obtained by making explicit appeal to

Definition 5.1 [Principle of optimality] The principle of optimality states that an optimal sequence of decisions has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision. □

Thus, the essential difference between the greedy method and dynamic programming is that in the greedy method only one decision sequence is ever generated. In dynamic programming, many decision sequences may be generated. However, sequences containing suboptimal subsequences cannot be optimal (if the principle of optimality holds) and so will not (as far as possible) be generated.

Example 5.5 [Shortest path] Consider the shortest-path problem of Example 5.3. Assume that $i, i_1, i_2, \ldots, i_k, j$ is a shortest path from i to j. Starting with the initial vertex i, a decision has been made to go to vertex i_1. Following this decision, the problem state is defined by vertex i_1 and we need to find a path from i_1 to j. It is clear that the sequence $i_1, i_2, \ldots, i_k, j$ must constitute a shortest i_1 to j path. If not, let $i_1, r_1, r_2, \ldots, r_q, j$ be a shortest i_1 to j path. Then $i, i_1, r_1, \cdots, r_q, j$ is an i to j path that is shorter than the path $i, i_1, i_2, \ldots, i_k, j$. Therefore the principle of optimality applies for this problem. □

Example 5.6 [0/1 knapsack] The 0/1 knapsack problem is similar to the knapsack problem of Section 4.2 except that the x_i's are restricted to have a value of either 0 or 1. Using KNAP(l, j, y) to represent the problem

$$\begin{array}{c} \text{maximize} \sum_{l \le i \le j} p_i x_i \\ \text{subject to} \sum_{l \le i \le j} w_i x_i \le y \\ x_i = 0 \text{ or } 1, \ l \le i \le j \end{array} \tag{5.1}$$

the knapsack problem is KNAP$(1, n, m)$. Let $y_1, y_2, \ldots, y_n$ be an optimal sequence of 0/1 values for $x_1, x_2, \ldots, x_n$, respectively. If $y_1 = 0$, then $y_2, y_3, \ldots, y_n$ must constitute an optimal sequence for the problem KNAP$(2, n, m)$. If it does not, then $y_1, y_2, \ldots, y_n$ is not an optimal sequence for KNAP$(1, n, m)$. If $y_1 = 1$, then $y_2, \ldots, y_n$ must be an optimal sequence for the problem KNAP$(2, n, m - w_1)$. If it isn't, then there is another 0/1 sequence $z_2, z_3, \ldots, z_n$ such that $\sum_{2 \le i \le n} w_i z_i \le m - w_1$ and $\sum_{2 \le i \le n} p_i z_i > \sum_{2 \le i \le n} p_i y_i$. Hence, the sequence $y_1, z_2, z_3, \ldots, z_n$ is a sequence for (5.1) with greater value. Again the principle of optimality applies. □

Let S_0 be the initial problem state. Assume that n decisions d_i, $1 \le i \le n$, have to be made. Let $D_1 = \{r_1, r_2, \ldots, r_j\}$ be the set of possible decision values for d_1. Let S_i be the problem state following the choice of decision r_i, $1 \le i \le j$. Let Γ_i be an optimal sequence of decisions with respect to the problem state S_i. Then, when the principle of optimality holds, an optimal sequence of decisions with respect to S_0 is the best of the decision sequences r_i, Γ_i, $1 \le i \le j$.

Example 5.7 [Shortest path] Let A_i be the set of vertices adjacent to vertex i. For each vertex $k \in A_i$, let Γ_k be a shortest path from k to j. Then, a shortest i to j path is the shortest of the paths $\{i, \Gamma_k | k \in A_i\}$. □

Example 5.8 [0/1 knapsack] Let $g_j(y)$ be the value of an optimal solution to KNAP$(j + 1, n, y)$. Clearly, $g_0(m)$ is the value of an optimal solution to KNAP$(1, n, m)$. The possible decisions for x_1 are 0 and 1 ($D_1 = \{0, 1\}$). From the principle of optimality it follows that

$$g_0(m) = \max \ \{g_1(m), \ g_1(m - w_1) + p_1\} \tag{5.2}$$

□

While the principle of optimality has been stated only with respect to the initial state and decision, it can be applied equally well to intermediate states and decisions. The next two examples show how this can be done.

Example 5.9 [Shortest path] Let k be an intermediate vertex on a shortest i to j path $i, i_1, i_2, \ldots, k, p_1, p_2, \ldots, j$. The paths $i, i_1, \ldots, k$ and $k, p_1, \ldots, j$ must, respectively, be shortest i to k and k to j paths. □

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, $0 \le y \le m$. 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 $0 \le y \le m$. 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 $f_i(y_1) < f_i(y_2) < \cdots < f_i(y_k)$; $f_i(y) = -\infty$, $y < y_1$; $f_i(y) = f(y_k)$, $y \ge y_k$; and $f_i(y) = f_i(y_j)$, $y_j \le y < y_{j+1}$. So, we need to compute only $f_i(y_j)$, $1 \le j \le k$. We use the ordered set $S^i = \{(f(y_j), y_j) | 1 \le j \le k\}$ 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 $P_j \le P_k$ and $W_j \ge W_k$, 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_1^i are to be represented; provide an algorithm to merge S^i and S_1^i; 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 $\mathsf{DKP}(p, w, n, m)$
{
    $S^0 := \{(0,0)\}$;
    for $i := 1$ **to** $n-1$ **do**
    {
        $S_1^{i-1} := \{(P, W) | (P - p_i, W - w_i) \in S^{i-1}$ and $W \leq m\}$;
        $S^i :=$ 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 $W + w_n \leq m$;
    // Trace back for $x_n, x_{n-1}, \ldots, x_1$.
    if $(PX > PY)$ **then** $x_n := 0$;
    else $x_n := 1$;
    TraceBackFor$(x_{n-1}, \ldots, x_1)$;
}

Algorithm 5.6 Informal