, your algorithm must output a schedule $\beta : \{1, \ldots, T\} \rightarrow \{1, \ldots, n\}$ which tells the chef to work on order $\beta(t)$ during minute $t$. 