Solutions to Practice Problems 7

Solution 1:

(a) Our counterexample involves three bottles. The first is absolutely low-cost, but doesn't hold enough pills. The second has a low per-pill cost, but it has a high absolute cost. The third (Goldilocks) bottle, will be just right.

Let $W = 2$. Our first bottle holds $p_1 = 1$ pill and costs $c_1 = \$2$, for a per-pill cost of \$2. Our second bottle holds $p_2 = 6$ pills and costs $c_2 = \$6$, for a per-pill cost of \$1. Our third bottle holds $p_3 = 2$ pills, and costs $c_3 = \$4$, for a per-pill cost of \$2.

The optimal solution is to place both pills in bottle 3, for a total cost of \$4. The cost-greedy algorithm will select the first and third bottles, for a total cost of $\$2 + \$4 = \$6$. The per-pill greedy algorithm will select only the second bottle, for a cost of \$6.

(b) For $0 \le i \le n$ and $0 \le w \le W$, define $C(i, w)$ to be the minimum cost assuming that we are to place w pills using just the first i bottles (or ∞ if there is no feasible solution).

Let's first derive the basis case. If $w = 0$, we can trivially put the zero pills into any number bottles for a cost of \$0, and thus $C(i, 0) = 0$, for all i. If $i = 0$ and $w > 0$, then there are no bottles left in which to place a positive number of pills, and therefore, $C(0, w) = \infty$.

Otherwise, both i and w are at least 1. Before dealing with the general case, we begin with an important, but subtle, observation. The bottles are indexed in some arbitrary order. Suppose that b_{i_0} is the bottle with the smallest index that appears in the optimal solution. We assert that, among all the bottles in the optimal solution, we may assume that this is the only bottle that is not completely full. To see why, observe that if any of the other bottles used in the optimal were not completely full, we could move pills from b_{i_0} to this bottle. Because we pay full price for each bottle we used this will not increase the cost. If bottle b_{i_0} runs out of pills, we do not need it, which would contradict the hypothesis that this is an optimal solution.

There are two cases to consider. Either we do not use the ith bottle or we do. If not, we put all w pills in the first $i - 1$ bottles, for a cost of $C(i, w) = C(i - 1, w)$. Otherwise, we pay the cost of c_i and (by the above observation) put as many pills as we can in bottle i, that is, $\min(p_i, w)$. We then use the previous $i - 1$ bottles to place the remaining pills. Thus, the total cost is $C(i, w) = c_i + C(i - 1, w - \min(p_i, w))$. Clearly, we should take the smaller of the two options, which leads to the following recursive DP formulation:

$$C(i,w) \;=\; \begin{cases} 0 & \text{if } w = 0, \\ \infty & \text{if } i = 0 \text{ and } w > 0, \\ \min\left(\begin{array}{l} C(i-1,w), \\ c_i + C(i-1, w - \min(p_i, w)) \end{array}\right) & \text{otherwise.} \end{cases}$$

If we implement this, there is a table of size $(n + 1)(W + 1) = O(nW)$ to fill, and it takes constant time to fill each entry, for a total of $O(nW)$ time.

(c) The case where bottles can be used an arbitrary number of times, we need only make a minor adjustment to the above rule. When we place pills into bottle i, the recursion is still allowed to use (other copies of) this bottle. Rather on recursing on $C(i-1, w-\min(p_i, w))$, we instead recurse on $C(i, w-\min(p_i, w))$. Otherwise, the above rule is unchanged. Note that the recursion is still well grounded, because even though we do not decrease the i parameter, we decrease the number of pills, which must eventually go down to zero.

Solution 2: The final M and H matrices are shown in Fig. 1 along with the multiplication order.

- $M[1,3]$: The choices are $M[1,1]+M[2,3]+2\cdot 2\cdot 3 = 42$ or $M[1,2]+M[2,3]+2\cdot 5\cdot 3 = 50$. The first is better, and we set $M[1,3] = 29$ and $H[1,3] = 1$.
- $M[2,4]$: The choices are $M[2,2]+M[3,4]+2\cdot 5\cdot 1 = 25$ or $M[2,3]+M[4,4]+2\cdot 3\cdot 1 = 36$. The first is better, and we set $M[2,4] = 25$ and $H[2,4] = 2$.
- $M[1,4]$: The choices are $M[1,1]+M[2,4]+2\cdot 2\cdot 1 = 29$, $M[1,2]+M[3,4]+2\cdot 5\cdot 1 = 45$, or $M[1,3]+M[4,4]+2\cdot 3\cdot 1 = 48$. The first is the best, and we set $M[1,4] = 29$ and $H[1,4] = 1$.

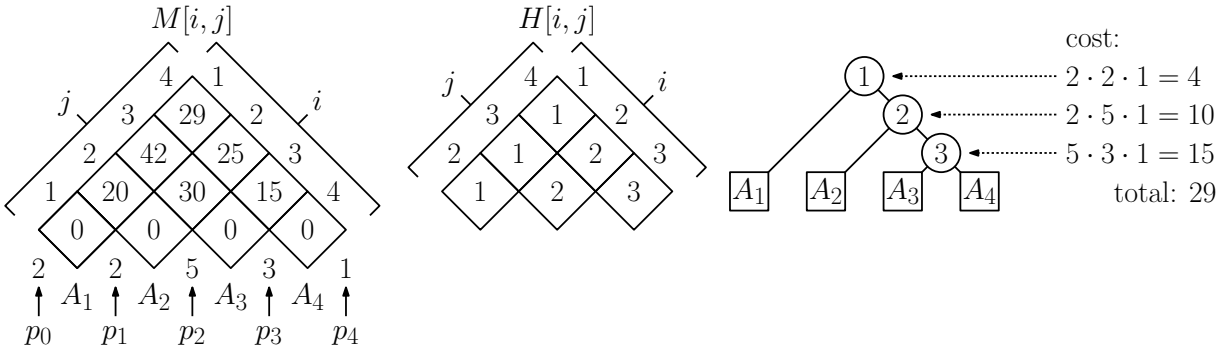

Figure 1: Chain-matrix multiplication.

To get the final multiplication order we see that $H[1,4] = 1$, so we multiply $A_1(A_2 \cdot A_3 \cdot A_4)$. We continue with $H[2,4] = 2$, so this leaves $A_1(A_2(A_3 \cdot A_4))$. The tree is shown in the figure.

Solution 3:

(a) The solution is based on dynamic programming. Let us assume that the points have been sorted by the x-coordinates. For $0 \le i \le n$, let $\text{lis}(i)$ denote the length of the LIS ending at point p_i. For the basis case, $\text{lis}(0) = 0$. For $i > 0$, the LIS that ends with p_i can be expressed as an LIS that ends at some point p_j, where $j < i$ and $y_j < y_i$ and to this we add p_i. Since we do not know what this point is, let us try them all and take the largest. This leads to the following DP formulation

$$\text{lis}(i) \;=\; \begin{cases} 0 & \text{if } i = 0, \\ \max\limits_{\substack{0 \le j < i \\ y_j < y_i}} 1 + \text{lis}(j) & \text{otherwise.} \end{cases}$$

(To avoid subscripting out of bounds, we can define $y_0 = -\infty$.) Since we do not know which point the LIS ends in, the overall answer is the maximum over all the possibilities. Thus, we have

$$\text{lis}(P) \;=\; \max_{1 \le i \le n} \text{lis}(i).$$

(b) The following code block gives an iterative implementation of the above DP formulation.

Longest increasing subsequence

```
LIS(x[1..n], y[1..n])
    // we assume that x is sorted in increasing order
    lis[0] = 0
    for (i = 1 to n)                    // compute lis[i] for each i
        for (j = 0 to i)
            if (y[j] < y[i]) lis[i] = max(lis[i], 1 + lis[j])
    result = 0                          // final result is the max lis[i]
    for (i = 1 to n)
        result = max(result, lis[i])
    return result
```

The running time is clearly $O(n^2)$.

Solution 4: While the problem is stated in terms of constructing two paths, one going out and the other coming back, a straightforward implementation of this idea will not lead to an efficient DP solution. (The issue is that the state information in the formulation would need to record the set of points that we missed on the outbound path, and there are 2^n possible such sets, which is way too many for an efficient solution.)

We will build two paths coming out of the start node until they both converge at p_n. (We use the observation that it doesn't matter which part of the path we call the outward path and which we call the return path, since swapping the two yields the same total distance.) We solve the problem through the application of dynamic programming. There are a number of ways of formulating this as a DP problem. This one is based on a particular way of viewing the process of growing

In order to motivate our approach, let's consider one way of building the two paths simultaneously from left to right. The shorter side (the one ending with the smaller index) selects the next point on the left to jump to. It might jump to the next point in the left-to-right sorted sequence or it might jump ahead many points. If one of the paths jumps way ahead, the shorter path is forced to visit all the subsequent points until it catches up. We don't know which option is best, so our formulation will take both possibilities into account.

In keeping with the recursive nature of DP formulations, we will think of the two paths as being built up using recursion by adding edges to an existing partial solution. For $0 \le i \le j \le n$, we define $C[i, j]$ to be the minimum possible total length of two paths that satisfy the following requirements:

- both paths start at p_0 and move monotonically to the right,
- one path ends at p_i and the other ends at p_j (and if $i = j$ both end at this point),

- every point of $\{p_1, \ldots, p_{i-1}\}$ is hit by exactly one of these two paths, and none of the points $\{p_{i+1}, \ldots, p_{j-1}\}$ is hit by either path.

Observe that the final answer is $C[n, n]$, since (by the above conditions) this is the shortest pair of paths starting at p_0, both end at p_n, and every point from p_1 to p_{n-1} is hit by exactly one of these paths, which is just what we want.

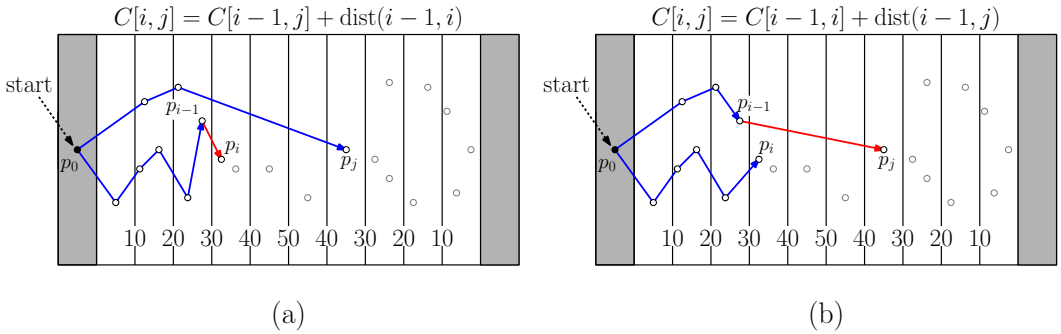

Figure 2: Solution to the double-path problem showing a possible path for $C[i, j]$.

Let's define $\text{dist}(i, j) = \text{dist}(p_i, p_j)$. For the basis case, we set $C[0, j] = \text{dist}(0, j)$, for $0 \le j \le n$, since the only point being hit is p_j. To compute $C[i, j]$ for $i \ge 1$, we observe that one of the two paths must visit the point p_{i-1}. Since we don't know which, we consider each case separately.

- If the path from p_1 to p_i passes through p_{i-1} (see Fig. 2(a)), then the path structure consists of one path from p_1 to p_j, the other path from p_1 to p_{i-1} (which has cost of $C[i-1, j]$) followed by the direct edge (p_{i-1}, p_i) for a total cost of $C[i-1, j] + \text{dist}(i-1, i)$.

- Otherwise, the path from p_1 to p_j passes through p_{i-1} (see Fig. 2(b)), then the path structure consists of one path from p_1 to p_i, the other path from p_1 to p_{i-1} (which has cost of $C[i-1, i]$) followed by the direct edge (p_{i-1}, p_j).

(Note that these two cases degenerate to the same case when $i = j$.) Between these two options, we take whichever yields the lower cost. This leads to the following DP formulation:

$$C[i, j] = \begin{cases} \text{dist}(0, j) & \text{if } i = 0 \\ \min \begin{pmatrix} C[i-1, j] + \text{dist}(i-1, i), \\ C[i-1, i] + \text{dist}(i-1, j) \end{pmatrix} & \text{otherwise.} \end{cases}$$

If implemented, this would involve constructing a matrix of size $O(n^2)$, in which each entry is computed in $O(1)$ time, for a total cost of $O(n^2)$.

We will not discuss recovery of the optimal path, but intuitively the way to do it is to maintain a boolean helper matrix H which records which of the two options we selected between. The paths can be reconstructed in the usual manner by tracing the decisions back from $H[n, n]$, prepending p_{i-1} to one path or the other.

Solution 5: Let $W = \sum_{i=1}^{n} w_i$. Define a boolean array $a[0..n, 0..W]$ to be $a[i, t] = 1$ if there exists a subset of $\{w_1, \ldots, w_i\}$ that sums to t and $a[i, t] = 0$ otherwise. We'll see late why this is useful.

As initialization, observe that $a[i,0] = 1$, since we can always achieve a weight of 0 by taking no objects. Also, for $t \geq 1$, $a[0,t] = 0$, since we cannot achieve any positive weight using no objects. For the induction, consider the computation of $a[i,t]$. If we don't take object i, then we can try to achieve weight t by taking any of the previous $i-1$ objects, and so $a[i,t] = a[i-1,t]$. If we do take object i, then we can achieve a total weight of t if and only if we can achieve a weight of $t - w_i$ using the previous $i-1$ objects. Thus, if $t \geq w_i$, then $a[i,t] = a[i-1,t-w_i]$. Since both of these are possible, we have

$$a[i,t] \;=\; a[i-1,t] \vee ((t \geq w_i) \wedge a[i-1,t-w_i]).$$

Since each entry $a[i,t]$ can be computed in $O(1)$ time, this can be done in $O(n \cdot W)$ time.

Once we have computed $a[n,t]$ for $0 \leq t \leq W$, we solve the partition problem as follows. Let $W' = \lfloor W/2 \rfloor$ be half the total weight. We want the smaller ship to get as close to W' as possible, but without going over. Therefore, we access $a[n,t]$ for $t = W', W'-1, W'-2, \ldots$ until finding the first entry that is equal to 1.

We place a subset of total weight t on Ship 1 and the remainder of $W - t$ goes on Ship 2. Clearly, this is optimal. We return the final weight ratio of $(W-t)/t$. The overall running time is $O(n \cdot W)$.