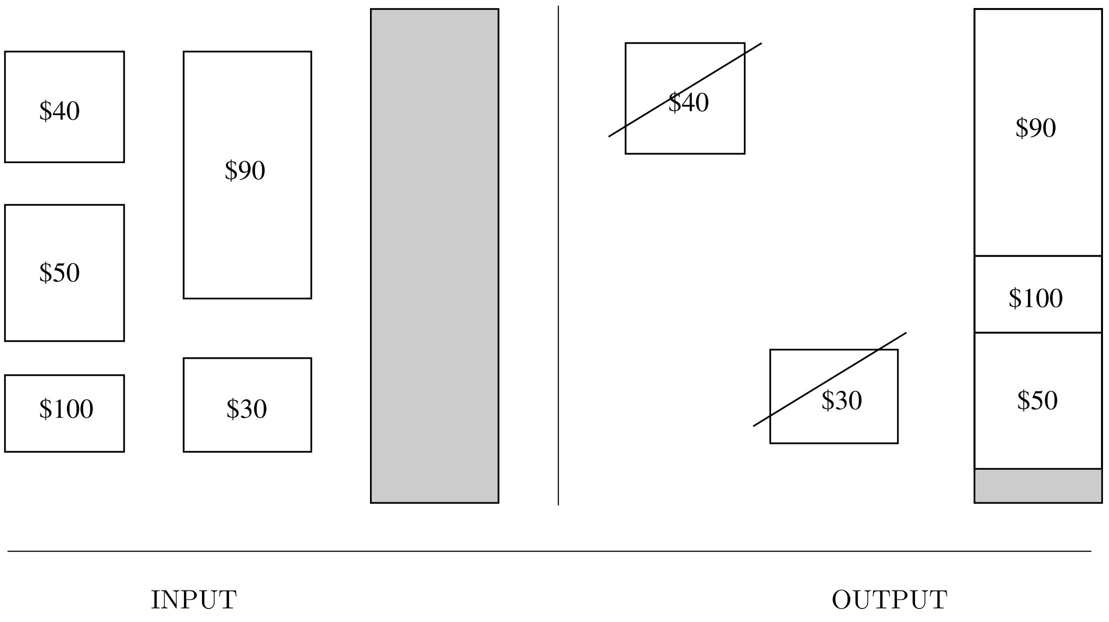

13.10 Knapsack Problem

Input description: A set of items $S = \{1, \ldots, n\}$, where item i has size s_i and value v_i. A knapsack capacity is C.

Problem description: Find the subset $S' \subset S$ that maximizes the value of $\sum_{i \in S'} v_i$, given that $\sum_{i \in S'} s_i \leq C$; i.e. , all the items fit in a knapsack of size C.

Discussion: The knapsack problem arises in resource allocation with financial constraints. How do you select what things to buy given a fixed budget? Everything has a cost and value, so we seek the most value for a given cost. The name *knapsack problem* invokes the image of the backpacker who is constrained by a fixed-size knapsack, and so must fill it only with the most useful and portable items.

The most common formulation is the *0/1 knapsack problem*, where each item must be put entirely in the knapsack or not included at all. Objects cannot be broken up arbitrarily, so it is not fair taking one can of Coke from a six-pack or opening a can to take just a sip. It is this 0/1 property that makes the knapsack problem hard, for a simple greedy algorithm finds the optimal selection when we are allowed to subdivide objects. We compute the "price per pound" for each item, and take the most expensive item or the biggest part thereof until the knapsack is full. Repeat with the next most expensive item. Unfortunately, the 0/1 constraint is usually inherent in most applications.

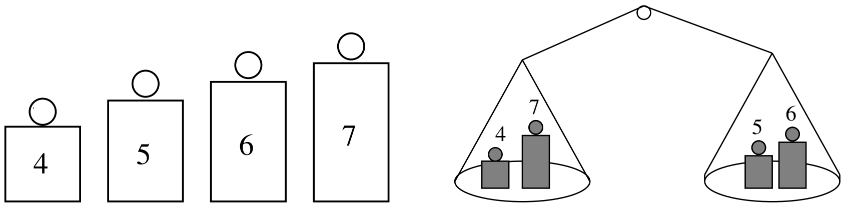

Figure 13.1: Integer partition is a special case of the knapsack problem

Issues that arise in selecting the best algorithm include:

- *Does every item have the same cost/value or the same size?* – When all items are worth exactly the same amount, we maximize our value by taking the greatest number of items. Therefore, the optimal solution is to sort the items in order of increasing size and insert them into the knapsack in this order until no more fit. The problem is similarly solved when each object has the same size but the costs are different. Sort by cost, and take the cheapest elements first. These are the easy cases of knapsack.

- *Does each item have the same "price per pound"?* – In this case, our problem is equivalent to ignoring the price and just trying to minimize the amount of empty space left in the knapsack. Unfortunately, even this restricted version is NP-complete, so we cannot expect an efficient algorithm that always solves the problem. Don't lose hope, however, because knapsack proves to be an "easy" hard problem, and one that can usually be handled with the algorithms described below.

  An important special case of a constant "price-per-pound" knapsack is the *integer partition* problem, presented in cartoon form in Figure 13.1. Here, we seek to partition the elements of S into two sets A and B such that $\sum_{a \in A} a = \sum_{b \in B}$, or alternately make the difference as small as possible. Integer partition can be thought of as bin packing into two equal-sized bins or knapsack with a capacity of half the total weight, so all three problems are closely related and NP-complete.

  The constant "price-per-pound" knapsack problem is often called the *subset sum* problem, because we seek a subset of items that adds up to a specific target number C; i.e. , the capacity of our knapsack.

- *Are all the sizes relatively small integers?* – When the sizes of the items and the knapsack capacity C are all integers, there exists an efficient dynamic programming algorithm to find the optimal solution in time $O(nC)$ and $O(C)$ space. Whether this works for you depends upon how big C is. It is great for $C \leq 1{,}000$, but not so great for $C \geq 10{,}000{,}000$.

The algorithm works as follows: Let S' be a set of items, and let $C[i, S']$ be true if and only if there is a subset of S' whose size adds up exactly to i. Thus, $C[i, \emptyset]$ is false for all $1 \leq i \leq C$. One by one we add a new item s_j to S' and update the affected values of $C[i, S']$. Observe that $C[i, S' \cup s_j] =$ true iff $C[i, S']$ or $C[i - s_j, S']$ is true, since we either use s_j in realizing the sum or we don't. We identify all sums that can be realized by performing n sweeps through all C elements—one for each s_j, $1 \leq j \leq n$—and so updating the array. The knapsack solution is given by the largest index of a true element of the largest realizable size. To reconstruct the winning subset, we must also store the name of the item number that turned $C[i]$ from false to true for each $1 \leq i \leq C$ and then scan backwards through the array.

This dynamic programming formulation ignores the values of the items. To generalize the algorithm, use each element of the array to store the value of the best subset to date summing up to i. We now update when the sum of the cost of $C[i - s_j, S']$ plus the cost of s_j is better than the previous cost of $C[i]$.

- *What if I have multiple knapsacks?* – When there are multiple knapsacks, your problem might be better thought of as a bin-packing problem. Check out Section 17.9 (page 595) for bin-packing/cutting-stock algorithms. That said, algorithms for optimizing over multiple knapsacks are provided in the Implementations section below.

Exact solutions for large capacity knapsacks can be found using integer programming or backtracking. A 0/1 integer variable x_i is used to denote whether item i is present in the optimal subset. We maximize $\sum_{i=1}^{n} x_i \cdot v_i$ given the constraint that $\sum_{i=1}^{n} x_i \cdot s_i \leq C$. Integer programming codes are discussed in Section 13.6 (page 411).

Heuristics must be used when exact solutions prove too costly to compute. The simple greedy heuristic inserts items according to the maximum "price per pound" rule described previously. Often this heuristic solution is close to optimal, but it can be arbitrarily bad depending upon the problem instance. The "price per pound" rule can also be used to reduce the problem size in exhaustive search-based algorithms by eliminating "cheap but heavy" objects from future consideration.

Another heuristic is based on *scaling*. Dynamic programming works well if the knapsack capacity is a reasonably small integer, say $\leq C_s$. But what if we have a problem with capacity $C > C_s$? We scale down the sizes of all items by a factor of C/C_s, round the size down to the nearest integer, and then use dynamic programming on the scaled items. Scaling works well in practice, especially when the range of sizes of items is not too large.

Implementations: Martello and Toth's collection of Fortran implementations of algorithms for a variety of knapsack problem variants are available at *http://www.or.deis.unibo.it/kp.html*. An electronic copy of the associated book [MT90a] has also been generously made available.

David Pisinger maintains a well-organized collection of C-language codes for knapsack problems and related variants like bin packing and container loading. These are available at *http://www.diku.dk/∼pisinger/codes.html*. The strongest code is based on the dynamic programming algorithm of [MPT99].

Algorithm 632 [MT85] of the *Collected Algorithms of the ACM* is a Fortran code for the 0/1 knapsack problem, with the twist that it supports multiple knapsacks. See Section 19.1.5 (page 659).

Notes: Keller, Pferschy, and Pisinger [KPP04] is the most current reference on the knapsack problem and variants. Martello and Toth's book [MT90a] and survey article [MT87] are standard references on the knapsack problem, including both theoretical and experimental results. An excellent exposition on integer programming approaches to knapsack problems appears in [SDK83]. See [MPT00] for a computational study of algorithms for 0-1 knapsack problems.

A polynomial-time approximation scheme is an algorithm that approximates the optimal solution of a problem in time polynomial in both its size and the approximation factor ϵ.

This very strong condition implies a smooth tradeoff between running time and approximation quality. Good expositions on polynomial-time approximation schemes [IK75] for knapsack and subset sum includes [BvG99, CLRS01, GJ79, Man89].

The first algorithm for generalized public key encryption by Merkle and Hellman [MH78] was based on the hardness of the knapsack problem. See [Sch96] for an exposition.

Related Problems: Bin packing (see page 595), integer programming (see page 411).