Solutions to Homework Assignment 7

---

1. Chapter 6, Exercise 3 (page 314).

Solution Please see the attached sheet.

2. Chapter 6, Exercise 5 (page 316) from the text.

Solution Please see the attached sheet.

3. Chapter 6, Exercise 9 (page 319) from the text.

Solution Please see the attached sheet.

4. Chapter 6, Exercise 12 (page 323) from the text.

Solution Please see the attached sheet.

5. Professor Stewart is consulting for the president of a corporation that is planning a company party. The party has a hierarchical structure; that is, the supervisor relation forms a tree rooted at the president. The personnel office has ranked each employee with a convivality rating, which is a real number. In order to make the party fun for all attendees, the president does not want both an employee and his or her immediate supervisor to attend.

Professor Stewart is given the tree that describes the structure of the corporation. Each node of the tree holds in addition to the pointers, the name of an employee and that employee's convivality ranking.

(a) Describe an algorithm to make up a guest list that maximizes the sum of the convivality ratings of the guests. Analyze the running time of your algorithm.

(b) How can the professor ensure that the president gets invited to his own party?

Solution. Let $T = (V, E)$ be the tree structure representing the supervisor relationship. Let r be the root of the tree. For any vertex v, let $C(v)$ denote the children of v. For each vertex (employee) v, let $f(v)$ be its convivality rating. We consider the following subproblem. For each tree rooted at a vertex v, T_v, we find two subsets: (i) a subset containing v that maximizes the total rating, (ii) a subset not containing v that maximizes the total rating. Let $R^+(v)$ ($R^-(v)$) be the maximum rating of any subset of vertices of T_v that contains (does not contain) v. The recursive formulation is as follows:

$$R^+(v) = \begin{cases} f(v) & : \quad \text{v is a leaf} \\ f(v) + \sum_{u \in C(v)} R^-(u) & : \quad \text{otherwise} \end{cases}$$

$$R^-(v) = \begin{cases} 0 & : \quad \text{v is a leaf} \\ \sum_{u\in C(v)} \max\{R^+(u), R^-(u)\} & : \quad \text{otherwise} \end{cases}$$

The dynamic programming algorithm is given below.

```
MaxRating($T, f$)
1   Topological sort $V$ s.t. child $\prec$ parent in the sorted list.
2   Let $v_1, v_2, \ldots, v_n$ be the sorted list.
3   for $i \leftarrow 1$ to $n$ do
4       $R^+(v_i) \leftarrow f(v_i)$
5       $R^-(v_i) \leftarrow 0$
6   for $i \leftarrow 1$ to $n$ do
7       for $u \in C(v_i)$ do
8           $R^+(v_i) \leftarrow R^+(v_i) + R^-(u)$
9           $R^-(v_i) \leftarrow R^-(v_i) + \max\{R^+(u), R^-(u)\}$
10  return $\max\{R^+(r), R^-(r)\}$
```

The running time of the algorithm is $O(\sum_v deg(v)) = O(E) = O(V)$. The subset of employees that maximize the rating can be obtained easily using recursion starting at the root of the tree and using the R^+ and R^- vectors as indicators.

(b) To compute the maximum rating of any subset of V that includes the root (the president), the $R^-(r)$ term must be ignored from the last line of the pseudo-code.

6. Suppose we are given a set $S = \{s_1, s_2, \ldots, s_n\}$ of positive integers such that $\sum_{i=1}^{n} s_i = T$. Give an $O(nT)$-time algorithm that determines whether there exists a subset $U \subset S$ such that $\sum_{s_i\in U} s_i = \sum_{s_i\in S\setminus U} s_i$.

Solution. This problem can be reduced to the knapsack problem in which we have n items with the ith item having weight s_i as well as value s_i. The knapsack has capacity $T/2$. If the maximum value of a subset of items that can be packed in the knapsack is $T/2$ then the answer to the above question is Yes, otherwise the answer is No.

1.

(a) The graph on nodes $v_1, \ldots, v_5$ with edges $(v_1, v_2), (v_1, v_3), (v_2, v_5), (v_3, v_4)$ and (v_4, v_5) is such an example. The algorithm will return 2 corresponding to the path of edges (v_1, v_2) and (v_2, v_5), while the optimum is 3 using the path $(v_1, v_3), (v_3, v_4)$ and (v_4, v_5).

(b) The idea is to use dynamic programming. The simplest version to think of uses the subproblems $OPT[i]$ for the length of the longest path from v_1 to v_i. One point to be careful of is that not all nodes v_i necessarily have a path from v_1 to v_i. We will use the value "$-\infty$" for the $OPT[i]$ value in this case. We use $OPT(1) = 0$ as the longest path from v_1 to v_1 has 0 edges.

```
Long-path(n)
  Array $M[1 \ldots n]$
  $M[1] = 0$
  For $i = 2, \ldots, n$
     $M = -\infty$
     For all edges $(j, i)$ then
        if $M[j] \neq -\infty$
             if $M < M[j] + 1$ then
                 $M = M[j] + 1$
             endif
        endif
     endfor
     $M[i] = M$
  endfor
  Return $M[n]$ as the length of the longest path.
```

The running time is $O(n^2)$ if you assume that all edges entering a node i can be listed in $O(n)$ time.

[1]ex961.606.761

2.

The key observation to make in this problem is that if the segmentation $y_1 y_2 \ldots y_n$ is an optimal one for the string y, then the segmentation $y_1 y_2 \ldots y_{n-1}$ would be an optimal segmentation for the prefix of y that excludes y_n (because otherwise we could substitute the optimal solution for the prefix in the original problem and get a better solution).

Given this observation, we design the subproblems as follows. Let $Opt(i)$ be the score of the best segmentation of the prefix consisting of the first i characters of y. We claim that the recurrence

$$Opt(i) = min_{j \leq i}\{Opt(j-1) + Quality(j \ldots n)\}$$

would give us the correct optimal segmentation (where $Quality(\alpha \ldots \beta)$ means the quality of the word that is formed by the characters starting from position α and ending in position β). Notice that the desired solution is $Opt(n)$.

We prove the correctness of the above formula by induction on the index i. The base case is trivial, since there is only one word with one letter.

For the inductive step, assume that we know that the Opt function as written above finds the optimum solution for the indices less than i, and we want to show that the value $Opt(i)$ is the optimum cost of any segmentation for the prefix of y up to the i-th character. We consider the last word in the optimal segmentation of this prefix. Let's assume it starts at index $j \leq i$. Then according to our key observation above, the prefix containing only the first $j-1$ characters must also be optimal. But according to our induction hypothesis, $Opt(j)$ will yield us the value of the aforementioned optimal segmentation. Therefore the optimal cost $Opt(i)$ would be equal to $Opt(j)$ plus the cost of the last word.

But notice that our above recurrence exactly does this calculation for each possibility of the last word. Therefore our recurrence will correctly find the cost of the optimal segmentation.

As for the running time, a simple implementation (direct evaluation of the above formula starting at index 1 until n, where n is the number of characters in the input string) will yield a quadratic algorithm.

[1]ex931.924.160

3.

(a) Supopose $s_1 = 10$ and $s_i = 1$ for all $i > 1$; and $x_i = 11$ for all i. Then the optimal solution should re-boot in every other day, thereby processing 10 terabytes every two days.

(b) This problem has quite a few correct dynamic programming solutions; we describe several of the more natural ones here.

1. Let $\mathsf{Opt}(i, j)$ denote the maximum amount of work that can be done starting from day i through day n, given the last reboot occurred j days prior, i.e., the system was rebooted on day $i - j$.

   On each day, there are two options:

   - *Reboot*: which means you don't process anything on day i and day $i + 1$ is the first day after the reboot. Hence, the optimal solution in this case is
   $$\mathsf{Opt}(i, j) = \mathsf{Opt}(i + 1, 1).$$
   - *Continue Processing*: which means that on day i you process the minimum of x_i and s_j. Hence, the optimal solution in this case is
   $$\mathsf{Opt}(i, j) = min\{x_i, s_j\} + \mathsf{Opt}(i + 1, j + 1).$$

   On the last day, there is no advantage gained in rebooting and hence
   $$\mathsf{Opt}(n, j) = min\{x_n, s_j\}$$

   The Algorithm:

   ```
   Set Opt($n, j$) = $min\{x_n, s_j\}$, for all $j$ from $1$ to $n$
   for $i = n - 1$ downto $1$
   for $j = 1$ to $i$
   Opt($i, j$) = $max\{$ Opt($i + 1, 1$), $min\{x_i, s_j\}$+Opt($i + 1, j + 1$)$\}$
   endfor$_j$
   endfor$_i$
   return Opt(1,1)
   ```

   Running Time: Note that the max is over only 2 values and hence is a constant time operation. Since there are only $O(n^2)$ values being calculated, and each one takes $O(1)$ time to calculate, the algorithm takes $O(n^2)$ time.

2. Let $\mathsf{Opt}(i, j)$ to be the maximum number of terabytes that can be processed from days 1 to i, given that the last reboot occurred j days prior to the current day.

   When $j > 0$ (i.e., the system is not rebooted on day i), $min\{x_i, s_j\}$ terabytes are processed and hence,
   $$\mathsf{Opt}(i, j) = \mathsf{Opt}(i - 1, j - 1) + min\{x_i, s_j\}$$

[1] ex736.816.103

When $j = 0$ (i.e., the system is rebooted on day i), no processing is done on day i. Also, the previous reboot could have happened on any of the days prior to day i. Hence,

$$\mathsf{Opt}(i,0) = max_{k=1}^{i-1}\{\mathsf{Opt}(i-1,k)\}$$

Strictly speaking k should run from 0 to $i-1$, i.e., the last reboot could have happened either on day $i-1$ or on day $i-2$ and so on ... or on day 0 (which means no previous reboot). In our case, however, it is not advantageous to reboot on 2 successive days – you might as well do some computation on the first day and reboot on the second day. Since there is a reboot on day i, we can be sure that there is no reboot on day $i-1$, and hence k starts from 1.

The base case for the recursion is:

$$\mathsf{Opt}(0,j) = 0, \forall j = 0,1,\ldots,n$$

A simple algorithm calculating the $\mathsf{Opt}(i,j)$ values can be designed as before taking care that i runs from 1 to n. The final value to be returned is $max_{j=1}^{n}\{\mathsf{Opt}(n,j)\}$.

Running Time: All $\mathsf{Opt}(i,j)$ values take $O(1)$ time when $j \neq 0$. $\mathsf{Opt}(i,0)$ values take $O(n)$ time. Hence the algorithm runs in $n^2 \times O(1) + n \times O(n) = O(n^2)$ time.

3. Let $\mathsf{Opt}(i)$ denote the maximum number of bytes that can be processed starting from day 1 to day i. Suppose that the system was last rebooted on day $j < i$ ($j = 0$ means there was no reboot). Then since day $j+1$, the total number of bytes processed will be $b_{ji} = \Sigma_{k=1}^{i-j} min\{x_{j+k}, s_k\}$. (*Remember that there is no use rebooting on the last day.*) The total work processed till day i would then be $\mathsf{Opt}(j-1) + b_{ji}$. To get the maximum number of bytes processed, maximize over all values of $j < i$. Hence,

$$\mathsf{Opt}(i) = max_{j=0}^{i-1}\{\mathsf{Opt}(j)\} + b_{ji}$$

The base case:

$$\mathsf{Opt}(0) = 0$$

Compute $\mathsf{Opt}(i)$ values starting from $i = 1$ and return the value of $\mathsf{Opt}(n)$.

Running Time: Each b_{ji} value takes $O(n)$ time to calculate, and since there are $O(n^2)$ such values being calculated, the algorithm takes, $O(n^3)$ time.

4.

Let $OPT(j)$ denote the minimum cost of a solution on servers 1 through j, *given* that we place a copy of the file at server j. We want to search over the possible places to put the highest copy of the file before j; say in the optimal solution this at position i. Then the cost for all servers up to i is $OPT(i)$ (since we behave optimally up to i), and the cost for servers $i+1,\ldots,j$ is the sum of the access costs for $i+1$ through j, which is $0+1+\cdots+(j-i-1)=\binom{j-i}{2}$. We also pay c_j to place the server at j.

In the optimal solution, we should choose the best of these solutions over all i. Thus we have

$$OPT(j) = c_j + \min_{0\leq i<j}(OPT(i) + \binom{j-i}{2}),$$

with the initializations $OPT(0)=0$ and $\binom{1}{2}=0$. The values of OPT can be built up in order of increasing j, in time $O(j)$ for iteration j, leading to a total running time of $O(n^2)$. The value we want is $OPT(n)$, and the configuration can be found by tracing back through the array of OPT values.

[1]ex25.372.49