WIKIPEDIA
The Free Encyclopedia

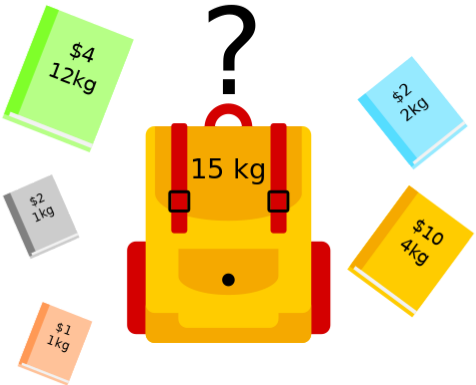
Knapsack problem

The **knapsack problem** is the following problem in combinatorial optimization:

> *Given a set of items, each with a weight and a value, determine which items to include in the collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.*

It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items. The problem often arises in resource allocation where the decision-makers have to choose from a set of non-divisible projects or tasks under a fixed budget or time constraint, respectively.

The knapsack problem has been studied for more than a century, with early works dating as far back as 1897.[1]



Example of a one-dimensional (constraint) knapsack problem: which books should be chosen to maximize the amount of money while still keeping the overall weight under or equal to 15 kg? A multiple constrained problem could consider both the weight and volume of the books.
(Solution: if any number of each book is available, then three yellow books and three grey books; if only the shown books are available, then all except for the green book.)

Applications

Knapsack problems appear in real-world decision-making processes in a wide variety of fields, such as finding the least wasteful way to cut raw materials,[2] selection of investments and portfolios,[3] selection of assets for asset-backed securitization,[4] and generating keys for the Merkle–Hellman[5] and other knapsack cryptosystems.

One early application of knapsack algorithms was in the construction and scoring of tests in which the test-takers have a choice as to which questions they answer. For small examples, it is a fairly simple process to provide the test-takers with such a choice. For example, if an exam contains 12 questions each worth 10 points, the test-taker need only answer 10 questions to achieve a maximum possible score of 100 points. However, on tests with a heterogeneous distribution of point values, it is more difficult to provide choices. Feuerman and Weiss proposed a system in which students are given a heterogeneous test with a total of 125 possible points. The students are asked to answer all of the questions to the best of their abilities. Of the possible subsets of problems whose total point values add up to 100, a knapsack algorithm would determine which subset gives each student the highest possible score.[6]

A 1999 study of the Stony Brook University Algorithm Repository showed that, out of 75 algorithmic problems related to the field of combinatorial algorithms and algorithm engineering, the knapsack problem was the 19th most popular and the third most needed after suffix trees and the bin packing problem.[7]

Definition

The most common problem being solved is the **0-1 knapsack problem**, which restricts the number x_i of copies of each kind of item to zero or one. Given a set of n items numbered from 1 up to n, each with a weight w_i and a value v_i, along with a maximum weight capacity W,

$$\text{maximize} \sum_{i=1}^{n} v_i x_i$$

$$\text{subject to} \sum_{i=1}^{n} w_i x_i \leq W \text{ and } x_i \in \{0, 1\}.$$

Here x_i represents the number of instances of item i to include in the knapsack. Informally, the problem is to maximize the sum of the values of the items in the knapsack so that the sum of the weights is less than or equal to the knapsack's capacity.

The **bounded knapsack problem** (**BKP**) removes the restriction that there is only one of each item, but restricts the number x_i of copies of each kind of item to a maximum non-negative integer value c:

Our assignment problem

Replace $v_i \to -v_i$ and then maximize (Thus absolute val minimize, assignment solved)

$$\text{maximize} \sum_{i=1}^{n} v_i x_i$$

$$\text{subject to} \sum_{i=1}^{n} w_i x_i \leq W \text{ and } x_i \in \{0, 1, 2, \ldots, c\}.$$

The **unbounded knapsack problem** (**UKP**) places no upper bound on the number of copies of each kind of item and can be formulated as above except that the only restriction on x_i is that it is a non-negative integer.

$$\text{maximize} \sum_{i=1}^{n} v_i x_i$$

$$\text{subject to} \sum_{i=1}^{n} w_i x_i \leq W \text{ and } x_i \in \mathbb{Z},\ x_i \geq 0.$$

One example of the unbounded knapsack problem is given using the figure shown at the beginning of this article and the text "if any number of each book is available" in the caption of that figure.

Computational complexity

The knapsack problem is interesting from the perspective of computer science for many reasons:

- The decision problem form of the knapsack problem (*Can a value of at least* V *be achieved without exceeding the weight* W*?*) is NP-complete, thus there is no known algorithm that is both correct and fast (polynomial-time) in all cases.
- There is no known polynomial algorithm which can tell, given a solution, whether it is optimal (which would mean that there is no solution with a larger *V*). This problem is co-NP-complete.
- There is a pseudo-polynomial time algorithm using dynamic programming.
- There is a fully polynomial-time approximation scheme, which uses the pseudo-polynomial time algorithm as a subroutine, described below.
- Many cases that arise in practice, and "random instances" from some distributions, can nonetheless be solved exactly.

There is a link between the "decision" and "optimization" problems in that if there exists a polynomial algorithm that solves the "decision" problem, then one can find the maximum value for the optimization problem in polynomial time by applying this algorithm iteratively while increasing the value of k. On the other hand, if an algorithm finds the optimal value of the optimization problem in polynomial time, then the decision problem can be solved in polynomial time by comparing the value of the solution output by this algorithm with the value of k. Thus, both versions of the problem are of similar difficulty.

One theme in research literature is to identify what the "hard" instances of the knapsack problem look like,[8][9] or viewed another way, to identify what properties of instances in practice might make them more amenable than their worst-case NP-complete behaviour suggests.[10] The goal in finding these "hard" instances is for their use in public key cryptography systems, such as the Merkle-Hellman knapsack cryptosystem.

Furthermore, notable is the fact that the hardness of the knapsack problem depends on the form of the input. If the weights and profits are given as integers, it is weakly NP-complete, while it is strongly NP-complete if the weights and profits are given as rational numbers.[11] However, in the case of rational weights and profits it still admits a fully polynomial-time approximation scheme.

Unit-cost models

The NP-hardness of the Knapsack problem relates to computational models in which the size of integers matters (such as the Turing machine). In contrast, decision trees count each decision as a single step. Dobkin and Lipton[12] show an $\frac{1}{2}n^2$ lower bound on linear decision trees for the knapsack problem, that is, trees where decision nodes test the sign of affine functions.[13] This was generalized to algebraic decision trees by Steele and Yao.[14] If the elements in the problem are real numbers or rationals, the decision-tree lower bound extends to the real random-access machine model with an instruction set that includes addition, subtraction and multiplication of real numbers, as well as comparison and either division or remaindering ("floor").[15] This model covers more algorithms than the algebraic decision-tree model, as it encompasses algorithms that use indexing into tables. However, in this model all program steps are counted, not just decisions. An *upper bound* for a decision-tree model was given by Meyer auf der Heide[16] who showed that for every *n* there exists an $O(n^4)$-deep linear decision tree that solves the subset-sum problem with *n* items. Note that this does not imply any upper bound for an algorithm that should solve the problem for *any given n*.

Solving

Several algorithms are available to solve knapsack problems, based on the dynamic programming approach,[17] the branch and bound approach[18] or hybridizations of both approaches.[10][19][20][21]

Dynamic programming in-advance algorithm

The **unbounded knapsack problem** (**UKP**) places no restriction on the number of copies of each kind of item. Besides, here we assume that $x_i > 0$

$$m[w'] = \max\left(\sum_{i=1}^{n} v_i x_i\right)$$

$$\text{subject to } \sum_{i=1}^{n} w_i x_i \leq w' \text{ and } x_i > 0$$

Observe that $m[w]$ has the following properties:

1. $m[0]=0$ (the sum of zero items, i.e., the summation of the empty set).

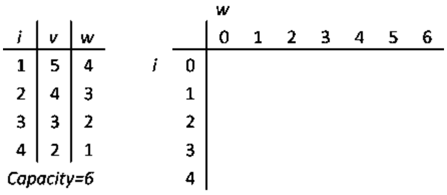
2. $m[w]=\max(v_1+m[w-w_1],v_2+m[w-w_2],\ldots,v_n+m[w-w_n])$, $w_i \le w$, where v_i is the value of the i-th kind of item.

The second property needs to be explained in detail. During the process of the running of this method, how do we get the weight w? There are only i ways and the previous weights are $w-w_1, w-w_2, \ldots, w-w_i$ where there are total i kinds of different item (by saying different, we mean that the weight and the value are not completely the same). If we know each value of these i items and the related maximum value previously, we just compare them to each other and get the maximum value ultimately and we are done.

Here the maximum of the empty set is taken to be zero. Tabulating the results from $m[0]$ up through $m[W]$ gives the solution. Since the calculation of each $m[w]$ involves examining at most n items, and there are at most W values of $m[w]$ to calculate, the running time of the dynamic programming solution is $O(nW)$. Dividing $w_1,\ w_2,\ \ldots,\ w_n,\ W$ by their greatest common divisor is a way to improve the running time.

Even if P≠NP, the $O(nW)$ complexity does not contradict the fact that the knapsack problem is NP-complete, since W, unlike n, is not polynomial in the length of the input to the problem. The length of the W input to the problem is proportional to the number of bits in W, $\log W$, not to W itself. However, since this runtime is pseudopolynomial, this makes the (decision version of the) knapsack problem a weakly NP-complete problem.

0-1 knapsack problem



i	v	w
1	5	4
2	4	3
3	3	2
4	2	1

Capacity=6

i \ w	0	1	2	3	4	5	6
0							
1							
2							
3							
4							

A demonstration of the dynamic programming approach.

A similar dynamic programming solution for the 0-1 knapsack problem also runs in pseudo-polynomial time. Assume $w_1,\ w_2,\ \ldots,\ w_n,\ W$ are strictly positive integers. Define $m[i,w]$ to be the maximum value that can be attained with weight less than or equal to w using items up to i (first i items).

We can define $m[i,w]$ recursively as follows: **(Definition A)**

- $m[0,\ w]=0$
- $m[i,\ w]=m[i-1,\ w]$ if $w_i > w$ (the new item is more than the current weight limit)
- $m[i,\ w]=\max(m[i-1,\ w],\ m[i-1,w-w_i]+v_i)$ if $w_i \leqslant w$.

The solution can then be found by calculating $m[n,W]$. To do this efficiently, we can use a table to store previous computations.

The following is pseudocode for the dynamic program:

```
// Input:
// Values (stored in array v)
// Weights (stored in array w)
// Number of distinct items (n)
// Knapsack capacity (W)
// NOTE: The array "v" and array "w" are assumed to store all relevant values starting at index 1.

array m[0..n, 0..W];
for j from 0 to W do:
    m[0, j] := 0
for i from 1 to n do:
    m[i, 0] := 0

```

```
for i from 1 to n do:
    for j from 1 to W do:
        if w[i] > j then:
            m[i, j] := m[i-1, j]
        else:
            m[i, j] := max(m[i-1, j], m[i-1, j-w[i]] + v[i])
```

This solution will therefore run in $O(nW)$ time and $O(nW)$ space. (If we only need the value m[n,W], we can modify the code so that the amount of memory required is O(W) which stores the recent two lines of the array "m".)

However, if we take it a step or two further, we should know that the method will run in the time between $O(nW)$ and $O(2^n)$. From **Definition A**, we know that there is no need to compute all the weights when the number of items and the items themselves that we chose are fixed. That is to say, the program above computes more than necessary because the weight changes from 0 to W often. From this perspective, we can program this method so that it runs recursively.

```
// Input:
// Values (stored in array v)
// Weights (stored in array w)
// Number of distinct items (n)
// Knapsack capacity (W)
// NOTE: The array "v" and array "w" are assumed to store all relevant values starting at index 1.

Define value[n, W]

Initialize all value[i, j] = -1

Define m:=(i,j)         // Define function m so that it represents the maximum value we can get under the condition: use first i items, total weight limit is j
{
    if i == 0 or j <= 0 then:
        value[i, j] = 0
        return

    if (value[i-1, j] == -1) then:     // m[i-1, j] has not been calculated, we have to call function m
        m(i-1, j)

    if w[i] > j then:                      // item cannot fit in the bag
        value[i, j] = value[i-1, j]
    else:
        if (value[i-1, j-w[i]] == -1) then:     // m[i-1,j-w[i]] has not been calculated, we have to call function m
            m(i-1, j-w[i])
        value[i, j] = max(value[i-1,j], value[i-1, j-w[i]] + v[i])
}

Run m(n, W)
```

For example, there are 10 different items and the weight limit is 67. So,

$$w[1]=23, w[2]=26, w[3]=20, w[4]=18, w[5]=32, w[6]=27, w[7]=29, w[8]=26, w[$$
$$v[1]=505, v[2]=352, v[3]=458, v[4]=220, v[5]=354, v[6]=414, v[7]=498, v[8]=54$$

If you use above method to compute for $m(10,67)$, you will get this, excluding calls that produce $m(i,j)=0$:

$$m(10,67)=1270$$
$$m(9,67)=1270, m(9,40)=678$$
$$m(8,67)=1270, m(8,40)=678, m(8,37)=545$$
$$m(7,67)=1183, m(7,41)=725, m(7,40)=678, m(7,37)=505$$
$$m(6,67)=1183, m(6,41)=725, m(6,40)=678, m(6,38)=678, m(6,37)=505$$
$$m(5,67)=1183, m(5,41)=725, m(5,40)=678, m(5,38)=678, m(5,37)=505$$
$$m(4,67)=1183, m(4,41)=725, m(4,40)=678, m(4,38)=678, m(4,37)=505, m(4,3$$
$$m(3,67)=963, m(3,49)=963, m(3,41)=505, m(3,40)=505, m(3,38)=505, m(3,37$$
$$m(2,67)=857, m(2,49)=857, m(2,47)=505, m(2,41)=505, m(2,40)=505, m(2,38$$
$$m(1,67)=505, m(1,49)=505, m(1,47)=505, m(1,41)=505, m(1,40)=505, m(1,38$$

Besides, we can break the recursion and convert it into a tree. Then we can cut some leaves and use parallel computing to expedite the running of this method.

To find the actual subset of items, rather than just their total value, we can run this after running the function above:

```
/**
 * Returns the indices of the items of the optimal knapsack.
 * i: We can include items 1 through i in the knapsack
 * j: maximum weight of the knapsack
 */
function knapsack(i: int, j: int): Set<int> {
    if i == 0 then:
        return {}
    if m[i, j] > m[i-1, j] then:
        return {i} ∪ knapsack(i-1, j-w[i])
    else:
        return knapsack(i-1, j)
}

knapsack(n, W)
```

Meet-in-the-middle

Another algorithm for 0-1 knapsack, discovered in 1974[22] and sometimes called "meet-in-the-middle" due to parallels to a similarly named algorithm in cryptography, is exponential in the number of different items but may be preferable to the DP algorithm when W is large compared to n. In particular, if the w_i are nonnegative but not integers, we could still use the dynamic programming algorithm by scaling and rounding (i.e. using fixed-point arithmetic), but if the problem requires d fractional digits of precision to arrive at the correct answer, W will need to be scaled by 10^d, and the DP algorithm will require $O(W10^d)$ space and $O(nW10^d)$ time.

```
algorithm Meet-in-the-middle is
    input: A set of items with weights and values.
    output: The greatest combined value of a subset.

    partition the set {1...n} into two sets A and B of approximately equal size
    compute the weights and values of all subsets of each set

    for each subset of A do
        find the subset of B of greatest value such that the combined weight is less than W

    keep track of the greatest combined value seen so far
```

The algorithm takes $O(2^{n/2})$ space, and efficient implementations of step 3 (for instance, sorting the subsets of B by weight, discarding subsets of B which weigh more than other subsets of B of greater or equal value, and using binary search to find the best match) result in a runtime of $O(n2^{n/2})$. As with the meet in the middle attack in cryptography, this improves on the $O(n2^n)$ runtime of a naive brute force approach (examining all subsets of $\{1...n\}$), at the cost of using exponential rather than constant space (see also baby-step giant-step). The current state of the art improvement to the meet-in-the-middle algorithm, using insights from Schroeppel and Shamir's Algorithm for Subset Sum, provides as a corollary a randomized algorithm for Knapsack which preserves the $O^*(2^{n/2})$ (up to polynomial factors) running time and reduces the space requirements to $O^*(2^{0.249999n})$ (see [23] Corollary 1.4). In contrast, the best known deterministic algorithm runs in $O^*(2^{n/2})$ time with a slightly worse space complexity of $O^*(2^{n/4})$.[24]

Approximation algorithms

As for most NP-complete problems, it may be enough to find workable solutions even if they are not optimal. Preferably, however, the approximation comes with a guarantee of the difference between the value of the solution found and the value of the optimal solution.

As with many useful but computationally complex algorithms, there has been substantial research on creating and analyzing algorithms that approximate a solution. The knapsack problem, though NP-Hard, is one of a collection of algorithms that can still be approximated to any specified degree. This means that the problem has a polynomial time approximation scheme. To be exact, the knapsack problem has a fully polynomial time approximation scheme (FPTAS).[25]

Greedy approximation algorithm

George Dantzig proposed a greedy approximation algorithm to solve the unbounded knapsack problem.[26] His version sorts the items in decreasing order of value per unit of weight, $v_1/w_1 \geq \cdots \geq v_n/w_n$. It then proceeds to insert them into the sack, starting with as many copies as possible of the first kind of item until there is no longer space in the sack for more. Provided that there is an unlimited supply of each kind of item, if m is the maximum value of items that fit into the sack, then the greedy algorithm is guaranteed to achieve at least a value of $m/2$.

For the bounded problem, where the supply of each kind of item is limited, the above algorithm may be far from optimal. Nevertheless, a simple modification allows us to solve this case: Assume for simplicity that all items individually fit in the sack ($w_i \leq W$ for all i). Construct a solution S_1 by packing items greedily as long as possible, i.e. $S_1 = \{1, \ldots, k\}$ where $k = \max_{1 \leq k' \leq n} \sum_{i=1}^{k'} w_i \leq W$. Furthermore, construct a second solution $S_2 = \{k+1\}$ containing the first item that did not fit. Since $S_1 \cup S_2$ provides an upper bound for the LP relaxation of the problem, one of the sets must have value at least $m/2$; we thus return whichever of S_1 and S_2 has better value to obtain a $1/2$-approximation.

It can be shown that the average performance converges to the optimal solution in distribution at the error rate $n^{-1/2}$ [27]

Fully polynomial time approximation scheme

The fully polynomial time approximation scheme (FPTAS) for the knapsack problem takes advantage of the fact that the reason the problem has no known polynomial time solutions is because the profits associated with the items are not restricted. If one rounds off some of the least significant digits of the profit values then they will be bounded by a polynomial and 1/ε where ε is a bound on the correctness of the solution. This restriction then means that an algorithm can find a solution in polynomial time that is correct within a factor of (1-ε) of the optimal solution.[25]

```
algorithm FPTAS is
    input: ε ∈ (0,1]
           a list A of n items, specified by their values, $v_i$, and weights
    output: S' the FPTAS solution

    P := max $\{v_i \mid 1 \leq i \leq n\}$  // the highest item value
    K := ε $\frac{P}{n}$

    for i from 1 to n do
        $v'_i$ := $\left\lfloor \frac{v_i}{K} \right\rfloor$
    end for

    return the solution, S', using the $v'_i$ values in the dynamic program outlined above
```

Theorem: The set S' computed by the algorithm above satisfies $\mathrm{profit}(S') \geq (1-\varepsilon) \cdot \mathrm{profit}(S^*)$, where S^* is an optimal solution.

Dominance relations

Solving the unbounded knapsack problem can be made easier by throwing away items which will never be needed. For a given item i, suppose we could find a set of items J such that their total weight is less than the weight of i, and their total value is greater than the value of i. Then i cannot appear in the optimal solution, because we could always improve any potential solution containing i by replacing i with the set J. Therefore, we can disregard the i-th item altogether. In such cases, J is said to **dominate** i. (Note that this does not apply to bounded knapsack problems, since we may have already used up the items in J.)

Finding dominance relations allows us to significantly reduce the size of the search space. There are several different types of dominance relations,[10] which all satisfy an inequality of the form:

$$\sum_{j \in J} w_j\,x_j \ \leq \alpha\,w_i, \text{ and } \sum_{j \in J} v_j\,x_j \ \geq \alpha\,v_i \text{ for some } x \in Z^n_+$$

where $\alpha \in Z_+$, $J \subsetneq N$ and $i \notin J$. The vector x denotes the number of copies of each member of J .

Collective dominance

The i-th item is **collectively dominated** by J, written as $i \ll J$, if the total weight of some combination of items in J is less than w_i and their total value is greater than v_i. Formally, $\sum_{j \in J} w_j\,x_j \ \leq w_i$ and $\sum_{j \in J} v_j\,x_j \ \geq v_i$ for some $x \in Z^n_+$, i.e. $\alpha = 1$. Verifying this dominance is computationally hard, so it can only be used with a dynamic programming approach. In fact, this is equivalent to solving a smaller knapsack decision problem where $V = v_i$, $W = w_i$, and the items are restricted to J.

Threshold dominance

The i-th item is **threshold dominated** by J, written as $i \prec\!\prec J$, if some number of copies of i are dominated by J. Formally, $\sum_{j \in J} w_j\, x_j \ \leq \alpha\, w_i$, and $\sum_{j \in J} v_j\, x_j \ \geq \alpha\, v_i$ for some $x \in Z_+^n$ and $\alpha \geq 1$. This is a generalization of collective dominance, first introduced in[17] and used in the EDUK algorithm. The smallest such α defines the **threshold** of the item i, written $t_i = (\alpha - 1)w_i$. In this case, the optimal solution could contain at most $\alpha - 1$ copies of i.

Multiple dominance

The i-th item is **multiply dominated** by a single item j, written as $i \ll_m j$, if i is dominated by some number of copies of j. Formally, $w_j\, x_j \ \leq w_i$, and $v_j\, x_j \ \geq v_i$ for some $x_j \in Z_+$ i.e. $J = \{j\}, \alpha = 1, x_j = \lfloor \frac{w_i}{w_j} \rfloor$. This dominance could be efficiently used during preprocessing because it can be detected relatively easily.

Modular dominance

Let b be the *best item*, i.e. $\frac{v_b}{w_b} \geq \frac{v_i}{w_i}$ for all i. This is the item with the greatest density of value. The i-th item is **modularly dominated** by a single item j, written as $i \ll_\equiv j$, if i is dominated by j plus several copies of b. Formally, $w_j + tw_b \leq w_i$, and $v_j + tv_b \geq v_i$ i.e. $J = \{b, j\}, \alpha = 1, x_b = t, x_j = 1$.

Variations

There are many variations of the knapsack problem that have arisen from the vast number of applications of the basic problem. The main variations occur by changing the number of some problem parameter such as the number of items, number of objectives, or even the number of knapsacks.

Multi-objective knapsack problem

This variation changes the goal of the individual filling the knapsack. Instead of one objective, such as maximizing the monetary profit, the objective could have several dimensions. For example, there could be environmental or social concerns as well as economic goals. Problems frequently addressed include portfolio and transportation logistics optimizations.[28][29]

As an example, suppose you ran a cruise ship. You have to decide how many famous comedians to hire. This boat can handle no more than one ton of passengers and the entertainers must weigh less than 1000 lbs. Each comedian has a weight, brings in business based on their popularity and asks for a specific salary. In this example, you have multiple objectives. You want, of course, to maximize the popularity of your entertainers while minimizing their salaries. Also, you want to have as many entertainers as possible.

Multi-dimensional knapsack problem

In this variation, the weight of knapsack item i is given by a D-dimensional vector $\overline{w_i} = (w_{i1}, \ldots, w_{iD})$ and the knapsack has a D-dimensional capacity vector $(W_1, \ldots, W_D)$. The target is to maximize the sum of the values of the items in the knapsack so that the sum of weights in each dimension d does not exceed W_d.

Multi-dimensional knapsack is computationally harder than knapsack; even for $D = 2$, the problem does not have EPTAS unless P=NP.[30] However, the algorithm in[31] is shown to solve sparse instances efficiently. An instance of multi-dimensional knapsack is sparse if there is a set $J = \{1, 2, \ldots, m\}$ for $m < D$ such that for every knapsack item i, $\exists z > m$ such that

$\forall j \in J \cup \{z\}$, $w_{ij} \geq 0$ and $\forall y \notin J \cup \{z\}, w_{iy} = 0$. Such instances occur, for example, when scheduling packets in a wireless network with relay nodes.[31] The algorithm from[31] also solves sparse instances of the multiple choice variant, multiple-choice multi-dimensional knapsack.

The IHS (Increasing Height Shelf) algorithm is optimal for 2D knapsack (packing squares into a two-dimensional unit size square): when there are at most five square in an optimal packing.[32]

Multiple knapsack problem

This variation is similar to the Bin Packing Problem. It differs from the Bin Packing Problem in that a subset of items can be selected, whereas, in the Bin Packing Problem, all items have to be packed to certain bins. The concept is that there are multiple knapsacks. This may seem like a trivial change, but it is not equivalent to adding to the capacity of the initial knapsack. This variation is used in many loading and scheduling problems in Operations Research and has a Polynomial-time approximation scheme.[33]

Quadratic knapsack problem

The quadratic knapsack problem maximizes a quadratic objective function subject to binary and linear capacity constraints.[34] The problem was introduced by Gallo, Hammer, and Simeone in 1980,[35] however the first treatment of the problem dates back to Witzgall in 1975.[36]

Subset-sum problem

The subset sum problem is a special case of the decision and 0-1 problems where each kind of item, the weight equals the value: $w_i = v_i$. In the field of cryptography, the term *knapsack problem* is often used to refer specifically to the subset sum problem. The subset sum problem is one of Karp's 21 NP-complete problems.[37]

A generalization of subset sum problem is called multiple subset-sum problem, in which multiple bins exist with the same capacity. It has been shown that the generalization does not have an FPTAS.[38]

Geometric knapsack problem

In the geometric knapsack problem, there is a set of rectangles with different values, and a rectangular knapsack. The goal is to pack the largest possible value into the knapsack.[39]

See also

Computer programming portal

- Bin packing problem – Mathematical and computational problem
- Change-making problem – the computational problem of choosing as few coins as possible that add up to a given amount of money
- Combinatorial auction – smart market in which participants can place bids on combinations of discrete items, rather than individual items or continuous quantities
- Combinatorial optimization – Subfield of mathematical optimization
- Continuous knapsack problem