Algorithmics of the Knapsack Type Tasks

Pătcaş Csaba

patcas.csaba@gmail.com

BBTE

Abstract. We propose a new kind of approach of the teaching of knapsack type problems in the classroom. We will remind you the context of the general knapsack-task and we will classify it. We will focus on the solving of the discrete variant, and we will determine the complexity of the algorithms, looking for different optimiziation possibilities. All these issues are presented in a useful way for highschool teachers, who are preparing students in order to participate in different programming contests.

1. General task

We will present - in a general manner - the text of the one-dimensional knapsack task:

Let us consider a knapsack of volume V and n items, with volumes v_i *and costs* c_i, ($i = 1, 2, \ldots, n$). *Determine the subset of the given items, which has the total volume either smaller or equal to the knapsack's capacity and for which the sum of the costs is* ***optimal***.

Before going to the different variants in which the given task might be encountered (in giving an explicit form to the task's general terms), let us see how these variants could be classified. From a methodological point of view (while teaching) it is also recommended to make the students aware from the beginning that they must pay special attention to the differences that might appear - most of them hidden, wrapped in a story - in various task texts.

2. Classifying

An ideal case would be that this classifying, the „discovery" of the differences between the problem statements to be sort out together with the students, while studying more text variants. Obviously, from the point of view of task type, we could add texts that only seem different, and yet they are part of the same category. The classifying of this task's variants is realized on the following three criteria: from the point of view of *continuity*, from the point of view of the *number of items* and from the point of view of the *optimal filling* of the knapsack.

From the continuity point of view of the knapsack task, this can be given in both the *discrete* and the *continuous* variant.

a) **the discrete variant**: we have to select only *uncut* items.

b) **the continuous (fractional) variant**: in this case we are *not* forced to select an item as a whole, because the items *can be cut*. In this case, if an item is cut, we gain a cost proportional to the selected volume. As known, it is usually solved with the help of a *greedy* type algorithm.

From the point of view of the number of items we have three cases:

a) **the unbounded variant**: we have an *infinite number* of pieces corresponding to each item type.

b) **the bounded variant**: each i item must be used at least l_i times and at most u_i times ($i = 1, 2, \ldots, n$).

c) **the binary variant (0-1)**: we have a special case of the bounded task in which $l_i=0$ and $u_i=1$, where from each item we only have one piece. The naming (0-1) of the binary variant comes from the fact that an item *can* appear in the knapsack once or it *can not* appear at all.

If the teacher is lucky enough the students will notice by themselves the following:

- In the bounded variant it does not make sense for the values l_i and u_i ($i = 1, 2, \ldots, n$) to be higher than $[V/v_i]$. Thus, the unbounded variant can be changed into the bounded variant, by setting the inferior limits $l_i=0$ and the superior limits $u_i=[V/v_i]$.
- We can simplify the bounded variant, by setting the inferior limits $l_i=0$, a case in which the superior limits u_i will be changed in u_i-l_i ($i = 1, 2, \ldots, n$). In this case the value V has to be fixed on $V – (l_1 \cdot v_1 + l_2 \cdot v_2 + \ldots + l_n \cdot v_n)$. In this way we will lose the inferior limits.
- The bounded variant can be changed in the binary one. We will consider the text of the binary variant, saying that from each type of item we have u_i ($i = 1, 2, \ldots, n$) pieces.

The teacher will emphasize these extremely important remarks, because of the fact that ***all the discrete variants of the task can be reduced to the binary variant.*** Out of this reason, we will further focus on the solving of this variant, only.

From the point of view of the *optimal filling of the knapsack*, we have four cases:

a) **Case (max, ≤)**: in such tasks we are interested in the item configuration, for which the costs' sum is maximal and the knapsack does not necessarily have to be full.

b) **Case (max, =)**: similar to the precedent case, differing in that we are interested only in the solutions for which the sum of the volumes of the selected items is equal to the knapsack's volume.

c) Case (min, ≤): here the sum of the costs must be minimal, and the knapsack does not have to be full.

d) Case (min, =): the sum of the volumes of the items must be equal to the knapsack's volume and the costs' sum must be minimal.

The teacher will present a lot of tasks and will ask the students to decide which kind type they are.

3. The coins task

As we already mentioned, the discrete variant can be solved by using the *dynamic programming method*. We will start from a simplified variant of the one-dimensional task.

Let us consider n coins, where the i^{th} has the v_i (i= 1, 2, …, n) value. The v_i values are strictly positive integer numbers. Determine if it is possible or not the exact payment of the V sum, using these coins!

We used on purpose the same notations as in the first text, the one of the knapsack task, in order to facilitate the noticing of the likeness of the two tasks. We hope, that the students will notice, that the simplifying consists in the elimination of each item's cost, so there is no need to optimize another value, we must just answer the question. So we have the case (∅, =), where ∅ indicates the fact that we do not have an optimum function. (The classical variant of the task asks for the sum to be obtained by using a minimal number of coins. Thus a cost that has to be minimized is introduced, and in this case, this will be $c_i = 1$, $i = 1, 2, \ldots, n$).

The teacher will remark that, in the case in which the values *vi* are real numbers, the coins task is NP-complete. Actually, there are no polynomial algorithms for any of the discrete variants of the knapsack task. The solutions using the dynamic programming method have pseudopolynomial complexity, because these algorithms have an $\Theta(V \cdot n)$ complexity. That means that the execution time does not only depend on the number n of the items, but also on the knapsack's volume V. The complexity is not polynomial, because it is possible for the value V to exponentially increase as compared to the value n. So, if the value V is very high, we have an inappropriate complexity. Even if the value V is low, we can have a big number of items, so the execution time can increase in this case, too.

So we have to decide whether or not the given sum payment is possible. As we mentioned above, in order to answer to this question, we will use the dynamic programming method. We will build a matrix a (a helping data structure) of $(n+1) \cdot (V+1)$ dimensions:

a_{ij} = **true**, if sum j can be obtained using the first i type of coins

false, otherwise

$i = 0, 1, ..., n$, $j = 0, 1, ..., V$. Obviously, for $i = 0$, the only element with **true** value will be a_{00} (with zero coins you can only get the sum zero). The other values can be thus determined:

a_{ij} = **true**, if $a_{i-1,j-vi}$ = **true**

$a_{i-1,j}$, otherwise

After this analysis and the preparation of the needed formulas, the solving algorithm can be presented by a student. A negative aspect of this solution is the fact that it utilizes a space memory of $\Theta(V \cdot n)$. But, skillfully, a teacher could lead the analyzes of the solving so that a student observes that *for building a line in the matrix, only the precedent one is used*. Thus, it seems natural the "lucky" idea of not keeping in memory the whole matrix of $(n+1)\times(V+1)$ dimensions, but to memorize just two vectors, one in which the actual line is kept and the other for the precedent one. Each time we start the determination of a new line of the matrix, we will overwrite the vector of the precedent line with the vector of the actual one, so that the determination of a new line will be possible.

Further on, we will present the most elegant method, which uses only one vector and which is based on the following remark: in a certain sum we can have the same coin many times, only if first we already got a smaller sum by using at least once the respective coin. Thus we come to the idea of calculating the sum in decreasing order of the values (we will refer to this as the 1st variant).

4. Optimization possibilities

Another optimizing possibility is memorizing the highest and the lowest value achieved at a certain point, and parsing the values only in this interval. But when do these values modify? One can notice that in the case of the coins' task the minimum always remains zero, because the coins have positive values. Thus, for our task we do not even need this minimum. The maximum will always be the sum of the first i elements, if this sum does not exceed the V value.

Because the b vector contains Boolean values, we can utilize bit processing. Thus, we will reduce the dimension of the needed memory from V bytes to $[V/8]$ bytes. In the second variant we implement this idea, by using two subprograms to read/write a certain bit.

This method has one more advantage because of the fact that the vector from the i^{th} step can be obtained by shifting to right the vector from the $(i-1)^{th}$ step with vi positions (bits) and making a logical disjunction between the initial vector and the vector got after the shift. More exactly, we will utilize a formula like this: $a_i = a_i - 1$ or $(a_i - 1 \text{ shr } v_i)$. Using this property, we do not have to calculate the a vector bit by bit; we can utilize bigger steps (byte by byte, word by word etc.) significantly improving the program's running speed. This is due to the fact that bit operations are very fast (3rd variant).

Let us see how these three variants perform in practice. Does the improved memory usage result in a worse running time? We measured the performance of the variants using a randomly generated test case (with n=5000 and V=50000), on two sets of implementations: using an old 16-bit programming language (Borland Pascal) and a modern 32-bit programming language (Visual C++). The following tables contain the results.

Borland Pascal

Variant	Memory usage	Running time
1	60000 bytes	2,5s
2	16250 bytes	4s
3	16250 bytes	0,5s

Visual C++ (Release mode)

Variant	Memory usage	Running time
1	60000 bytes	0,2s
2	16250 bytes	0,4s
3	16250 bytes	0,05s

In conclusion the programming language used, didn't affect the results. The third variant was the fastest and it also used the least memory.

5. The reconstruction of the solution

In the coins' task text, it was asked to decide only if the sum can be paid or not. We might need also the method through which we got to a certain sum, not just the checking of the fact that this can be obtained. The determination of paying modalities can be either easily or with much difficulty realized, depending on the size of input data and the available memory.

The first idea would be to memorize in a matrix called *ind* the highest index of a coin that is part of that sum, for each obtained sum. This method provides the correct result only in the case of utilizing matrices.

In all of the other versions we cannot reconstruct the solution using this method for all cases. As a counter-example let us consider n=3, V=7 and v=(5,2,3). Using the first coin, we can pay the sums 0 and 5. Using the first two coins, we can get 0, 2, 5 and 7. But in the next iteration we have a problem. (Here we will remind the students, that we are discussing the binary variant of the knapsack problem.) Using the first three coins, we will pay the sum 5 with the second and the third coin, so when trying to pay the sum 7 we will use the second coin two times which is not allowed (7 = 2 + 3 + 2 instead of 7 = 2 + 5). This problem arises with algorithms which are allowed to overwrite previously achieved values.

Even if we do not allow the program to overwrite values that have been achieved, the solution won't work in the classical variant of the problem, when we must pay the sum with minimum number of coins. Let us consider the following example n=5, V=23 and v=(19,7,7,7,2). We observe that the sum 21 can be achieved minimally with two coins (19 + 2). In the same time the sum 23 cannot be obtained by the algorithm, which is wrong, because 23 = 7 + 7 + 7 + 2.

In conclusion, the correct solution uses two matrices that contain n+1 lines and V+1 columns, the *a* matrix, and the *ind* matrix with the meanings described above. So the solution needs an $\Theta(n \cdot V)$ memory space, while the execution time is also $\Theta(n \cdot V)$ (1st variant).

If the memory limits are harder, we could improve this method by memorizing the matrix only from k to k lines and by running of the algorithm "on parts" of k lines. More exactly, after we have determined the $[n/k]$ lines, we reconstruct the solution starting from the last memorized line (the $([n/k])^{th}$) and constructing the next k lines. Afterwards, we start from the one before last memorized line (the $([n/k]-1)^{th}$) and we determine again the following k lines etc. Thus, we will need an $\Theta(k \cdot V+[n/k] \cdot V)$ memory space, while the algorithm's execution time will be $\Theta(n \cdot V+k \cdot [n/k] \cdot V)=\Theta(n \cdot V)$. It is noticeable that the best choice for k is $n^{1/2}$. In this case we have a memory space of $\Theta(n^{1/2} \cdot V)$.

Still – sometimes we must find a solution that is able to use less memory. We will present a solution that uses a memory space of $\Theta(V)$ and runs in $\Theta(n \cdot V)$ time**.** The two matrices will be constructed by using an implementation similar to the one presented above, with the following change: instead of a vector with the significance of the *ind* vector we will use another vector (named *goto*), where $goto_j$ has the value of the partial sum of the j sum, wherein there are all the coins used for obtaining the j sum, which have the indexes smaller than or equal to $[n/2]$ (j=0,…,V). For better understanding, we can think of the vector *goto*, as a "super-parent vector", that for values of i greater than $[n/2]$ "points back" to line $[n/2]$ of the matrix. The construction of the *goto* vector is realized in the following way:

$goto_j = j$ for $i<=[n/2]$

$goto_{j-vi}$ otherwise

Thus, after the execution of the n iterations, the $goto_V$ value will represent the sum of the elements in the searched solution, if coins whose indexes are at most equal to $[n/2]$ are used.

By using this value, we will apply the *Divide and Conquer* method and we will solve the task for the first $[n/2]$ coins and the $goto_V$ sum, respectively the rest of the coins and the $V - goto_V$ sum.

Let us analyze the complexity of this algorithm. Each subtask is made of two subtasks, one of $[n/2]$ dimension and $goto_V$ and one of $n-[n/2]$ dimension and $V - goto_V$. A subtask of n dimension and V has the $\Theta(n \cdot V)$ complexity. So, the execution time necessary for the algorithm is given by the recursive formula: $T(n,V)=\Theta(n \cdot V)+T([n/2],gotoV)+T(n-[n/2],V-gotoV)$, where we obtain $T(n,V)=\Theta(n \cdot V)$. The proof of this is based on the usual methods, the most comfortable being the *substitution method* or applying the *Master Theorem*.

The practical behavior of these variants is summarized in the following table. This time only Borland Pascal implementations were used, on a random test case with n=250 and V=2000. The algorithm was run 100 times, so the running time became big enough to be measureable.

Variant	Memory complexity	Memory usage	Running time
1	$O(n \cdot V)$	502751 bytes	0,6s
2	$O(n^{0.5} \cdot V)$	136068 bytes	0,9s
3	$O(V)$	16508 bytes	1,4s

References

1. T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, MIT Massachussetts, 2001.
2. K. Ionescu – Introduction to Algorithms, University Press, Cluj, 2005 (in Hungarian).
3. Cs. Pătcaş, Algorithmics of the Knapsack Type Tasks, GInfo, Agora Media, Tg. Mureş, 15/6 (2006), 31–35 (in Romanian).
4. C. Silvestru-Negruşeri, Optimization of the Programs Using Bit Operations, GInfo, Agora Media, Tg. Mureş, 14/5 (2004), 32–36 (in Romanian).