Fundamental Algorithms, Assignment 7

Solutions

1. Set \( W = \lfloor \sqrt{N} \rfloor \). We are given \( \text{PRICE}[I], 1 \leq I \leq W \), the price of a rod of length \( I \). Give a program that will output the optimal revenue for a rod of length \( N^2 \) and give the time, in \( \Theta \)-land, of the algorithm. Use an auxiliary array \( R[J], 0 \leq J \leq N^2 \). You may not use the term \( \text{MAX} \) in your program. Explain, in clear words, how your program is working. (You can use \( \text{MAX} \) in your explanations.) Use an auxiliary array \( R[J], 0 \leq J \leq N^2 \).

Solution: The idea is that

\[
R[J] = \max[\text{PRICE}[I] + R[J - I]]
\]

where, critically, \( I \) ranges over \( 1 \leq I \leq \min[I, W] \).

\( R[0] = 0; R[1] = \text{PRICE}[1] \) (* initialization *)

FOR \( J = 2 \) to \( N^2 \) (*here we calculate \( R[J] \) given previous values*)

\( S = J; \) IF \( W \leq S \) THEN \( S = W \) (*so \( S = \min(J, W) \))

\( TEMP = 0 \) (*initializing to find max*)

FOR \( I = 1 \) TO \( S \); IF \( \text{PRICE}[I] + R[J - I] \geq TEMP \) THEN \( TEMP \leftarrow \text{PRICE}[I] + R[J - I]; \) ENDFOR (* \( TEMP \) becomes the maximal value *)

\( R[J] \leftarrow TEMP \)

ENDFOR (* for \( J \) *)

RETURN \( R[N^2] \).

The outer loops goes \( 1 \leq J \leq N^2 \). So for the time we need to add the times for the inner loop over \( J \). The time for the inner loop is basically \( \min(J, W) \). There are two ranges. While \( 1 \leq J < W \) this is \( J \) steps and so it adds to \( 1 + 2 + \ldots + (W - 1) \sim W^2/2 = \Theta(N) \). While \( W \leq J \leq N^2 \) this is \( W \) steps so it adds to \( W(N^2 - W) = \Theta(N^{2.5}) \). The total is the sum which is dominated by the second range so the time is \( \Theta(N^{2.5}) \).

2. (*) Suppose that the Huffman Code for \( \{ v, w, x, y, z \} \) has 0 or 1 as the code word for \( z \). Prove that the frequency for \( z \) cannot be less than \( \frac{1}{3} \). (Hint: Consider the situation in implementing the Huffman code when there are three letters remaining.) Give an example where the frequency for \( z \) is 0.36 and \( z \) does get code word 0 or 1.

Solution: First Part. If \( z \) has code word 0 or 1 then at the penultimate step there would be three nodes \( z, \alpha, \beta \) whose frequencies added to 1. With \( z \) having frequency less than \( \frac{1}{3} \) at least one of \( \alpha, \beta \) must have
frequency greater than \( \frac{1}{3} \). Thus the penultimate step would “join” \( z \) and one of \( \alpha, \beta \), and so \( z \) would not at the end get a code word of length one. **Second Part.** There are many correct answers here but perhaps the easiest is if \( v, w, x, y \) all have frequency 0.16. In the first three steps \( v, w \) create \( \alpha \) with frequency 0.32; \( x, y \) create \( \beta \) with frequency 0.32 and, as 0.36 > 0.32, \( \alpha, \beta \) create \( \gamma \) with frequency 0.64 and finally \( \gamma, z \) join.

3. Suppose, in the Activity Selector problem, we instead select the last activity to start that is compatible with all previously selected activities. Describe how this approach works, write a program for it (pseudocode allowed) and prove that it yields an optimal algorithm. **Solution:** This approach is symmetric to the one presented in the textbook. It is a greedy solution in that at each point, we’re selecting the last activity to start, and recursing down to the single subproblem of finding the optimal solution for all remaining activities compatible with the ones already scheduled. We can give the recursive algorithm as follows:

1. **RECURSIVE-ACTIVITY-SELECTOR**\((s, f, i)\)
   2. \( m \leftarrow i - 1 \)
   3. **while** \( m > 0 \) and \( s_i < f_m \)
   4. **do** \( m \leftarrow m - 1 \)
   5. **if** \( m > 0 \)
   6. **then return** \( \{a_m\} \cup \text{RECURSIVE-ACTIVITY-SELECTOR} (s, f, m) \)
   7. **else return** \( \emptyset \)

We initially run this algorithm with \( i = n \) where \( n \) is the number of tasks. This is a greedy algorithm, in that we’re decomposing the problem recursively into a single optimal subproblem. We can rigorously prove that this gives us an optimal solution by induction on the number of activities. But it is easier to note that a dynamic programming solution to this problem would yield an optimal result, and, as we saw in the book, there are two key observations we can use on the general recurrence of this problem that show that the greedy solution is equivalent to the dynamic programming solution. The recurrence for this solution is given in the book. We note the following two observations (and prove them):
Consider any nonempty subproblem $S_{ij}$, and let $a_m$ be the activity in $S_{ij}$ with the latest start time:

$$s_m = \max\{s_k : a_k \in S_{ij}\}$$

1 Activity $a_m$ is used in some maximum-size subset of mutually compatible activities of $S_{ij}$.

[Pf] Suppose that $A_{ij}$ is a maximum-size subset of mutually compatible activities of $S_{ij}$. We also suppose that the activities in $A_{ij}$ are ordered in monotonically increasing order of starting time. Let $a_k$ be the last activity in $A_{ij}$. If $a_k = a_m$, we’re done, since we’ve shown that $a_m$ is used in constructing the schedule. Otherwise, we construct the subset $A'_{ij} = A_{ij} - \{a_k\} \cup \{a_m\}$. We know the activities in the subset are disjoint, since $a_k$ is the last activity to start, and $s_m \geq s_k$. The number of activities in the subset are the same, so it is a maximum-size subset of activities that includes $a_m$.

2 The subproblem $S_{mj}$ is empty, so that choosing $a_m$ leaves the subproblem $S_{im}$ as the only one that may be nonempty.

[Pf] Suppose that $S_{mj}$ is nonempty, so there is some activity $a_k$ such that $f_m \leq s_k < f_k \leq s_j < f_j$. Then, $a_k$ is also in $S_{ij}$, which has a later start time than $a_m$, contradicting our initial assumption.

4. Students (professors too!) often come up with very clever ideas for optimization programs. The problem (often!) is that they (sometimes, but that is enough) give the wrong answer. Here are three approaches and your problem, in each case, is to give an example where it yields the wrong answer.

(a) Pick the activity of the shortest duration from amongst those which do not overlap previously selected activities.

(b) Pick the activity which overlaps the fewest other remaining activities from amongst those which do not overlap previously selected activities.

(c) Pick the activity with the earliest start time from amongst those which do not overlap previously selected activities.

Solution: One example to show that the approach of selecting the activity of least duration does not yield an optimal solution is the set
of tasks \( a_1 = (5, 7) \), \( a_2 = (1, 6) \), \( a_3 = (6, 10) \). \( a_1 \) is selected first, but this locks out the other two, which clearly comprise the optimal solution.

An example to show that the strategy of choosing the activities that overlap the fewest other remaining activities is the following set of tasks: \( a_1 = (0, 1) \), \( a_2 = (1, 3) \), \( a_3 = (3, 5) \), \( a_4 = (5, 6) \), \( a_5 = (0, 2) \), \( a_6 = (0, 2) \), \( a_7 = (2, 4) \), \( a_8 = (4, 6) \), \( a_9 = (4, 6) \). \( a_1, a_4, \) and \( a_6 \) have two overlapping activities, while the others have three. This means that \( a_1, a_4, \) and \( a_6 \) will be the activities selected, but the optimal solution is \( a_1, a_2, a_3, a_4 \).

An example to show that the strategy of choosing the compatible remaining activities with the earliest start time is the following set of tasks: \( a_1 = (1, 4) \), \( a_2 = (4, 5) \), \( a_3 = (2, 3) \), \( a_4 = (3, 4) \). \( a_1 \) will be selected first, followed by \( a_2 \), when clearly, the optimal solution consists of \( a_3, a_4, \) and \( a_2 \).

5. (a) What is an optimal Huffman code for the following code when the frequencies are the first eight Fibonacci number?

\[
\begin{align*}
a &: 1, b &: 1, c &: 2, d &: 3, e &: 5, f &: 8, g &: 13, h &: 21
\end{align*}
\]

(b) The Fibonacci sequence is defined by initial values 0, 1 with each further term the sum of the previous two terms. Generalize the previous answer to find the optimal code when there are \( n \) letters with frequencies the first \( n \) (excluding the 0) Fibonacci numbers.

Give a nice picture of the tree.

Solution:
minimal \( a, b \) have parent \( i : 2 \)
minimal \( i, c \) have parent \( j : 4 \)
minimal \( j, d \) have parent \( k : 7 \)
minimal \( k, e \) have parent \( l : 12 \)
minimal \( l, f \) have parent \( m : 20 \)
minimal \( m, g \) have parent \( n : 34 \)
minimal \( n, h \) have parent \( o : 55 \) and \( o \) is the root
So \( h : 0; g : 10; f : 110; e : 1110; d : 11110; c : 111110; b : 1111110; a : 1111111 \)
(Of course, 0, 1 can be reversed at any spot.)
The tree is a long line. This generalizes to the first \( n \) Fibonacci though proving it takes some nice properties of Fibonacci!

6. Suppose that in implementing the Huffman code we weren’t so clever as to use Min-Heaps. Rather, at each step we found the two letters of
minimal frequency and replaced them by a new letter with frequency their sum. (That is, use the “standard” method to find the minimum of a set of numbers and apply it twice.) How long would that algorithm take, in Thetaland, as a function of the initial number of letters $n$.

Solution: Finding the minimum of $s$ numbers takes time $O(s)$. So when there were $s$ letters we’d need to find the min twice – and then inserting takes only time one as there is no order. But that is $O(s) + O(s) = O(s)$. In the implementation $s$ runs down from $n$ to 2 so the total time is $O(n + (n - 1) + \ldots + 1) = O(n^2)$. 