Problem Set 7 Solutions

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.

**Solution:** Note that \( \text{LINECOST}(i, j) \) is defined to be \( \infty \) if the words \( i \) through \( j \) do not fit on a line to guarantee that no lines in the optimal solution overflow. (This relies on the assumption that the length of each word is not more than \( M \).) Second, notice that \( \text{LINECOST}(i, j) \) is defined to be 0 when \( j = n \), where \( n \) is the total number of words; only the actual last line has zero cost, not the recursive last lines of subproblems,
which, since they are not the last line overall, have the same cost formula as any other line.

Now, consider an optimal solution of printing words 1 through \(n\). Let \(i\) be the index of the first word printed on the last line of this solution. Then typesetting of words \(1, \ldots, i - 1\) must be optimal. Otherwise, we could paste in an optimal typesetting of these words and improve the total cost of solution, a contradiction. Also note that the same cut-and-paste argument can be applied if we take \(i\) to be the index of the first word printed on the \(k\)th line, where \(2 \leq k \leq n\). Therefore this problem displays optimal substructure.

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

Solution: Let \(c(j)\) be the optimal cost of printing words 1 through \(j\). From part (a), it is clear that given the optimal \(i\) (i.e., the index of the first word printed on the last line of an optimal solution), we have

\[ c(j) = c(i - 1) + \text{LINE\textsc{Cost}}(i, j). \]

However, since we do not know what \(i\) is optimal, we need to consider every possible \(i\), so our recursive definition of the optimal cost is

\[ c(j) = \min_{1 \leq i \leq j} \{c(i - 1) + \text{LINE\textsc{Cost}}(i, j)\}. \]

To accommodate this recursive definition, we define \(c(0) = 0\).

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

Solution: We calculate the values of the array \(c\) from index 1 to \(n\), bottom up, which can be done efficiently since each \(c(k)\) for \(1 \leq k < j\) will be available by the time \(c(j)\) is computed. To keep track of the actual optimal arrangement of the words, we record an array \(p\), where \(p(k)\) is the \(i\) (in the recursive definition of \(c\)) which led to the optimal \(c(k)\). Then, after the arrays for \(c\) and \(p\) are computed, the optimal cost is \(c(n)\) and the optimal solution can be found by printing words \(p(n)\) through \(n\) on the last line, words \(p(p(n) - 1)\) through \(p(n) - 1\) on the next to last line, and so on.

A good optimization can be obtained by noticing that computing \(\text{LINE\textsc{Cost}}(i, j)\) takes in general \(O(j - i + 1)\) time because of the summation in the formula. However, it is possible to do this computation in \(O(1)\) time with some additional pre-processing. We create an auxiliary array \(L[0 \ldots n]\), where \(L[i]\) is a cumulative sum of lengths of words 1 through \(i\).
\[ L[0] = 0 \]
\[ L[i] = L[i-1] + \ell_i \equiv \sum_{k=1}^{i} \ell_k \]

Filling in this array takes \( O(n) \) time using recursion. In order to then compute \( \text{LINECOST}(i, j) \) in \( O(1) \) time, we modify the formula as follows:

\[
\text{LINECOST}(i, j) = \begin{cases} 
\infty & \text{if words } i \text{ through } j \text{ do not fit into a line}, \\
0 & \text{if } j = n \text{ (i.e. last line)}, \\
(M - j + i - (L[j] - L[i-1]))^3 & \text{otherwise}.
\end{cases}
\]

This algorithm uses \( \Theta(n) \) space for the arrays and runs in \( O(n^2) \) time, since each value of \( c \) takes up to \( n \) calculations as each value of \( i \) is considered. By noticing that at most \( \lfloor (M + 1)/2 \rfloor \) words can fit on a single line, we can reduce running time to \( O(nM) \)—another significant improvement—by considering only those \( i \) for which \( j - \lfloor (M + 1)/2 \rfloor + 1 \leq i \leq j \) when calculating each \( c(j) \).

(d) Write code (in any programming language you wish) 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 (‘\r’), 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 \( \text{words} = s\text{.split()} \), and you can test whether a character \( c \) is a space using \( c\text{.isspace()} \).
- In Perl, you can split a string \( \$s \) into words using \( @\text{words} = \text{split} \ '/\s+/' , \$s \), and you can test whether a character \( \$c \) is a space using \( \$c = \text{~} '/\s/' \).
- In Ruby, you can split a string \( s \) into words using \( \text{words} = s\text{.split} \), and you can test whether a character \( c \) is a space using \( c = \text{~} '/\s/' \).
- In Java, you can split a string \( s \) into words using
  \[
  \text{String[]} \ \text{words} = s\text{.split("\s+")}
  \]
  and you can test whether a character \( c \) is a space using \( \text{java.lang.Character.isWhitespace}(c) \).

1 Solutions will be written in C and Python.
• 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.

**Solution:**

On the 6.046 website, we provide a solution, written in C, and the correct results for
each of the sample inputs. Our solution represents a straightforward implementation
of the \( O(nM) \) algorithm described in part (c). The program takes as arguments the
name of the input file and a value for \( M \), reads the input words from the file, computes
the optimal cost, and then reconstructs an optimal solution, which is printed out.

**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 fulfil 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.

**Solution:**

Consider \(n\) customers with preparation times \(p_1, \ldots, p_n\). Let \(k\) be any customer for which there exists some optimal ordering in which customer \(k\) is served first. Suppose \(\sigma' : \{1, \ldots, n-1\} \rightarrow \{1, \ldots, k-1, k+1, \ldots, n\}\) is an optimal ordering of the remaining customers, dictating that those customers be served in the order \((\sigma'(1), \sigma'(2), \ldots, \sigma'(n-1))\). Then we claim that the ordering \(\sigma : \{1, \ldots, n\} \rightarrow \{1, \ldots, n\}\) defined by

\[
\sigma(i) = \begin{cases} 
  k & \text{if } i = 1 \\
  \sigma'(i) & \text{if } 2 \leq i \leq n 
\end{cases}
\]

is an optimal ordering of the \(n\) customers, dictating that the customers be served in the order \((\sigma(1), \sigma(2), \ldots, \sigma(n))\).

**Proof.** Assume, for the purpose of contradiction, that \(\sigma\) is not an optimal ordering of the \(n\) customers. Let \(\pi : \{1, \ldots, n\} \rightarrow \{1, \ldots, n\}\) be an optimal ordering in which customer \(k\) is served first (so that \(\pi(1) = k\)). Then the average waiting time \(W(\pi)\) under the ordering \(\pi\) must be less than the average waiting time \(W(\sigma)\) under the ordering \(\sigma\).

Let \(\pi' : \{1, \ldots, n-1\} \rightarrow \{1, \ldots, k-1, k+1, \ldots, n\}\) be the ordering of the remaining customers defined by \(\pi'(i) = \pi(i + 1)\) for all \(i \in \{1, \ldots, n-1\}\). Then we get

\[
W(\pi') - W(\sigma') = \frac{1}{n-1} \sum_{i=1}^{n-1} w_{\pi'(i)} - \frac{1}{n-1} \sum_{i=1}^{n-1} w_{\sigma'(i)}
\]

\[
= \frac{n}{n-1} \left[ \frac{1}{n} \left( p_k + \sum_{i=1}^{n-1} (p_k + w_{\pi'(i)}) \right) - \frac{1}{n} \left( p_k + \sum_{i=1}^{n-1} (p_k + w_{\sigma'(i)}) \right) \right]
\]

\[
= \frac{n}{n-1} \left[ \frac{1}{n} \sum_{i=1}^{n-1} w_{\pi(i)} - \frac{1}{n} \sum_{i=1}^{n-1} w_{\sigma(i)} \right]
\]

\[
= \frac{n}{n-1} (W(\pi) - W(\sigma))
\]

< 0,

a contradiction since \(\sigma'\) is an optimal ordering of \(\{1, \ldots, k-1, k+1, \ldots, n\}\).

Thus, \(\sigma\) must be an optimal ordering of the \(n\) customers.
(b) Argue that this problem exhibits the greedy-choice property.

Solution:
Consider $n$ customers with preparation times $p_1, \ldots, p_n$. We claim that there is an optimal ordering of the customers in which a customer with minimal preparation time is served first.

Proof. Let $i_{\text{min}} = \arg \min_{1 \leq i \leq n} p_i$. Let $\sigma : \{1, \ldots, n\} \to \{1, \ldots, n\}$ be an optimal ordering of the $n$ customers. If $\sigma(1) = i_{\text{min}}$, we are done. Suppose $\sigma(1) \neq i_{\text{min}}$; then we show that there is another optimal ordering $\sigma'$ such that $\sigma'(1) = i_{\text{min}}$. In particular, let $\sigma'$ be the ordering obtained by exchanging the positions of customers $\sigma(1)$ and $i_{\text{min}}$ in $\sigma$. Thus, if $i_{\text{min}} = \sigma(i^*)$, we have

$$
\sigma'(i) = \begin{cases} 
  i_{\text{min}} & \text{if } i = 1 \\
  \sigma(1) & \text{if } i = i^* \\
  \sigma(i) & \text{otherwise}
\end{cases}
$$

Note that the waiting times under ordering $\sigma$ are given by

$$
w_{\sigma(i)} = \sum_{j=1}^{i} p_{\sigma(j)},
$$

so that the average waiting time under $\sigma$ is

$$
W(\sigma) = \frac{1}{n} \sum_{i=1}^{n} w_{\sigma(i)} = \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{i} p_{\sigma(j)} = \frac{1}{n} \sum_{i=1}^{n} (n - i + 1) p_{\sigma(i)}.
$$

Now, we can relate the average waiting times under $\sigma$ and $\sigma'$ as follows:

$$
W(\sigma') - W(\sigma) = \frac{1}{n} \sum_{i=1}^{n} (n - i + 1)(p_{\sigma'(i)} - p_{\sigma(i)})
= \frac{1}{n} \left[ n(p_{\sigma'(1)} - p_{\sigma(1)}) + (n - i^* + 1)(p_{\sigma'(i^*)} - p_{\sigma(i^*)}) \right]
= \frac{1}{n} \left[ n(p_{i_{\text{min}}} - p_{\sigma(1)}) + (n - i^* + 1)(p_{\sigma(1)} - p_{i_{\text{min}}}) \right]
= \frac{1}{n} (i^* - 1)(p_{i_{\text{min}}} - p_{\sigma(1)})
\leq 0, \quad \text{since } i^* > 1 \text{ and } p_{i_{\text{min}}} \leq p_{\sigma(1)}.
$$

Since $\sigma$ is an optimal ordering, we cannot have $W(\sigma') < W(\sigma)$. Therefore, by the above relation, we must have $W(\sigma') = W(\sigma)$. Thus, $\sigma'$ is an optimal ordering in which customer $i_{\text{min}}$ is served first.
(c) Give an efficient algorithm to compute an optimal order to serve the customers. Analyze the running time and space requirements of your algorithm.

Solution:
It follows from the optimal substructure and greedy-choice properties proved in parts (a) and (b) above that an optimal ordering of \( n \) customers is one in which customers are served in increasing order of their preparation times. Thus an algorithm to compute an optimal ordering need only sort by preparation times; this can be done in \( O(n \log n) \) time and \( O(n) \) space (using heapsort, for example).

Alternative Proof of Correctness
Note that in the case of the above problem, it is also possible to prove directly (without proving separately the optimal substructure and greedy-choice properties) that an optimal ordering of \( n \) customers is one in which customers are served in increasing order of their preparation times.

In particular, suppose (for contradiction) that \( \sigma \) is an optimal ordering which does not order the customers in increasing order of their preparation times. Then there exist some \( i < j \) such that \( p_{\sigma(i)} > p_{\sigma(j)} \). We construct a new ordering \( \sigma' \) from \( \sigma \) that interchanges the positions of customers \( \sigma(i) \) and \( \sigma(j) \) as follows:

\[
\sigma'(k) = \begin{cases} 
\sigma(i) & \text{if } k = j \\
\sigma(j) & \text{if } k = i \\
\sigma(k) & \text{otherwise}
\end{cases}
\]

Then we have
\[
W(\sigma') - W(\sigma) = \frac{1}{n} \sum_{k=1}^{n} \left[ (n - k + 1)(p_{\sigma'(k)} - p_{\sigma(k)}) \right]
\]
\[
= \frac{1}{n} \left[ (n - i + 1)(p_{\sigma'(i)} - p_{\sigma(i)}) + (n - j + 1)(p_{\sigma'(j)} - p_{\sigma(j)}) \right]
\]
\[
= \frac{1}{n} \left[ (n - i + 1)(p_{\sigma(j)} - p_{\sigma(i)}) + (n - j + 1)(p_{\sigma(i)} - p_{\sigma(j)}) \right]
\]
\[
= \frac{1}{n} (j - i)(p_{\sigma(j)} - p_{\sigma(i)}) < 0,
\]
a contradiction.

(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
Problem Set 7 Solutions

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 \( \beta : \{1, \ldots, T\} \to \{1, \ldots, n\} \) which tells the chef to work on order \( \beta(t) \) during minute \( t \).

Solution:
We can show that the optimal scheduling algorithm is one where the chef always works on an order which is closest to completion. This scheduling algorithm is sometimes referred to as \( \text{SRPT} \), for shortest remaining processing time.

To prove that \( \text{SRPT} \) generates an optimal schedule, we first define some notation for the processing time remaining for a given order at any given time. In particular, for each \( i \in \{1, \ldots, n\} \) and \( t \in \{1, \ldots, T\} \), let \( q_i(t; \beta) \) denote the remaining processing time for order \( i \) at time \( t \) under schedule \( \beta \). If order \( i \) is not yet available at time \( t \), we define \( q_i(t; \beta) \) to be infinity. That is,

\[
q_i(t; \beta) = \begin{cases} 
p_i - \sum_{k=1}^{t-1} \text{I}(\beta(k) = i) & \text{if } d_i < t \\
\infty & \text{otherwise}
\end{cases}
\]

where \( \text{I}(\beta(k) = i) \) is 1 if \( \beta(k) = i \) and 0 otherwise.

Now, suppose (for contradiction) that \( \beta \) is an optimal schedule that does not always work on the job with shortest remaining processing time.

Then there exists a time \( t_0 \) when the chef works on job \( j \), i.e., \( \beta(t_0) = j \), even though some other job \( i \) with shorter processing time is available, i.e., \( q_i(t_0; \beta) < q_j(t_0; \beta) \).

Let \( S_\beta = \{ t \geq t_0 \mid \beta(t) = i \text{ or } \beta(t) = j \} \) be the sequence of time steps \( t \geq t_0 \) in \( \beta \) where the chef is working on either \( i \) or \( j \)’s order. We know that \( |S_\beta| = q_i(t_0; \beta) + q_j(t_0; \beta) \), and at time \( t_0 \), the chef works on order \( j \). If \( w_i(\beta) \) and \( w_j(\beta) \) are the finishing times of jobs \( i \) and \( j \) under schedule \( \beta \), they also belong to \( S_\beta \). Let \( t_{\max} = \max\{w_i(\beta), w_j(\beta)\} \).

Now, we construct a new schedule \( \beta' \) that is identical to \( \beta \) at times \( t \not\in S_\sigma \), but at times \( t \in S_\beta \) it works on \( i \) first before starting on \( j \). In particular, let \( t_1 \) be the \( q_i(t_0; \beta) \)-th element in \( S_\beta \); time \( t_1 \) represents the earliest possible time when order \( i \) could have been completed if, starting at time \( t_0 \), we worked on order \( i \) first whenever we worked
on \(i\) or \(j\) in the original schedule. Then we have,

\[
\beta'(t) = \begin{cases} 
\beta(t) & \text{if } t \notin S_\beta \\
i & \text{if } t \in S_\beta \text{ and } t \leq t_1 \\
j & \text{if } t \in S_\beta \text{ and } t > t_1
\end{cases}
\]

Now we show that \(W(\beta') < W(\beta)\), i.e., that \(\beta'\) has a smaller average waiting time than \(\beta\), leading to a contradiction. Since the waiting times of customers other than \(i\) and \(j\) remain the same in \(\beta'\) as they were in \(\beta\), we have

\[
W(\beta') - W(\beta) = \frac{1}{n} [(w_i(\beta') + w_j(\beta')) - (w_i(\beta) + w_j(\beta))] \\
= \frac{1}{n} [(t_1 + t_{\max}) - (w_i(\beta) + w_j(\beta))] .
\]

Now, since in the original schedule \(\beta\), the chef worked on order \(j\) at time \(t_0\), we have \(w_i(\beta) > t_1\). In addition, since the processing time remaining for order \(j\) at time \(t_0\) was greater than that remaining for order \(i\), we also have \(w_j(\beta) > t_1\). Thus we have

\[
w_i(\beta) + w_j(\beta) = \max\{w_i(\beta), w_j(\beta)\} + \min\{w_i(\beta), w_j(\beta)\} > t_{\max} + t_1 ,
\]

which from the above leads to \(W(\beta') < W(\beta)\), a contradiction.