EXERCISES

1. Let $X = a, a, b, a, a, b, a, b, a, a$ and $Y = b, a, b, a, a, b, a, b$. Find a minimum-cost edit sequence that transforms X into Y.

2. Present a pseudocode algorithm that implements the string editing algorithm discussed in this section. Program it and test its correctness using suitable data.

3. Modify the above program not only to compute $cost(n, m)$ but also to output a minimum-cost edit sequence. What is the time complexity of your program?

4. Given a sequence X of symbols, a subsequence of X is defined to be any contiguous portion of X. For example, if $X = x_1, x_2, x_3, x_4, x_5$, x_2, x_3 and x_1, x_2, x_3 are subsequences of X. Given two sequences X and Y, present an algorithm that will identify the longest subsequence that is common to both X and Y. This problem is known as *the longest common subsequence problem.* What is the time complexity of your algorithm?

5.7 0/1 KNAPSACK

The terminology and notation used in this section is the same as that in Section 5.1. A solution to the knapsack problem can be obtained by making a sequence of decisions on the variables $x_1, x_2, \ldots, x_n$. A decision on variable x_i involves determining which of the values 0 or 1 is to be assigned to it. Let us assume that decisions on the x_i are made in the order $x_n, x_{n-1}, \ldots, x_1$. Following a decision on x_n, we may be in one of two possible states: the capacity remaining in the knapsack is m and no profit has accrued or the capacity remaining is $m - w_n$ and a profit of p_n has accrued. It is clear that the remaining decisions $x_{n-1}, \ldots, x_1$ must be optimal with respect to the problem state resulting from the decision on x_n. Otherwise, $x_n, \ldots, x_1$ will not be optimal. Hence, the principle of optimality holds.

Let $f_j(y)$ be the value of an optimal solution to KNAP$(1, j, y)$. Since the principle of optimality holds, we obtain

$$f_n(m) = \max \ \{f_{n-1}(m), f_{n-1}(m - w_n) + p_n\} \tag{5.14}$$

For arbitrary $f_i(y)$, $i > 0$, Equation 5.14 generalizes to

$$f_i(y) = \max \ \{f_{i-1}(y), f_{i-1}(y - w_i) + p_i\} \tag{5.15}$$

Equation 5.15 can be solved for $f_n(m)$ by beginning with the knowledge $f_0(y) = 0$ for all y and $f_i(y) = -\infty, y < 0$. Then $f_1, f_2, \ldots, f_n$ can be successively computed using (5.15).

When the w_i's are integer, we need to compute $f_i(y)$ for integer y, $0 \leq y \leq m$. Since $f_i(y) = -\infty$ for $y < 0$, these function values need not be computed explicitly. Since each f_i can be computed from f_{i-1} in $\Theta(m)$ time, it takes $\Theta(mn)$ time to compute f_n. When the w_i's are real numbers, $f_i(y)$ is needed for real numbers y such that $0 \leq y \leq m$. So, f_i cannot be explicitly computed for all y in this range. Even when the w_i's are integer, the explicit $\Theta(mn)$ computation of f_n may not be the most efficient computation. So, we explore an alternative method for both cases.

Notice that $f_i(y)$ is an ascending step function; i.e., there are a finite number of y's, $0 = y_1 < y_2 < \cdots < y_k$, such that $f_i(y_1) < f_i(y_2) < \cdots < f_i(y_k)$; $f_i(y) = -\infty$, $y < y_1$; $f_i(y) = f(y_k)$, $y \geq y_k$; and $f_i(y) = f_i(y_j)$, $y_j \leq y < y_{j+1}$. So, we need to compute only $f_i(y_j)$, $1 \leq j \leq k$. We use the ordered set $S^i = \{(f(y_j), y_j) | 1 \leq j \leq k\}$ to represent $f_i(y)$. Each member of S^i is a pair (P, W), where $P = f_i(y_j)$ and $W = y_j$. Notice that $S^0 = \{(0, 0)\}$. We can compute S^{i+1} from S^i by first computing

$$S_1^i = \{(P, W) | (P - p_i, W - w_i) \in S^i\} \tag{5.16}$$

Now, S^{i+1} can be computed by merging the pairs in S^i and S_1^i together. Note that if S^{i+1} contains two pairs (P_j, W_j) and (P_k, W_k) with the property that $P_j \leq P_k$ and $W_j \geq W_k$, then the pair (P_j, W_j) can be discarded because of (5.15). Discarding or purging rules such as this one are also known as *dominance rules*. Dominated tuples get purged. In the above, (P_k, W_k) dominates (P_j, W_j).

Interestingly, the strategy we have come up with can also be derived by attempting to solve the knapsack problem via a systematic examination of the up to 2^n possibilities for $x_1, x_2, \ldots, x_n$. Let S^i represent the possible states resulting from the 2^i decision sequences for $x_1, \ldots, x_i$. A state refers to a pair (P_j, W_j), W_j being the total weight of objects included in the knapsack and P_j being the corresponding profit. To obtain S^{i+1}, we note that the possibilities for x_{i+1} are $x_{i+1} = 0$ or $x_{i+1} = 1$. When $x_{i+1} = 0$, the resulting states are the same as for S^i. When $x_{i+1} = 1$, the resulting states are obtained by adding (p_{i+1}, w_{i+1}) to each state in S^i. Call the set of these additional states S_1^i. The S_1^i is the same as in Equation 5.16. Now, S^{i+1} can be computed by merging the states in S^i and S_1^i together.

Example 5.21 Consider the knapsack instance $n = 3, (w_1, w_2, w_3) = (2, 3, 4)$, $(p_1, p_2, p_3) = (1, 2, 5)$, and $m = 6$. For these data we have

$$\begin{aligned}
S^0 &= \{(0,0)\}; S_1^0 = \{(1,2)\} \\
S^1 &= \{(0,0),(1,2)\}; S_1^1 = \{(2,3),(3,5)\} \\
S^2 &= \{(0,0),(1,2),(2,3),(3,5)\}; S_1^2 = \{(5,4),(6,6),(7,7),(8,9)\} \\
S^3 &= \{(0,0),(1,2),(2,3),(5,4),(6,6),(7,7),(8,9)\}
\end{aligned}$$

Note that the pair (3, 5) has been eliminated from S^3 as a result of the purging rule stated above. □

When generating the S^i's, we can also purge all pairs (P, W) with $W > m$ as these pairs determine the value of $f_n(x)$ only for $x > m$. Since the knapsack capacity is m, we are not interested in the behavior of f_n for $x > m$. When all pairs (P_j, W_j) with $W_j > m$ are purged from the S^i's, $f_n(m)$ is given by the P value of the last pair in S^n (note that the S^i's are ordered sets). Note also that by computing S^n, we can find the solutions to all the knapsack problems KNAP$(1, n, x)$, $0 \leq x \leq m$, and not just KNAP$(1, n, m)$. Since, we want only a solution to KNAP$(1, n, m)$, we can dispense with the computation of S^n. The last pair in S^n is either the last one in S^{n-1} or it is $(P_j + p_n, W_j + w_n)$, where $(P_j, W_j) \in S^{n-1}$ such that $W_j + w_n \leq m$ and W_j is maximum.

If $(P1, W1)$ is the last tuple in S^n, a set of 0/1 values for the x_i's such that $\sum p_i x_i = P1$ and $\sum w_i x_i = W1$ can be determined by carrying out a search through the S^is. We can set $x_n = 0$ if $(P1, W1) \in S^{n-1}$. If $(P1, W1) \notin S^{n-1}$, then $(P1 - p_n, W1 - w_n) \in S^{n-1}$ and we can set $x_n = 1$. This leaves us to determine how either $(P1, W1)$ or $(P1 - p_n, W1 - w_n)$ was obtained in S^{n-1}. This can be done recursively.

Example 5.22 With $m = 6$, the value of $f_3(6)$ is given by the tuple (6, 6) in S^3 (Example 5.21). The tuple (6, 6) $\notin S^2$, and so we must set $x_3 = 1$. The pair (6, 6) came from the pair $(6 - p_3, 6 - w_3) = (1, 2)$. Hence (1, 2) $\in S^2$. Since $(1, 2) \in S^1$, we can set $x_2 = 0$. Since (1, 2) $\notin S^0$, we obtain $x_1 = 1$. Hence an optimal solution is $(x_1, x_2, x_3) = (1, 0, 1)$. □

We can sum up all we have said so far in the form of an informal algorithm DKP (Algorithm 5.6). To evaluate the complexity of the algorithm, we need to specify how the sets S^i and S^i_1 are to be represented; provide an algorithm to merge S^i and S^i_1; and specify an algorithm that will trace through $S^{n-1}, \ldots, S^1$ and determine a set of 0/1 values for $x_n, \ldots, x_1$.

We can use an array *pair*[ ] to represent all the pairs (P, W). The P values are stored in *pair*[ ].*p* and the W values in *pair*[ ].*w*. Sets $S^0, S^1, \ldots, S^{n-1}$ can be stored adjacent to each other. This requires the use of pointers $b[i]$, $0 \leq i \leq n$, where $b[i]$ is the location of the first element in S^i, $0 \leq i < n$, and $b[n]$ is one more than the location of the last element in S^{n-1}.

Example 5.23 Using the representation above, the sets S^0, S^1, and S^2 of Example 5.21 appear as

```
Algorithm DKP(p, w, n, m)
{
    S^0 := {(0,0)};
    for i := 1 to n − 1 do
    {
        S_1^{i−1} := {(P, W)|(P − p_i, W − w_i) ∈ S^{i−1} and W ≤ m};
        S^i := MergePurge(S^{i−1}, S_1^{i−1});
    }
    (PX, WX) :=last pair in S^{n−1};
    (PY, WY) := (P′ + p_n, W′ + w_n) where W′ is the largest W in
        any pair in S^{n−1} such that W + w_n ≤ m;
    // Trace back for x_n, x_{n−1}, ..., x_1.
    if (PX > PY) then x_n := 0;
    else x_n := 1;
    TraceBackFor(x_{n−1}, ..., x_1);
}
```

Algorithm 5.6 Informal knapsack algorithm

	1	2	3	4	5	6	7	
pair[ ].*p*	0	0	1	0	1	2	3	
pair[ ].*w*	0	0	2	0	2	3	5	
	↑	↑		↑				↑
	$b[0]$	$b[1]$		$b[2]$				$b[3]$

□

The merging and purging of S^{i-1} and S_1^{i-1} can be carried out at the same time that S_1^{i-1} is generated. Since the pairs in S^{i-1} are in increasing order of P and W, the pairs for S^i are generated in this order. If the next pair generated for S_1^{i-1} is (PQ, WQ), then we can merge into S^i all pairs from S^{i-1} with W value $\leq WQ$. The purging rule can be used to decide whether any pairs get purged. Hence, no additional space is needed in which to store S_1^{i-1}.

DKnap (Algorithm 5.7) generates S^i from S^{i-1} in this way. The S^i's are generated in the **for** loop of lines 7 to 42 of Algorithm 5.7. At the start of each iteration $t = b[i-1]$ and h is the index of the last pair in S^{i-1}. The variable k points to the next tuple in S^{i-1} that has to be merged into S^i. In line 10, the function Largest determines the largest q, $t \leq q \leq h$, for which $pair[q].w + w[i] \leq m$. This can be done by performing a binary search. The code for this function is left as an exercise. Since u is set such that for all $W_j, h \geq j > u$, $W_j + w_i > m$, the pairs for S_1^{i-1} are $(P(j) + p_i, W(j) + w_i)$, $1 \leq j \leq u$. The **for** loop of lines 11 to 33 generates these pairs. Each time a pair (pp, ww) is generated, all pairs (P, W) in S^{i-1} with $W < ww$ not yet purged or merged into S^i are merged into S^i. Note that none of these may be purged. Lines 21 to 25 handle the case when the next pair in S^{i-1} has a W value equal to ww. In this case the pair with lesser P value gets purged. In case $pp > P(next-1)$, then the pair (pp, ww) gets purged. Otherwise, (pp, ww) is added to S^i. The **while** loop of lines 31 and 32 purges all unmerged pairs in S^{i-1} that can be purged at this time. Finally, following the merging of S_1^{i-1} into S^i, there may be pairs remaining in S^{i-1} to be merged into S^i. This is taken care of in the **while** loop of lines 35 to 39. Note that because of lines 31 and 32, none of these pairs can be purged. Function TraceBack (line 43) implements the **if** statement and trace-back step of the function DKP (Algorithm 5.6). This is left as an exercise.

If $|S^i|$ is the number of pairs in S^i, then the array *pair* should have a minimum dimension of $d = \sum_{0 \leq i \leq n-1} |S^i|$. Since it is not possible to predict the exact space needed, it is necessary to test for $next > d$ each time $next$ is incremented. Since each S^i, $i > 0$, is obtained by merging S^{i-1} and S_1^{i-1} and $|S_1^{i-1}| \leq |S^{i-1}|$, it follows that $|S^i| \leq 2|S^{i-1}|$. In the worst case no pairs will get purged and

$$\sum_{0 \leq i \leq n-1} |S^i| = \sum_{0 \leq i \leq n-1} 2^i = 2^n - 1$$

The time needed to generate S^i from S^{i-1} is $\Theta(|S^{i-1}|)$. Hence, the time needed to compute all the S^i's, $0 \leq i < n$, is $\Theta(\sum |S^{i-1}|)$. Since $|S^i| \leq 2^i$, the time needed to compute all the S^i's is $O(2^n)$. If the p_j's are integers, then each pair (P, W) in S^i has an integer P and $P \leq \sum_{1 \leq j \leq i} p_j$. Similarly, if the w_j's are integers, each W is an integer and $W \leq m$. In any S^i the pairs have distinct W values and also distinct P values. Hence,

$$|S^i| \leq 1 + \sum_{1 \leq j \leq i} p_j$$

when the p_j's are integers and

$$|S^i| \leq 1 + \min \left\{ \sum_{1 \leq j \leq i} w_j, m \right\}$$

```
PW = record {float p; float w; }

Algorithm DKnap(p, w, x, n, m)
{
    // pair[ ] is an array of PW's.
    b[0] := 1; pair[1].p := pair[1].w := 0.0; // S^0
    t := 1; h := 1; // Start and end of S^0
    b[1] := next := 2; // Next free spot in pair[ ]
    for i := 1 to n - 1 do
    { // Generate S^i.
        k := t;
        u := Largest(pair, w, t, h, i, m);
        for j := t to u do
        { // Generate S_1^{i-1} and merge.
            pp := pair[j].p + p[i]; ww := pair[j].w + w[i];
                // (pp, ww) is the next element in S_1^{i-1}.
            while ((k ≤ h) and (pair[k].w ≤ ww)) do
            {
                pair[next].p := pair[k].p;
                pair[next].w := pair[k].w;
                next := next + 1; k := k + 1;
            }
            if ((k ≤ h) and (pair[k].w = ww)) then
            {
                if pp < pair[k].p then pp := pair[k].p;
                k := k + 1;
            }
            if pp > pair[next - 1].p then
            {
                pair[next].p := pp; pair[next].w := ww;
                next := next + 1 ;
            }
            while ((k ≤ h) and (pair[k].p ≤ pair[next - 1].p))
                do k := k + 1;
        }
        // Merge in remaining terms from S^{i-1}.
        while (k ≤ h) do
        {
            pair[next].p := pair[k].p; pair[next].w := pair[k].w;
            next := next + 1; k := k + 1;
        }
        // Initialize for S^{i+1}.
        t := h + 1; h := next - 1; b[i + 1] := next;
    }
    TraceBack(p, w, pair, x, m, n);
}
```

Algorithm 5.7 Algorithm for 0/1 knapsack problem when the w_j's are integers. When both the p_j's and w_j's are integers, the time and space complexity of DKnap (excluding the time for TraceBack) is $O(\min\{2^n, n\sum_{1\le i\le n} p_i, nm\})$. In this bound $\sum_{1\le i\le n} p_i$ can be replaced by $\sum_{1\le i\le n} p_i/\gcd(p_1,\ldots,p_n)$ and m by $\gcd(w_1,w_2,\ldots,w_n,m)$ (see the exercises). The exercises indicate how TraceBack may be implemented so as to have a space complexity $O(1)$ and a time complexity $O(n^2)$.

Although the above analysis may seem to indicate that DKnap requires too much computational resource to be practical for large n, in practice many instances of this problem can be solved in a reasonable amount of time. This happens because usually, all the p's and w's are integers and m is much smaller than 2^n. The purging rule is effective in purging most of the pairs that would otherwise remain in the S^i's.

Algorithm DKnap can be speeded up by the use of heuristics. Let L be an estimate on the value of an optimal solution such that $f_n(m) \ge L$. Let PLEFT$(i) = \sum_{i<j\le n} p_j$. If S^i contains a tuple (P, W) such that $P +$ PLEFT$(i) < L$, then (P, W) can be purged from S^i. To see this, observe that (P, W) can contribute at best the pair $(P + \sum_{i<j\le n} p_j, W + \sum_{i<j\le n} w)$ to S_1^{n-1}. Since $P + \sum_{i<j\le n} p_j = P +$ PLEFT$(i) < L$, it follows that this pair cannot lead to a pair with value at least L and so cannot determine an optimal solution. A simple way to estimate L such that $L \le f_n(m)$ is to consider the last pair (P, W) in S^i. Then, $P \le f_n(m)$. A better estimate is obtained by adding some of the remaining objects to (P, W). Example 5.24 illustrates this. Heuristics for the knapsack problem are discussed in greater detail in the chapter on branch-and-bound. The exercises explore a divide-and-conquer approach to speed up DKnap so that the worst case time is $O(2^{n/2})$.

Example 5.24 Consider the following instance of the knapsack problem: $n = 6, (p_1, p_2, p_3, p_4, p_5, p_6) = (w_1, w_2, w_3, w_4, w_5, w_6) = (100, 50, 20, 10, 7, 3)$, and $m = 165$. Attempting to fill the knapsack using objects in the order 1, 2, 3, 4, 5, and 6, we see that objects 1, 2, 4, and 6 fit in and yield a profit of 163 and a capacity utilization of 163. We can thus begin with $L = 163$ as a value with the property $L \le f_n(m)$. Since $p_i = w_i$, every pair $(P, W) \in S^i$, $0 \le i \le 6$ has $P = W$. Hence, each pair can be replaced by the singleton P or W. PLEFT(0) = 190, PLEFT(1) = 90, PLEFT(2) = 40, PLEFT(3) = 20, PLEFT(4) = 10, PLEFT(5) = 3, and PLEFT(6) = 0. Eliminating from each S^i any singleton P such that $P+$ PLEFT$(i) < L$, we obtain

$$S^0 = \{0\}; \quad S_1^0 = \{100\}$$
$$S^1 = \{100\}; \quad S_1^1 = \{150\}$$
$$S^2 = \{150\}; \quad S_1^2 = \phi$$

$$
\begin{aligned}
S^3 = \{150\};\quad S_1^3 = \{160\} \\
S^4 = \{160\};\quad S_1^4 = \phi \\
S^5 = \{160\}
\end{aligned}
$$

The singleton 0 is deleted from S^1 as $0 + \text{PLEFT}(1) < 163$. The set S_1^2 does not contain the singleton $150 + 20 = 170$ as $m < 170$. S^3 does not contain the 100 or the 120 as each is less than $L - \text{PLEFT}(3)$. And so on. The value $f_6(165)$ can be determined from S^5. In this example, the value of L did not change. In general, L will change if a better estimate is obtained as a result of the computation of some S^i. If the heuristic wasn't used, then the computation would have proceeded as

$$
\begin{aligned}
S^0 &= \{0\} \\
S^1 &= \{0, 100\} \\
S^2 &= \{0, 50, 100, 150\} \\
S^3 &= \{0, 20, 50, 70, 100, 120, 150\} \\
S^4 &= \{0, 10, 20, 30, 50, 60, 70, 80, 100, 110, 120, 130, 150, 160\} \\
S^5 &= \{0, 7, 10, 17, 20, 27, 30, 37, 50, 57, 60, 67, 70, 77, 80, 87, 100, \\
&\quad\ 107, 110, 117, 120, 127, 130, 137, 150, 157, 160\}
\end{aligned}
$$

The value $f_6(165)$ can now be determined from S^5, using the knowledge $(p_6, w_6) = (3, 3)$. □

EXERCISES

1. Generate the sets S^i, $0 \le i \le 4$ (Equation 5.16), when $(w_1, w_2, w_3, w_4) = (10, 15, 6, 9)$ and $(p_1, p_2, p_3, p_4) = (2, 5, 8, 1)$.

2. Write a function Largest($pair, w, t, h, i, m$) that uses binary search to determine the largest q, $t \le q \le h$, such that $pair[q].w + w[i] \le m$.

3. Write a function TraceBack to determine an optimal solution $x_1, x_2, \ldots, x_n$ to the knapsack problem. Assume that $S^i, 0 \le i < n$, have already been computed as in function DKnap. Knowing $b(i)$ and $b(i+1)$, you can use a binary search to determine whether $(P', W') \in S^i$. Hence, the time complexity of your algorithm should be no more than $O(n \max_i\{\log |S^i|\}) = O(n^2)$.

4. Give an example of a set of knapsack instances for which $|S^i| = 2^i$, $0 \le i \le n$. Your set should include one instance for each n.