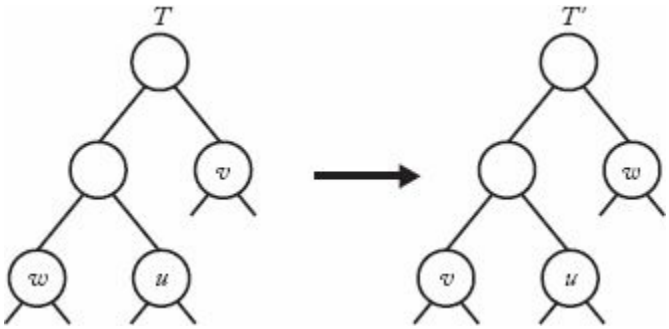
Figure 4.12 The branches rooted at v and w are swapped.



$$\begin{aligned} bits(T') &= bits(T) + [depth(w) - depth(v)]\,[frequency(v) - frequency(w)] \\ &\leq bits(T). \end{aligned}$$

which means the code corresponding to T' is optimal. Clearly, the set of trees obtained in the $(i + 1)$st step of Huffman's algorithm are branches in T'.

4.5 The Greedy Approach versus Dynamic Programming: The Knapsack Problem

The greedy approach and dynamic programming are two ways to solve optimization problems. Often a problem can be solved using either approach. For example, the Single-Source Shortest Paths problem is solved using dynamic programming in Algorithm 3.3 and is solved using the greedy approach in Algorithm 4.3. However, the dynamic programming algorithm is overkill in that it produces the shortest paths from all sources. There is no way to modify the algorithm to produce more efficiently only shortest paths from a single source because the entire array D is needed regardless. Therefore, the dynamic programming approach yields a $\Theta(n^3)$ algorithm for the problem, whereas the greedy approach yields a $\Theta(n^2)$ algorithm. Often when the greedy approach solves a problem, the result is a simpler, more efficient algorithm.

On the other hand, it is usually more difficult to determine whether a greedy algorithm always produces an optimal solution. As the Change problem shows, not all greedy algorithms do. A proof is needed to show that a particular greedy algorithm always produces an optimal solution, whereas a counterexample is needed to show that it does not. Recall that in the case of dynamic programming we need only determine whether the principle of optimality applies.

To illuminate further the differences between the two approaches, we will present two very similar problems, the 0-1 Knapsack problem and the Fractional Knapsack problem. We will develop a greedy algorithm that successfully solves the Fractional Knapsack problem but fails in the case of the 0-1 Knapsack problem. Then we will

successfully solve the 0-1 Knapsack problem using dynamic programming.

• 4.5.1 A Greedy Approach to the 0-1 Knapsack Problem

An example of this problem concerns a thief breaking into a jewelry store carrying a knapsack. The knapsack will break if the total weight of the items stolen exceeds some maximum weight W. Each item has a value and a weight. The thief's dilemma is to maximize the total value of the items while not making the total weight exceed W. This problem is called the 0-1 Knapsack problem. It can be formalized as follows.

Suppose there are n items. Let

$$\begin{aligned} S &= \{item_1, item_2, ..., item_n\} \\ w_i &= \text{ weight of } item_i \\ p_i &= \text{ profit of } item_i \\ W &= \text{ maximum weight the knapsack can hold,} \end{aligned}$$

where w_i, p_i, and W are positive integers. Determine a subset A of S such that

$$\sum_{item_i \in A} p_i \quad \text{is maximized subject to} \quad \sum_{item_i \in A} w_i \leq W.$$

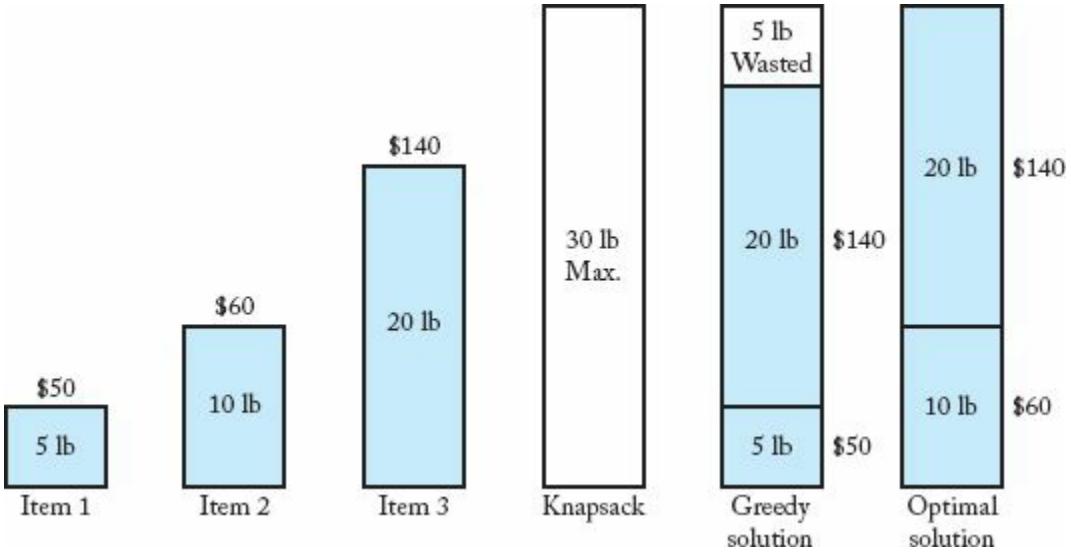
The brute-force solution is to consider all subsets of the n items; discard those subsets whose total weight exceeds W; and, of those remaining, take one with maximum total profit. Example A.10 in Appendix A shows that there are 2^n subsets of a set containing n items. Therefore, the brute-force algorithm is exponential-time.

An obvious greedy strategy is to steal the items with the largest profit first; that is, steal them in nonincreasing order according to profit. This strategy, however, would not work very well if the most profitable item had a large weight in comparison to its profit. For example, suppose we had three items, the first weighing 25 pounds and having a profit of $10, and the second and third each weighing 10 pounds and having a profit of $9. If the capacity W of the knapsack was 30 pounds, this greedy strategy would yield only a profit of $10, whereas the optimal solution is $18.

Another obvious greedy strategy is to steal the lightest items first. This strategy fails badly when the light items have small profits compared with their weights.

To avoid the pitfalls of the previous two greedy algorithms, a more sophisticated greedy strategy is to steal the items with the largest profit per unit weight first. That is, we order the items in nonincreasing order according to profit per unit weight, and select them in sequence. An item is put in the knapsack if its weight does not bring the total weight above W. This approach is illustrated in Figure 4.13. In that figure, the weight and profit for each item are listed by the item, and the value of W, which is 30, is listed in the knapsack. We have the following profits per unit weight:

Figure 4.13 A greedy solution and an optimal solution to the 0-1 Knapsack problem.



$$item_1 : \frac{\$50}{5} = \$10 \qquad item_2 : \frac{\$60}{10} = \$6 \qquad item_3 : \frac{\$140}{20} = \$7.$$

Ordering the items by profit per unit weight yields

$$item_1, item_3, item_2.$$

As can be seen in the figure, this greedy approach chooses $item_1$ and $item_3$, resulting in a total profit of \$190, whereas the optimal solution is to choose $item_2$ and $item_3$, resulting in a total profit of \$200. The problem is that after $item_1$ and $item_3$ are chosen, there are 5 pounds of capacity left, but it is wasted because $item_2$ weighs 10 pounds. Even this more sophisticated greedy algorithm does not solve the 0-1 Knapsack problem.

• 4.5.2 A Greedy Approach to the Fractional Knapsack Problem

In the Fractional Knapsack problem, the thief does not have to steal all of an item, but rather can take any fraction of the item. We can think of the items in the 0-1 Knapsack problem as being gold or silver ingots and the items in the Fractional Knapsack problem as being bags of gold or silver dust. Suppose we have the items in Figure 4.13. If our greedy strategy is again to choose the items with the largest profit per unit weight first, all of $item_1$ and $item_3$ will be taken as before. However, we can use the 5 pounds of remaining capacity to take 5/10 of $item_2$. Our total profit is

$$\$50 + \$140 + \frac{5}{10}(\$60) = \$220.$$

Our greedy algorithm never wastes any capacity in the Fractional Knapsack problem as it does in the 0-1 Knapsack problem. As a result, it always yields an optimal solution. You are asked to prove this in the exercises.

• 4.5.3 A Dynamic Programming Approach to the 0-1 Knapsack Problem

If we can show that the principle of optimality applies, we can solve the 0-1 Knapsack problem using dynamic programming. To that end, let A be an optimal subset of the n items. There are two cases: either A contains $item_n$ or it does not. If A does not contain $item_n$, A is equal to an optimal subset of the first $n - 1$ items. If A does contain $item_n$, the total profit of the items in A is equal to p_n plus the optimal profit obtained when the items can be chosen from the first $n - 1$ items under the restriction that the total weight cannot exceed $W - w_n$. Therefore, the principle of optimality applies.

The result just obtained can be generalized as follows. If for $i > 0$ and $w > 0$, we let $P[i][w]$ be the optimal profit obtained when choosing items only from the first i items under the restriction that the total weight cannot exceed w,

$$P[i][w] = \begin{cases} maximum\left(P[i-1][w], p_i + P[i-1][w-w_i]\right) & \text{if } w_i \leq w \\ P[i-1][w] & \text{if } w_i > w. \end{cases}$$

The maximum profit is equal to $P[n][W]$. We can determine this value using a two-dimensional array P whose rows are indexed from 0 to n and whose columns are indexed from 0 to W. We compute the values in the rows of the array in sequence using the previous expression for $P[i][w]$. The values of $P[0][w]$ and $P[i][0]$ are set to 0. You are asked to actually write the algorithm in the exercises. It is straightforward that the number of array entries computed is

$$nW \in \Theta(nW).$$

• 4.5.4 A Refinement of the Dynamic Programming Algorithm for the 0-1 Knapsack Problem

The fact that the previous expression for the number of array entries computed is linear in n can mislead one into thinking that the algorithm is efficient for all instances containing n items. This is not the case. The other term in that expression is W, and there is no relationship between n and W. Therefore, for a given n, we can create instances with arbitrarily large running times by taking arbitrarily large values of W. For example, the number of entries computed is in $\Theta(n \times n!)$ if W equals $n!$. If $n = 20$ and $W = 20!$, the algorithm will take thousands of years to run on a modern-day computer. When W is extremely large in comparison with n, this algorithm is worse than the brute-force algorithm that simply considers all subsets.

The algorithm can be improved so that the worst-case number of entries computed is in $\Theta(2^n)$. With this improvement, it never performs worse than the brute-force

algorithm and often performs much better. The improvement is based on the fact that it is not necessary to determine the entries in the ith row for every w between 1 and W. Rather, in the nth row we need only determine $P[n][W]$. Therefore, the only entries needed in the $(n-1)$st row are the ones needed to compute $P[n][W]$. Because

$$P[n][W] = \begin{cases} maximum\left(P[n-1][W], p_n + P[n-1][W-w_n]\right) & \text{if } w_n \le W \\ P[n-1][W] & \text{if } w_n > W, \end{cases}$$

the only entries needed in the $(n-1)$st row are

$$P[n-1][W] \qquad \text{and} \qquad P[n-1][W-w_n].$$

We continue to work backward from n to determine which entries are needed. That is, after we determine which entries are needed in the ith row, we determine which entries are needed in the $(i-1)$st row using the fact that

$$P[i][w] \qquad \text{is computed from} \qquad P[i-1][w] \qquad \text{and} \qquad P[i-1][w-w_i].$$

We stop when $n = 1$ or $w \le 0$. After determining the entries needed, we do the computations starting with the first row. The following example illustrates this method.

Example 4.9

Suppose we have the items in Figure 4.13 and $W = 30$. First we determine which entries are needed in each row.

Determine entries needed in row 3:
We need

$$P[3][W] = P[3][30].$$

Determine entries needed in row 2:
To compute $P[3][30]$, we need

$$P[3-1][30] = P[2][30] \qquad \text{and} \qquad P[3-1][30-w_3] = P[2][10].$$

Determine entries needed in row 1:
To compute $P[2][30]$, we need

$$P[2-1][30] = P[1][30] \qquad \text{and} \qquad P[2-1][30-w_2] = P[1][20].$$

To compute $P[2][10]$, we need

$$P[2-1][10] = P[1][10] \qquad \text{and} \qquad P[2-1][10-w_2] = P[1][0].$$

Next we do the computations.