COMP 271 Design and Analysis of Algorithms
2003 Spring Semester
Solutions to Question Bank Number 3 (Selected Problems)
Revised April 15, 2003

2. The solution doesn't work. Here is a counterexample. Suppose n=3 and $p_0 = 1$, $p_1 = 2$, $p_2 = 32$, and $p_3 = 12$. The suggested algorithm parenthesizes the product as $M_1 \cdot (M_2 \cdot M_3)$, at a cost of $2 \cdot 23 \cdot 12 + 1 \cdot 2 \cdot 12 = 792$ multiplications. The optimal way is $(M_1 \cdot M_2) \cdot M_3$, using $1 \cdot 2 \cdot 32 + 1 \cdot 32 \cdot 12 = 448$. *This solution from "Problems on Algorithms" by Ian Parberry.*

3. (a) Consider the case where $wt[i] = 1$ for all i (the worst case must be at least as bad as this special case). The proof boils down to observing that the recursion tree is a complete binary tree whose height is essentially $h = \min(n, W)$. The number nodes of will be 2^h.

   More formally, let $T(i, W)$ denote the running time of the algorithm for a given pair i and W. We can see that we have the following recurrence (up to constant factors):

$$T(i, W) = \begin{cases} 1 & \text{if } i = 0 \text{ or } w < 0 \\ T(i-1, W) + T(i-1, W-1) & \text{otherwise.} \end{cases}$$

   It is an easy induction proof that $T(i, W) \geq 2^{\min(i,W)}$. The basis case $i = 0$ or $W = 0$ is trivial. For the induction step we have

$$\begin{aligned} T(i, W) &\geq T(i-1, W) + T(i-1, W-1) \\ &\geq 2^{\min(i-1,W)} + 2^{\min(i-1,W-1)} \geq 2 \cdot 2^{\min(i-1,W-1)} = 2 \cdot 2^{\min(i,W)-1} \\ &= 2 \cdot 2^{\min(i,W)}/2 = 2^{\min(i,W)}. \end{aligned}$$

   (b) The problem with the recursive version is that it recomputes many of the same function values over and over again. Again assume that $wt[i] = 1$ for all i. Let $R(i, W)$ be a shorthand for the call with parameters i and W. $R(i, W)$ calls $R(i-1, W)$ and $R(i-1, W-1)$. Both of these call $R(i-2, W-1)$. As you trace the algorithm deeper, you will see that the same procedure is invoked over and over again. The dynamic programming version avoids this duplication, since once a value has been computed for a given i and W, this effort is never repeated.

4. (sketch of solution)

   The algorithm is based on defining a table

$$V(i, C_1, C_2), \quad 0 \leq i \leq n,\ 0 \leq C_1 \leq C,\ 0 \leq C_2 \leq C$$

   in which $V(i, C_1, C_2)$ is the maximum value of objects *from the set of the first i objects* that can be placed in two knapsacks, the first one having weight capacity

C_1, and the second having weight capacity C_2. The optimal solution to the problem is $V(n, C, C)$.

The algorithm is based on the following recurrence relation:

$$V(i, C_1, C_2) = \max\left(V(i-1, C_1, C_2),\ V(i-1, C_1 - w_i, C_2) + v_i,\ V(i-1, C_1, C_2 - w_i) + v_i,\right)$$

(whose formal proof will be omitted here). The initial conditions are $\forall i, V(i, C_1, C_2) = -\infty$ if $C_1 < 0$ or $C_2 < 0$ and $\forall C_1, C_2 \geq 0$, $V(0, C_1, C_2) = 0$. The basic idea behind the equation is is that the three terms on the right hand side correspond to the three cases in which the optimal solution for $V(i, C_1, C_2)$ (i) does not use item i at all, (ii) puts item i in the first knapsack and (iii) puts item i in the second knapsack.

Notice that, if all of the items on the right hand side were already known, then the left hand side could be calculated in $O(1)$ time. The following algorithm therefore fills in the table in $O(nC^2)$ time:

```
KnapSack(v, n, W_1, W_2)
{
  for (w_1 = 0 to W_1)
    for (w_2 = 0 to W_2)
      V[0, w_1, w_2] = 0;
  for (i = 1 to n)
    for (w_1 = 0 to W_1)
      for (w_2 = 0 to W_2)
        V(i, C_1, C_2) = max (V(i − 1, C_1, C_2), V(i − 1, C_1 − w_i, C_2) + v_i, V(i − 1, C_1, C_2 − w_i) + v_i, )
  return V[n, W_1, W_2];
}
```

Calling the procedure with KnapSack(v, n, C, C) solves the problem (we omit the standard technique for figuring out the actual contents of the knapsack from the table).

5. Let $X =< x_1, \ldots, x_n >$ be the given sequence of n numbers. We need to find the longest increasing subsequence in X.

   Algorithm: We first give an algorithm which finds the length of the longest increasing subsequence; later, we will modify it to report a subsequence with this length.

   Let $X_i =< x_1, \ldots, x_i >$ denote the prefix of X consisting of the first i items. Define $c[i]$ to be the length of the longest increasing subsequence that ends with x_i. It is clear that the length of the longest increasing subsequence in X is given by $\max_{1 \leq i \leq n} c[i]$.

   The longest increasing subsequence that ends with x_i has the form $< Z, x_i >$ where Z is the longest increasing subsequence that ends with x_r for some $r < i$ and $x_r \leq x_i$. Thus, we have the following recurrence relation:

$$c[i] = \begin{cases} 1 & \text{if } i = 1 \\ 1 & \text{if } x_r > x_i \text{ for } 1 \leq r < i \\ \max_{\substack{1 \leq r < i \\ x_r \leq x_i}} c[r] + 1 & \text{if } i > 1 \end{cases}$$

The basis follows from the fact the longest increasing subsequence in a sequence consisting of one number is the number itself. The recurrence relation says that if all the numbers to the left of i are greater than x_i then the length of the longest increasing subsequence ending in x_i is 1. Otherwise, the length of the longest increasing subsequence ending in x_i is 1 more than the length of the longest increasing subsequence ending at a number x_r to the left of x_i such that x_r is no greater than the x_i.

We store the $c[i]$'s in an array whose entries are computed in order of increasing i. After computing the c array we run through all the entries to find the maximum value. This is the length of the longest increasing subsequence in X.

In order to report the optimal subsequence we need to store for each i, not only $c[i]$ but also the value of r which achieves the maximum in the recurrence relation. Denote this by $r[i]$. Then we can trace the solution as follows. Let $c[k] = \max_{1<i<n} c[i]$. Then x_k is the last number in the optimal subsequence. The second to last number is $x_{r[k]}$, the third to last number is $x_{r[r[k]]}$ and so on until we have found the all the numbers of the optimal subsequence.

Running Time: Since it takes $O(i)$ time to compute the i-th entry of the c array, the total time to compute the c array is $O(\sum i) = O(n^2)$. It takes $O(n)$ time to find the maximum in the c array. Finally, the time to trace the solution is $O(n)$. Thus, the running time is dominated by the time it takes to compute the c array, which is $O(n^2)$.

7. The solution is to construct a boolean array $A[i, j]$, $0 \leq i \leq n$ and $0 \leq j \leq W$, defined as follows: $A[i, j] = \mathit{true}$ if there is a subset of $\{x_1, x_2, \ldots, x_i\}$ that sums to j, else $A[i, j] = false$. We start with some observations.

   Basis: $A[i, 0] = \mathit{true}$, $0 \leq i \leq n$, because given 0 or more items, you can always form the sum 0 by picking no item. Also, $A[0, j] = false$, $1 \leq j \leq W$, because if there are no items to pick from, then we cannot form any sum > 0.

   Last weight too large: $A[i, j] = A[i-1, j]$ if $i > 0$ and $x_i > j$. The solution cannot contain x_i if x_i exceeds j, the sum to be formed. Therefore the sum j can be formed using a subset of $\{x_1, x_2, \ldots, x_i\}$ if and only if it can be formed using a subset of $\{x_1, x_2, \ldots, x_{i-1}\}$.

   Last weight not too large: $A[i, j] = (A[i-1, j-x_i] \text{ OR } A[i-1, j])$, if $i > 0$ and $j \geq x_i$. This follows from the following observations. If sum j can be formed using a subset of $\{x_1, x_2, \ldots, x_{i-1}\}$, then either this subset includes item x_i or it does not. If it includes item x_i then it should be possible to form the sum $j - x_i$ using a subset of $\{x_1, x_2, \ldots, x_{i-1}\}$; otherwise if it does not include item x_i then it should be possible to form the sum j using a subset of $\{x_1, x_2, \ldots, x_{i-1}\}$.

Combining these observations we have the following recurrence relation:

$$A[i,j] = \begin{cases} true & \text{if } 0 \le i \le n \text{ and } j = 0 \\ false & \text{if } i = 0 \text{ and } 1 \le j \le W \\ A[i-1,j] & \text{if } i > 0 \text{ and } x_i > j \\ A[i-1,j-x_i] \text{ OR } A[i-1,j]) & \text{if } i > 0 \text{ and } j \ge x_i \end{cases}$$

The algorithm takes as inputs the sum to be formed W, the number of items n, and the sequence $x = x_1, x_2, \ldots, x_n$. It stores the $A[i,j]$ values in a table $A[0 \ldots n, 0 \ldots W]$ whose values are computed in order of increasing i (note that for any given i it does not matter in which order we compute the $A[i,j]$'s). Following this order ensures that the table entries used to compute $A[i,j]$ have all been computed *before* the algorithm evaluates $A[i,j]$. At the end of the computation, $A[n,W]$ is true, if there is a subset that sums to W, otherwise it is false.

```
Dynamic-SubsetSum(x, n, W)
    A[0,0] = true
    for j = 1 to W do
        A[0,j] = false
    for i = 1 to n do
        A[i,0] = true
        for j = 1 to W do
            if x_i > j then
                A[i,j] = A[i-1,j]
            else A[i,j] = A[i-1,j-x_i] OR A[i-1,j]
```

Running Time: Since the table has $O(nW)$ entries and it takes constant time to compute any one entry, the total time to build the table is $O(nW)$. The total running time is $O(nW)$.

8.

$$D^{(0)} = \begin{bmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & \infty & \infty \\ \infty & 2 & 0 & \infty & \infty & -8 \\ -4 & \infty & \infty & 0 & 3 & \infty \\ \infty & 7 & \infty & \infty & 0 & \infty \\ \infty & 5 & 10 & \infty & \infty & 0 \end{bmatrix}$$

$$D^{(1)} = \begin{bmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & 0 & \infty \\ \infty & 2 & 0 & \infty & \infty & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ \infty & 7 & \infty & \infty & 0 & \infty \\ \infty & 5 & 10 & \infty & \infty & 0 \end{bmatrix}$$

$$D^{(3)} = D^{(2)} = \begin{bmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & 0 & \infty \\ 3 & 2 & 0 & 4 & 2 & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ 8 & 7 & \infty & 9 & 0 & \infty \\ 6 & 5 & 10 & 7 & 5 & 0 \end{bmatrix}$$

$$D^{(4)} = \begin{bmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ 0 & 2 & 0 & 4 & -1 & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{bmatrix}$$

$$D^{(5)} = \begin{bmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ 0 & 2 & 0 & 4 & -1 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{bmatrix}$$

$$D^{(6)} = \begin{bmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ -5 & -3 & 0 & -1 & -6 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{bmatrix}$$