

# CS303 (Spring 2008)— Solutions to Assignment 9

## Problem 1

- (a) The three heavy jobs in our example have values 3, 3, 100, and the three light jobs are 2, 2, 2. Then, the greedy algorithm decides to do light jobs on days 1 and 2, and by the time day 3 comes along, it's too late. The optimum solution would get value 102 or 103 (depending on whether we allow a heavy job to be selected on day 1), while this solution gets 6.
- (b) The optimum solution for days  $1, \dots, i$  has two options on day  $i$  (well, really three, but we'll see in a moment that we only need to consider two). Either it does a heavy job, and skips day  $i - 1$ , doing the best possible with days  $1, \dots, i - 2$ . Or it does a light job on day  $i$ , and does the best possible for days  $1, \dots, i - 1$ . Of those two options, it does the best possible. On day 1, we allow only a light job (though the problem statement was not specific here, and we could have interpreted it either way).

Thus, we get the following recurrence. In it,  $\text{OPT}(i)$  denotes the maximum total profit that can be obtained from days  $1, \dots, i$ , inclusive.

$$\begin{aligned}\text{OPT}(i) &= \max(\ell_i + \text{OPT}(i - 1), h_i + \text{OPT}(i - 2)) \\ \text{OPT}(1) &= \ell_1 \\ \text{OPT}(0) &= 0.\end{aligned}$$

We just gave the justification above. We need two base cases, since the recurrence could jump straight from 2 to 0 or to 1. A bottom-up implementation would be as follows:

---

**Algorithm 1** Job Selection

---

- 1: Let  $a[0] = 0, a[1] = \ell_1$ .
  - 2: **for**  $i = 2$  to  $n$  **do**
  - 3:   Let  $a[i] = \max(\ell_i + a[i - 1], h_i + a[i - 2])$ .
  - 4: return  $a[n]$ .
- 

The running time is obviously  $\Theta(n)$ , as each of the  $n - 2$  iterations does constant work.

## Problem 15–2

As usual, the solution can be found by asking to find a statement about the optimal solution which

- (a) is always correct
- (b) reduces the size of the remaining problem

The optimum solution somehow divides up words  $1, \dots, n$  among one or more lines. What do we know about it? For some  $k < n$ , words  $k + 1, \dots, n$  are on the last line, and the remaining words  $1, \dots, k$  are on the previous lines. How are they divided on the previous lines? Optimally, since there is no interaction otherwise between the previous lines and the last one. And how does the optimum solution choose  $k$ ? Again, it is the best among all choices that would not overflow the last line.

Thus, we will let  $\text{OPT}(i)$  denote the total cost of typesetting words  $1, \dots, i$  optimally, where we also include the penalty on the last line (we'll see in a moment how to deal with this). For notational convenience, we define  $s(i, j) = j - i + \sum_{k=i}^j l_k$  to be the total space required to print words  $i, \dots, j$ , and  $\phi(i) = \min\{j \geq$

$1 \mid s(j, i) \leq M$  to be the index of the word  $j$  such that no more words could be added to a line containing words  $j, \dots, i$  already. We then obtain the following recurrence relation for the optimum solution:

$$\begin{aligned} \text{OPT}(i) &= \min_{k=\phi(i), \dots, i} (\text{OPT}(k-1) + (M - s(k, i))^3) \\ \text{OPT}(0) &= 0. \end{aligned}$$

As observed above, the optimum can place the line break after any word  $k-1$  (so long as the words  $k, \dots, i$  fit into one line), and subject to that, will arrange words  $1, \dots, k-1$  optimally. Of course, if there are no more words left to arrange, there is no cost.

To deal with the special case of the last line, we simply return in the end  $\min_{k=\phi(n), \dots, n} \text{OPT}(k-1)$ , since the last line will contain words  $k, \dots, n$  for some  $k$ , without incurring any penalty. Again, it will contain the words which make the cost for the remaining lines as cheap as possible.

We can now produce the following bottom-up implementation. It also keeps track of *where* the optimal line-breaks are, in addition to the cost, which we described above.

---

**Algorithm 2** Pretty-printing

---

- 1: Precompute all  $s[i, j]$  and  $\phi[i]$  with two simple loops.
  - 2: Let  $a[0] = 0$ .
  - 3: **for**  $i = 1$  to  $n$  **do**
  - 4:   Let  $k$  minimize  $a[k-1] + (M - s[k, i])^3$  among  $k = \phi[i], \dots, i$ .
  - 5:   Set  $a[i] = a[k-1] + (M - s[k, i])^3$ , and  $b[i] = k$ .
  - 6: Let  $k$  minimize  $a[k-1]$  among  $k = \phi[n], \dots, n$ .
  - 7: **while**  $k > 1$  **do**
  - 8:   Print “Line break after word  $k-1$ ”.
  - 9:   Let  $k = b[k-1]$ .
- 

The memory requirement of this is  $\Theta(n)$  for the two arrays  $a, b$ . The array  $s$ , as implemented, would require  $\Theta(n^2)$  space. However, one could easily get around this by storing only the sums from 1 until  $i$  for each  $i$ , and computing  $s[i, j]$  by subtraction. That would also be faster. The running time is  $O(n^2)$  for the pre-computation as specified, and  $O(n)$  with the smarter version. The loop runs  $O(n)$  times, and to compute the minimum inside takes time  $O(n)$  or  $O(1)$ , depending on whether  $M$  is considered as a constant or a variable. The final loop takes  $O(n)$ , since it has  $O(n)$  iterations with constant work each. The running time is thus  $O(n^2)$ . If  $M$  can be large, then one can also show a lower bound of  $\Omega(n^2)$  for this implementation, giving us  $\Theta(n^2)$ .

## Problem 15–4

Given a tree  $T$  with root  $r$ , what do we know about the optimum? It either includes  $r$ , or it does not. If it does, then it cannot include any of the children. Otherwise, it may or may not include some or all of the children. Subject to these constraints, the optimum solution includes the best possible solution from each of the subtrees. Thus, we want to think in terms of  $\text{OPT}(T_v)$ , where  $T_v$  is the subtree rooted at  $v$ .

We still need to incorporate the restriction about including or not including the root. There are two ways: either we add another parameter to  $\text{OPT}(T_v)$ , namely whether  $v$  may be included. Alternatively, we can directly skip one level further down in the tree. Here, we will use the first solution. So we use  $\text{OPT}(T_v, Y)$  to be the total conviviality of the optimum solution for the tree rooted at  $v$  if  $v$  is allowed to be included, and  $\text{OPT}(T_v, N)$  for the same if  $v$  itself must not be included.

We thus obtain the following recurrence relation. Here, for a node  $v$ , we let  $v_1, \dots, v_{k_v}$  denote its children. If the node is a leaf, then  $k_v = 0$ , of course. We denote the conviviality of node  $v$  by  $c_v$ .

$$\text{OPT}(T_v, Y) = \max(c_v + \sum_{i=1}^{k_v} \text{OPT}(T_{v_i}, N), \sum_{i=1}^{k_v} \text{OPT}(T_{v_i}, Y))$$

$$\text{OPT}(T_v, N) = \sum_{i=1}^{k_v} \text{OPT}(T_{v_i}, Y).$$

The reason is that if node  $v$  cannot be included, then the best total conviviality is simply the maximum that can be obtained from all of the subtrees. For each of those subtrees, the root may be included (since the supervisor is not included). On the other hand, if the node  $v$  may be included, the optimum solution has two choices: either include  $v$ , and obtain the maximum possible from each subtree, but without including any of those roots. Or do not include the root, and obtain the maximum possible from each subtree, with the possibility of including the root. The optimum obviously will choose the better of the two.

In general, dynamic programming can often be used to solve problems on trees efficiently, even if they are hard for general graphs. For instance, this problem is known as the maximum weighted independent set problem, and is hard in general, but can be solved on trees.

The bottom-up implementation of this algorithm would look as follows. It again keeps track of the way the optimal solution is obtained, in addition to just the value.

---

**Algorithm 3** Company Party

---

```

1: Perform DFS to number the nodes in DFS order.
2: for all nodes  $v$  in DFS order do
3:   Let  $v_1, \dots, v_{k_v}$  be the children of  $v$ .
4:   Set  $t = \sum_{i=1}^{k_v} a[v_i, Y]$ .
5:   Set  $t' = c_v + \sum_{i=1}^{k_v} a[v_i, N]$ .
6:   Set  $a[v, N] = t$ .
7:   if  $t' > t$  then
8:     Set  $a[v, Y] = t'$  and  $b[v, Y] = Y$ .
9:   else
10:    Set  $a[v, Y] = t$  and  $b[v, Y] = N$ .

```

---

The reason we sorted nodes in DFS order is to make sure by the time we process a node  $v$ , all its children have already been processed, so we can access their array entries safely. Outputting the solution is then done easiest with the following recursive algorithm:

---

**Algorithm 4** Output ( $v, s$ )

---

```

1: if  $b[v, s] = N$  then
2:   for  $j = 1, \dots, k_v$  do
3:     Output( $v_j, Y$ )
4: else
5:   Print "Include  $v$ ".
6:   for  $j = 1, \dots, k_v$  do
7:     Output( $v_j, N$ )

```

---

In each iteration, the inner loop runs over all children of  $v$ , i.e., takes time  $O(d_v)$ . Thus, the total time is  $O(m) = O(n)$ , because the graph is a tree. Clearly, this is also a lower bound, simply to read the input. The DFS also takes time  $O(n)$ , as does the simple recursive output procedure. Thus, the running time is  $\Theta(n)$ .