Introduction to Algorithms Octoberber 31, 2003
Massachusetts Institute of Technology 6.046J/18.410J
Professors Shafi Goldwasser and Silvio Micali Handout 20

Problem Set 5 Solutions

Problem 2-1. Spam and Customer Service.

You run a meat-processing plant in Cambridgeport. Unfortunately, due to a mixup with your suppliers in Peoria, you have accidentally produced a batch of spoiled sausages. When your customer service desk opens at 8am, there are n customers waiting in the lobby to return the rotten sausages. Say that customer i has s_i sausages, and that all the s_i's are distinct. Your underpaid customer service operator requires $k \cdot s_i$ minutes to handle customer i's complaint, for some constant k. You cannot "partially process" a complaint—all of customer i's s_i sausage returns must be handled together—and only one customer can be handled at a time.

Because you want to maintain the golden reputation of your company, you have decided to pay one dollar to each customer i for every minute before i's complaint is completely handled. For example, if customer 3 is handled first and customer 5 is handled second, you pay ks_3 dollars to customer 3 and $k(s_3 + s_5)$ dollars to customer 5. Give an efficient algorithm to find the order in which the customers' complaints should be processed so that the total cost to the company is minimized. Prove that your algorithm is optimal and analyze its running time.

Solution: Sort the s_i's in increasing order and handle the complaints in that order. The running time is $O(n \log n)$.

Now we consider correctness. Observe that there is never a reason to have any "idle time" in the schedule; removing any such holes from the schedule strictly reduces the cost of the solution.

Let $c := c_1, c_2, \ldots, c_n$ be an optimal schedule (i.e., person c_1 is handled first, then c_2, and so on). Suppose that our algorithm is not optimal. Then there must be an i and j so that $s_{c_i} > s_{c_j}$ but $i < j$. We claim that the schedule

$$c' := c_1, \ldots, c_{i-1}, c_j, c_{i+1}, \ldots, c_{j-1}, c_i, c_{j+1}, \ldots, c_n$$

resulting from swapping c_i and c_j reduces our costs. This contradicts the optimality of c, and implies that our algorithm is optimal.

To prove the claim, we will show that subtracting the cost of c' from the cost of c yields a positive number, which establishes that c' is better than c.

$$\begin{aligned}
& [\text{cost of } c] - [\text{cost of } c'] \\
= \;& \sum_{\ell=1}^{n} \text{payment to customer } \ell \text{ under } c - \sum_{\ell=1}^{n} \text{payment to customer } \ell \text{ under } c' \\
= \;& \sum_{\ell=1}^{n} \sum_{p=1}^{\ell} s_{c_p} - \sum_{\ell=1}^{n} \sum_{p=1}^{\ell} s_{c'_p}
\end{aligned}$$

$$
\begin{aligned}
&= \sum_{\ell=1}^{n}(n-\ell+1)\cdot s_{c_\ell} - \sum_{\ell=1}^{n}(n-\ell+1)\cdot s_{c'_\ell} \\
&= (n-i+1)\cdot[s_{c_i} - s_{c'_i}] + (n-j+1)\cdot[s_{c_j} - s_{c'_j}] \\
&= (n-i+1)\cdot[s_{c_i} - s_{c_j}] + (n-j+1)\cdot[s_{c_j} - s_{c_i}] \\
&= (n-i+1-n+j-1)\cdot[s_{c_i} - s_{c_j}] \\
&= (j-i)\cdot[s_{c_i} - s_{c_j}] \\
&> 0.
\end{aligned}
$$

Problem 2-2. Splitsville.

Let x be a string $x_1, x_2, \ldots, x_n$, where $x_i \in \Sigma$ for a finite *alphabet* Σ.

A *split* of x consists of two strings $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$ and $x_{j_1}, x_{j_2}, \ldots, x_{j_\ell}$, where

(i) $i_1 < i_2 < \cdots < i_k$;

(ii) $j_1 < j_2 < \cdots < j_\ell$;

(iii) $\{i_1, \ldots, i_k\} \cap \{j_1, \ldots, j_\ell\} = \emptyset$; and

(iv) $\{i_1, \ldots, i_k\} \cup \{j_1, \ldots, j_\ell\} = \{1, \ldots, n\}$.

For example, the following are splits of `banana`:

```
bnn        aaa
ban        ana
banaa      n
```

Give an algorithm to determine if two strings y and z form a *split* of x. You algorithm should take as input three strings, x, y, and z, where x is a string of n symbols from the alphabet Σ. If strings y and z form a split of x, it should output "yes". Otherwise, it should output "no". Prove that your algorithm is correct and analyze its running time. (Your algorithm should run in time $O(n^c)$ for some constant c.)

Solution: First check that $|y| + |z| \leq |x| = n$. This can be done in time $O(n)$. We now use dynamic programming. For $0 \leq i \leq |y|$ and $0 \leq j \leq |z|$, we set

$$
A[i,j] := \begin{cases} 1 & \text{if } y_{1,\ldots,i} \text{ and } z_{1,\ldots,j} \text{ are a split of } x_{1,\ldots,i+j} \\ 0 & \text{otherwise.} \end{cases}
$$

1.For every j, $0 \leq j \leq |z|$, set $A[0,j] := 1$ if $z_{1\ldots j} = x_{1\ldots j}$, and $A[0,j] := 0$ otherwise.

2.For every i, $0 \leq i \leq |y|$, set $A[i,0] := 1$ if $y_{1\ldots i} = x_{1\ldots i}$, and $A[i,0] := 0$ otherwise.

3.For every i, j, $1 \le i \le |y|$ and $1 \le j \le |z|$, set $A[i, j] = 1$ if either (i) $y_i = x_{i+j}$ and $A[i-1, j] = 1$, or (ii) $z_j = x_{i+j}$ and $A[i, j-1]$.

4.Return "yes" if and only if $A[|y|, |z|] = 1$.

Running time is obviously $O(|z||y|) = O(n^2)$. For correctness, observe that if $y_{1...i}$ and $z_{1...j}$ are a split of $x_{1...i+j}$, then the last character of $x_{1...i+j}$ must come from one of the two strings, and the remainder of that string plus the other string must be a split of $x_{1...i+j-1}$.

Problem 2-3. A Maki a Day ...

To maintain your health, you have decided that you must eat a sushi meal a day for the remaining n days, $d_1, d_2, \ldots, d_n$, of the fall semester. You have exactly two choices for acquiring a sushi meal:

1. On day d_i, you can buy a sushi meal (for consumption on day d_i) from the supermarket at price p_i. These prices are announced in advance, i.e. on day d_1.
2. On day d_i, you can place an order with `bulk-sushi.com` to have a sushi meal delivered to you every evening for days $d_i, d_{i+1}, d_{i+2}, \ldots, d_{i+11}$. The cost of this contract is Q. Note that Q is fixed and does not vary from day to day.

Since sushi is highly perishable, you cannot "stockpile" sushi meals; you must eat each sushi meal on the day that you get it. Also, because you hate to waste food and because you can only stomach exactly one sushi meal per day, you may not get two sushi meals on the same day. For example, you cannot order from `bulk-sushi.com` three days after placing a previous `bulk-sushi.com` order. Finally, you are not allowed to order from `bulk-sushi.com` on day d_i if there are less than 12 days remaining (including day d_i), i.e. you can not have leftover sushi meals at the end of the n day period.

The problem is to find the cost of the optimal sushi-ordering schedule.

(a) Consider the following greedy algorithm for this problem. This algorithm takes as input Q, the price of 12 sushi meals from `bulk-sushi.com`, and the prices $p_1, p_2, \ldots p_n$, the price of a sushi meal from the supermarket on each of the n days. At day d_i, it checks if the total cost of buying a sushi meal from the supermarket for that day and each of the next 11 days (i.e. days d_i through day d_{i+11}) is less than Q. If so, it buys a sushi meal for day d_i at price p_i and goes on to the next day. Otherwise, it orders 12 sushi meals (for day d_i through day d_{i+11}) from `bulk-sushi.com` at price Q and goes on to day d_{i+12}. Pseudocode for this algorithm is provided below.

BUYSUSHIGREEDILY$(Q, p_1, \ldots, p_n)$
Let $cost = 0$.
Go from day d_1 to day d_n.
On day d_i:
If Q is greater than the sum of p_i through p_{i+11} or if there are fewer than 12 days (meals) left,
    Then buy a sushi meal from the supermarket, let $cost \leftarrow cost + p_i$ and go to day d_{i+1}.
Else
    Place an order with `bulk-sushi.com`, let $cost \leftarrow cost + Q$ and go to day d_{i+12}.
Output $cost$.

Prove that BUYSUSHIGREEDILY does not produce an optimal solution to the problem.

Solution: For example, if $Q = 10$ and $p = \{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1000\}$, then the stated algorithm will buy from `bulk-sushi.com` on day d_1 and from the supermarket on day d_{13}, and thus pay $10 + 1000$ total, where as buying from the supermarket on day d_1 and then `bulk-sushi` on day d_2 costs only $1 + 10 = 11$.

(b) Give a dynamic programming algorithm to find the cost of an optimal sushi-ordering schedule. Prove your algorithm is correct and analyze its running time.

Solution: Let $Opt(k)$ denote the cost of an optimal ordering schedule for the first k days. Recall that *we cannot have leftover `bulk-sushi` sushi meals at the end.* In other words, we cannot have made a bulk purchase in the last 11 of these k days.

Observe that

$$Opt(k) = \min(p_k + Opt(k-1), Q + Opt(k-11)).$$

since any optimal schedule must have been optimal up to the start of our last purchase. Note that $Opt(0) = 0$, and $Opt(k) = \sum_{i=1}^{k} p_i$ for all $k \leq 11$.

Our dynamic programming algorithm fills in the $Opt(\cdot)$ table from left to right in $O(n)$ time, and we then return $Opt(n)$. Correctness follows from the above argument.

(c) Modify your algorithm from part **(b)** to output an actual optimal sushi-ordering schedule.

Solution: Let $W(k) = bulksushi$ denote that the optimal schedule finishing at day d_k (as in part **(b)**, without leftover sushi) gets its sushi from `bulk-sushi.com` for the last 12 days of the schedule, and $W(k) = supermarket$ if it gets day d_k's sushi from the supermarket. Then we modify the algorithm as follows:

1. Set $Opt(0) = 0$, and $Opt(k) = \sum_{i=1}^{k} p_i$ for all $k \leq 11$.
   Set $W(k) = supermarket$ for all $k \leq 11$.

2. For $k = 12$ to n:
   Set $Opt(k) = \min(p_k + Opt(k-1), Q + Opt(k-11))$.
   If $p_k + Opt(k-1) < Q + Opt(k-11)$
   then $W(k) := supermarket$
   else $W(k) := bulksushi$.
3. Set $i = n$.
   Repeat until $i = 0$:
   Output "On day d_i, buy from $W(i)$".
   If $W(i) = supermarket$, then $i := i - 1$, else $i := i - 12$.

Problem 2-4. Printing Neatly.

You are given a sequence of words $w_1, w_2, \ldots, w_n$, where word w_i has length ℓ_i. You want to lay out these words neatly in lines of total length L each, by choosing "nice" line breaks. Each word contains its own whitespace.

A neat line is one that contains close to L characters (without going over). More precisely, the *badness*—this is actually a technical term in LaTeX—of a line $w_i, w_{i+1}, \ldots, w_j$ is given by $badness := L - \sum_{k=i}^{j} \ell_k$. The badness *must* be nonnegative; all lines must contain at most L characters.

For example with $L = 25$, one could lay out the following definition from Ambrose Bierce's *The Devil's Dictionary* in these ways:

```
HATRED, n. A sentiment<->| badness 3
appropriate to the<----->| badness 7
occasion of another's<-->| badness 4
superiority.<----------->| badness 13
```

or

```
HATRED, n. A<----------->| badness 13
sentiment appropriate to>| badness 1
the occasion of<-------->| badness 10
another's superiority.<->| badness 3
```

(a) The badness of a paragraph is the sum of the badnesses of each of the lines of the paragraph, except the last. Give the most efficient algorithm you can to lay out the given words $w_1, \ldots, w_n$ in a single paragraph in a way that minimizes the badness of the paragraph. Prove your algorithm is correct and analyze its running time.

Solution: A greedy algorithm is optimal: just start placing words until placing the next one would make the line exceed L characters, then go to the next line and continue.

We claim that this is optimal. Let $W := \sum_{i=1}^{n} \ell_n$ be the total length of the words in question. Consider any solution that produces a paragraph with lines $L_1, L_2, \ldots, L_p, L_{p+1}$ for some particular p. Then the badness of the paragraph is given by:

$$\begin{aligned}\sum_{i=1}^{p} \text{badness of line } L_i &= \sum_{i=1}^{p} \left(L - \sum_{w_j \in L_i} \ell_j \right) \\ &= Lp - \sum_{w_j \in L_1, \ldots, L_p} \ell_j \\ &= Lp - W + \sum_{w_j \in L_{p+1}} \ell_j.\end{aligned}$$

Thus the optimal algorithm must place as many words as possible in the first p lines of the paragraph. A moment's reflection reveals that to maximize the number of words in the first t lines, one must (1) maximize the number of words in the first $t-1$ lines, and then (2) add as many words as possible to tth line. If (1) were not true, then the tth line would have to hold the "extra" left-over words, and thus could only at most as many words as if (1) were true. Thus the greedy algorithm described above is optimal.

The running time is straightforwardly $O(n)$: for each word, we place it on the same line if it fits, and start a new line if not. This is a constant-time operation per word if we keep track of the number of characters that we have already output in the current line.

(b) Consider the following alternative definition: the badness of a paragraph is the sum of the *cubes* of the badnesses of all lines by the paragraph, except the last. Is your algorithm from part **(a)** still optimal? Give a proof or a counterexample.

Solution: No, the algorithm from part **(a)** does not work. If the word lengths are $5, 3, 2, 9$ and the line length is 9, then the greedy algorithm will output

line	length	badness	badness^3
`55555333 |`	8	1	1
`22 |`	2	7	343
`999999999|`	9	0	0
			total: 344

On the other hand, the following layout has better badness:

line	length	badness	badness^3
`55555 |`	5	4	64
`33322 |`	5	4	64
`999999999|`	9	0	0
			total: 128

(c) Consider a third definition: the badness of a paragraph is the *maximum* badness of any line in the paragraph other than the last. Give a dynamic programming algorithm to minimize the badness of a paragraph. Prove your algorithm is correct and analyze its running time.

Solution: Let $Opt(k)$ denote the cost of laying out words $w_k, w_{k+1}, \ldots, w_n$ optimally. That is, consider a paragraph of just the words $w_k, \ldots, w_n$; $Opt(k)$ is the minimum achievable badness for such a paragraph.

1. For all k such that $w_k, \ldots, w_n$ all fit on one line (i.e., $\sum_{i=k}^{n} \ell_i \leq L$), we set $Opt(k) := 0$. This is correct because the last line of any paragraph does not contribute to the badness of the paragraph.
2. For all smaller k, let x be the last word that could fit on the same line as k, i.e.,

$$\sum_{i=k}^{x} \ell_i \;\leq\; L \;<\; \sum_{i=k}^{x+1} \ell_i .$$

   Then we set

$$Opt(k) := \min_{k \leq y \leq x} \max(Opt(y+1), L - \sum_{i=k}^{y} \ell_i).$$

   Clearly $\max(Opt(y+1), L - \sum_{i=k}^{y} \ell_i)$ is the badness of the paragraph formed by taking the first line as $w_k, \ldots, w_y$ and then optimally laying out the remaining words. Thus choosing the optimal y here yields an optimal layout for words $w_k, \ldots, w_n$.
3. Return $Opt(1)$.

We have argued correctness above. As for the running time, step two takes at most $O(x-k)$ time, but we have no particular *a priori* bound on this value. We can (possibly loosely) upper-bound the running time by $O(n^2)$, since we execute step 2 at most n times and there are at most n candidates for the end of the first line that we consider in that step.