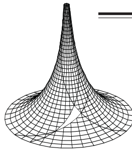


**INFODIDACT** (2008), 37–71
tmcs@inf.unideb.hu
http://tmcs.math.klte.hu
Teaching
Mathematics and
Computer Science

# Algorithmics of the knapsack type tasks

Csaba Pătcaş and Klára Ionescu

*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, including the two most popular task variants: the discrete and the continuous one. Once we briefly present the solving algorithm of the continuous variant, we will focus on the solving of the discrete task, and we will determine the complexity of the algorithms, looking for different optimizing 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.

*Key words and phrases:* teaching, algorithm, knapsack, dynamic programming method, optimization.

*ZDM Subject Classification:* P54, P55.

## 1. Introduction

The highschool educational system in Romania has an over 30 years tradition in teaching algorithmics. The participant students in different Olympiads in Informatics manifest a particular interest towards the special algorithmics and programming knowledge, since they want to learn new methods of program optimization, both from the point of view of time execution, and the needed memory dimension. The tasks known under the name of “knapsack type” constitute a real challenge due to numberless variants in which they can appear and to the diversity of solving possibilities.

Copyright © 2008 by University of Debrecen

## 2. 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.

Of course, if the students have already problem solving skills, they will observe immediately, that this task is an "optimality" one. Also, we assume, that they are familiarized with the greedy and the dynamic programming method [1].

## 3. 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.

(1) **The discrete variant**: we have to select only *uncut* items.

(2) **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:

(1) **The unbounded variant**: we have an *infinite number* of pieces corresponding to each item type.

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

(3) **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 $[V/v_i]$[1], ($i = 1, 2, \ldots, n$). 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 $V - (l_1 \cdot v_1 + l_2 \cdot v_1 + \ldots + l_n \cdot v_n)$. 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:

(1) **Case (max, $\leq$)**: 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.

(2) **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.

[1] We marked with $[x]$ the integer part of $x$.

(3) **Case (min, $\leq$):** here the sum of the costs must be minimal, and the knapsack does not have to be full.

(4) **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.

## 4. Solving the knapsack task—the continuous variant

Once the task is formulated (probably not in a formal way, but wrapped in a real story) we will consider an example. Let us consider $V = 10$, $n = 4$ and $v = (5, 4, 5, 10)$ (the items' volumes), and $c = (10, 20, 1, 15)$ (their costs). We will ask the students to initiate different ideas of their own. Maybe somebody will propose to select the items in the decreasing order of the costs and fill the knapsack with the second item, while the fourth one will be cut. One of the students will calculate the total cost of the knapsack: $c_2 + c_4 \cdot ((V - v_2)/v_4) = 20 + 15 \cdot (6/10) = 24.5$. The teacher knows that this is not the optimal value, so he/she will lead the analysis of the ideas of the students so that they will reach the conclusion themselves: before starting the packing of the items in the knapsack they have to calculate the values $c_i/v_i$ $(i = 1, 2, 3, 4)$ that in the case above are $(2, 5, 0.2, 1.5)$. The students familiarized with the steps of the *greedy* method will realize that they have to sort this vector and to select the items in the obtained order, until the next item can not be selected any longer, because the knapsack is full. It can happen, that the last selected item we have to cut, to fill the knapsack entirely. If we follow these steps in the given example, we obtain the second and the first item from the original array. From the fourth item a piece of 1 dimension will be cut and with this the knapsack will be filled. Thus, the total value of the packed items will be $c_2 + c_1 + ((V - v_2 - v_1)/v_4) = 20 + 10 + (15/10) = 31.5$, which is obviously more than 24.5 obtained on the first idea.

While teaching we have to introduce the necessary formalizing in order to give the algorithm and to prove the fact that this algorithm will reach the *global optimum* [2].

Let us consider that a certain solution is a vector $x = (x_1, x_2, \dots, x_n)$ where the elements $x_i$ $(i = 1, 2, \dots, n)$ have the properties:

$$\begin{cases} x_i \in [0,1], \quad i = 1, 2, \dots, n \\ \sum\limits_{i=1}^{n} v_i x_i \le V. \end{cases}$$

Out of these, an *optimal solution* is the one that maximizes the function

$$f(x) = \sum_{i=1}^{n} c_i x_i.$$

If the sum of the volumes of the items is smaller than $V$, we have the trivial case in which we will select all the items, meaning that the solution will be: $x = (1, 1, \dots, 1)$. Further on we will suppose that $v_1 + v_2 + \dots + v_n > V$.

Due to the *greedy* strategy, we order the items decreasingly accordingly to the profit of the unit volume, so we consider the hypothesis:

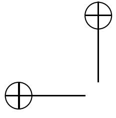
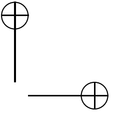
$$\frac{c_1}{v_1} \ge \frac{c_2}{v_2} \ge \dots \ge \frac{c_n}{v_n} \qquad (*)$$

The algorithm consists in selecting the items in this order, as long as the volume $V$ is not overloaded (if the last item does have room in the knapsack, it will be loaded only partially):

**Algorithm 1** ContKnapsack($n$, *Vol*, $v$, $x$)

▷ input: $n$ – the number of the given items
▷ *Vol* – the total volume of the knapsack
▷ $v$ – the array of the given items' volumes
▷ output: $x$ – the array of the solution, representing the selected items

1: $VD \leftarrow Vol$ ▷ $VD$ = the disposable weight
2: $term \leftarrow$ `false`
3: $i \leftarrow 1$
4: **while not** *term* **and** $(i \le n)$ **do**
5:     **if** $v_i \le VD$ **then** ▷ corresponding to those items
6:         $x_i \leftarrow 1$ ▷ which will be put entirely in the knapsack
7:         $VD \leftarrow VD - v_i$
8:     **else**
9:         $x_i \leftarrow VD/v_i$ ▷ corresponding to that item which will be cut
10:         **for** $j \leftarrow i+1, n$ **do** ▷ corresponding to those items





**Algorithm 1** continued

```
11:         x_j ← 0                     ▷ which will not be put in the knapsack
12:       end for
13:       term ← true
14:     end if
15:     i ← i + 1
16: end while
```

The solution given by the algorithm has the form $x = (1, 1, \dots, 1, x_j, 0, \dots, 0)$, where $1 \le j \le n$ and $x_j \in [0, 1)$. We will prove that this solution is optimal.

Let us suppose that "$x$ is not optimal" and there is another certain solution $y = (y_1, y_2, \dots, y_n)$ having the property: $\sum_{i=1}^{n} c_i y_i > \sum_{i=1}^{n} c_i x_i$. It can be considered without loss of generality, that $\sum_{i=1}^{n} v_i y_i = V$.

Let us suppose that $y \ne x$, and $k$ is the first position in which $y_k \ne x_k$. We will show that $y_k < x_k$. There are three possibilities [7]:

(1) $k < j$, that means $x_k = 1$; but $y_k \ne x_k$, so $y_k < 1$; results $y_k < x_k$.

(2) $k = j$

$$\begin{aligned} V = \sum_{i=1}^{n} v_i y_i &= \sum_{i=1}^{j-1} v_i y_i + v_j y_j + \sum_{i=j+1}^{n} v_i y_i \\ &= \sum_{i=1}^{j-1} v_i x_i + v_j y_j + \sum_{i=j+1}^{n} v_i y_i = V - v_j x_j + v_j y_j + \sum_{i=j+1}^{n} v_i y_i \\ &= V + v_j (y_j - x_j) + \sum_{i=j+1}^{n} v_i y_i. \end{aligned}$$

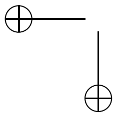
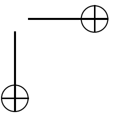
If $y_j > x_j$,

$$V = \sum_{i=1}^{n} v_i y_i = V + v_j (y_j - x_j) + \sum_{i=j+1}^{n} v_i y_i > V.$$

Impossible. Results $y_j < x_j$, and also $y_k < x_k$.

(3) $k > j$

$$\sum_{i=1}^{n} v_i y_i = \sum_{i=1}^{k-1} v_i y_i + \sum_{i=k}^{n} v_i y_i = \sum_{i=1}^{k-1} v_i x_i + \sum_{i=k}^{n} v_i y_i$$

$$= \sum_{i=1}^{j} v_i x_i + \sum_{i=j+1}^{k-1} v_i x_i + \sum_{i=k}^{n} v_i y_i = V + 0 + \sum_{i=k}^{n} v_i y_i > V.$$

Impossible.

Now we will increase $y_k$ in order to obtain $x_k$ and decrease $y_{k+1}, y_{k+2}, \ldots, y_n$ in order to have $\sum_{i=1}^{n} v_i y_i = V$ (the total capacity of the knapsack remains $V$). We obtain a solution $Z = (z_1, z_2, \ldots, z_n)$ where $z_i = x_i$, $i = 1, \ldots, k$ and

$$\sum_{i=k+1}^{n} v_i(y_i - z_i) = v_k(z_k - y_k).$$

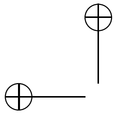
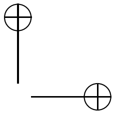
The sum of the decreasing's weight of $y_{k+1}, y_{k+2}, \ldots, y_n$ is equal to the increasing's weight of $y_k$. For $Z$ we have:

$$\begin{aligned}
\sum_{i=1}^{n} c_i z_i &= \sum_{i=1}^{n} c_i y_i + (z_k - y_k)c_k + \sum_{i=k+1}^{n} (z_i - y_i)c_i \\
&= \sum_{i=1}^{n} c_i y_i + (z_k - y_k)v_k c_k/v_k - \sum_{i=k+1}^{n} (y_i - z_i)v_i c_i/v_i \\
&\geq \sum_{i=1}^{n} c_i y_i + [(z_k - y_k)v_k - \sum_{i=k+1}^{n} (y_i - z_i)v_i]c_k/v_k \\
&= \sum_{i=1}^{n} c_i y_i > \sum_{i=1}^{n} c_i x_i.
\end{aligned}$$

Here we used the $(*)$ properties.

If $Z = X$, results an impossible equation: $\sum_{i=1}^{n} c_i x_i > \sum_{i=1}^{n} c_i x_i$, so the initial assumption that "$X$ is not an optimal solution" is false. If $Z \neq X$, we repeat the described process with $Z$ in place of $Y$ and we continue it until $Z = X$, i.e. the hypothesis that "$X$ is not an optimal solution" is false.

*Remark.* It could be relevant to observe—together with the students—that the application of the *greedy* method fails in the case of the discrete variant. For example, if $V = 10$, $n = 3$ and $v = (8, 6, 4)$, $c = (12, 8, 5)$ we would obtain as a result (for selecting the first item) the profit 12. But the selection of the last two items leads to the superior profit that is 13.



## 5. The discrete variant of the knapsack task

In order to show that the task could not be solved by the *greedy* method, the teacher will give an example for this like we did in the remark above. 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, and then we will take in turns the *one-dimensional*, *two-dimensional* and the *multidimensional* variant.

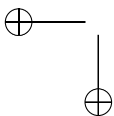
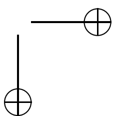
### 5.1. The coins task (one-dimensional simplified knapsack)

*Let us consider $n$ coins, where the $i^{th}$ has the $v_i$ $(i = 1, 2, \ldots, n)$ 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 $(\emptyset, =)$, where $\emptyset$ 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 $v_i$ are real numbers, the coins task is NP-complete[2]. Actually, there are no polynomial algorithms for any of the discrete variants of the knapsack task. The solutions using the dynamic programming method have pseudo-polynomial 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.

[2] We consider that both the students and the reader are familiarized with algorithm complexity notions [1].

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 $(n+1) \times (V+1)$ dimensions:

$$a_{ij} = \begin{cases} \texttt{true}, & \text{if sum } j \text{ can be obtained using the first } i \text{ type of coins,} \\ \texttt{false} & \text{otherwise,} \end{cases}$$

$i = 0, 1, \ldots, n$, $j = 0, 1, \ldots, V$. 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:

$$a_{ij} = \begin{cases} \texttt{true}, & \text{if } a_{i-1,j} = \texttt{true} \text{ or } a_{i-1,j-v_i} = \texttt{true}, \\ a_{i-1,j} & \text{otherwise.} \end{cases}$$

After this analysis and the preparation of the needed formulas, the solving algorithm can be presented by a student. The Knapsack1($n$, *Vol*, $v$, $a$) algorithm builds the $a$ matrix of logical values, in which the value of the $a_{nV}$ element will contain the answer to the question (`true`/`false`).

**Algorithm 2** Knapsack1($n$, *Vol*, $v$, $a$)

```
                                   ▷ input: n – the number of the given items
                                   ▷ Vol – the total volume of the knapsack
                                   ▷ v – the array of the given items' volumes
                                   ▷ output: a – two-dimensional Boolean array where
                                   ▷ a_{n,Vol} will represent the final result
 1: a_00 ← true                    ▷ with zero coins we can get the sum zero
 2: for j ← 1, Vol do
 3:     a_0j ← false               ▷ with zero coins we can not get sum j
 4: end for
 5: for i ← 1, n do
 6:     a_i ← a_{i-1}   ▷ we assign to the i^th line of the matrix the (i-1)^th line
 7:     for j ← v_i, Vol do        ▷ we build the matrix a
 8:         if a_{i-1,j-v_i} then
 9:             a_ij ← true
10:         end if
11:     end for
12: end for
```

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 [3]. Thus, we obtain the KNAPSACK2($n$, *Vol*, $v$, *new*) algorithm:

**Algorithm 3** KNAPSACK2($n$, *Vol*, $v$, *new*)

```
                                  ▷ input: n – the number of the given items
                                  ▷ Vol – the total volume of the knapsack
                                  ▷ v – the array of the given items' volumes
          ▷ output: new – array, where new_Vol will represent the final result
 1: old_0 ← true                            ▷ local: old – auxiliary array
 2: for j ← 1, Vol do
 3:     old_j ← false
 4: end for
 5: new ← old   ▷ initially the second line of the matrix equals to the first one
 6: for i ← 1, n do
 7:     for j ← v_i, Vol do                   ▷ the new vector is built by using
                                  ▷ the corresponding values of the old vector
 8:         if old_{j−v_i} then
 9:             new_j ← true
10:         end if
11:     end for
12:     old ← new
13: end for
```

But if we succeeded in reducing the necessary memory from a matrix with $n+1$ lines of $V+1$ elements to two vectors with $V+1$ elements, we will ask ourselves (and the students, of course) if there is the possibility of further reducing the data structure's dimension? But, when trying to solve the problem using just one vector, the students should observe the possibility of obtaining a certain sum, by using one coin *more than one time* (which is obviously wrong). For instance,

if we have $V = 4$ and $v = (2)$, when verifying if we can obtain the value 4, we will find that it is possible, because the second element of the vector is already `true`.

If the students will not propose it, the teacher will show that one solution is to memorize in another vector the highest coins' index that form the respective sum: $ind_j$ = *the index of the last coin that was used to obtain the j sum*, ($j = 0, 1, 2, \ldots, V$). Thus, we obtain the third variant of the solving algorithm:

**Algorithm 4** KNAPSACK3($n$, $Vol$, $v$, $b$)

▷ input: $n$ – the number of the given items
▷ $Vol$ – the total volume of the knapsack
▷ $v$ – the array of the given items' volumes
▷ output: $b$ – similar to the new vector from the previous algorithm
▷ where $b_{Vol}$ represents the final result

1: $b_0 \leftarrow$ `true`
2: $ind_0 \leftarrow -1$ ▷ we have to put there a value different from 0,
▷ we will see later why
3: **for** $j \leftarrow 1, Vol$ **do**
4: $b_j \leftarrow$ `false`
5: $ind_j \leftarrow 0$ ▷ local: $ind$ – auxiliary array
6: **end for**
7: **for** $i \leftarrow 1, n$ **do**
8: **for** $j \leftarrow v_i, Vol$ **do** ▷ we build the vectors $b$ and $ind$
9: **if** $b_{j-v_i}$ **and** $(ind_{j-v_i} \neq i)$ **then**
10: $b_j \leftarrow$ `true`
11: $ind_j \leftarrow i$
12: **end if**
13: **end for**
14: **end for**

The students should observe that it seems that no improvement has been done, because we still use two vectors. Maybe, when they will take a closer look they will notice that we do not need anymore the $b$ vector, because $b_j$ ($j = 0, 1, 2, \ldots, V$) is `true` only for the values for which $ind_j$ is different from zero. That explains the 3. line of the algorithm.

The implementing of this variant can be a homework for the students (or for the reader).

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. The final variant of the solution is presented in the KNAPSACK4($n$, $Vol$, $v$, $b$) algorithm. In the work-field this method is known as *Pull dynamic programming.*

**Algorithm 5** KNAPSACK4($n$, $Vol$, $v$, $b$)

▷ input: $n$ – the number of the given items
▷ $Vol$ – the total volume of the knapsack
▷ $v$ – the array of the given items' volumes
▷ output: $b$ – array, where $b_{Vol}$ represents the final result

```
1: b_0 ← true
2: for j ← 1, Vol do
3:     b_j ← false
4: end for
5: for i ← 1, n do
6:     for j = Vol, v_i, −1 do
7:         if b_{j−v_i} then
8:             b_j ← true
9:         end if
10:    end for
11: end for
```

### 5.2. Other optimization possibilities

Let us see if we can further optimize this solution. We notice that, for $V$ and $v_i$ $(i = 1, 2, \ldots, n)$ if they are high enough, we have "holes" in the $b$ vector, meaning that we obtain only a very small part of the sums from 0 to $V$, but still we parse all the interval $n$ times.

To improve this fact one has to parse only the values that are achieved at a certain step. This thing can be implemented with chained lists in the programming languages which do not offer either the set type or the possibility of using the iterators (like the STL from C++ for instance).

We mention the fact that this change most of the times leads to speed increase, in detriment of the dimension of the needed memory. The memory usage is increased by the following factors. In the case of the implementation with chained lists it is increased by the "next" pointers needed for the lists and by the fact that

we do not only store a Boolean value for each sum (that occupies usually a byte in the older programming languages) but a value that can occupy more bytes. In the case of the second implementation the memory usage is increased by the way iterators are usually implemented. In the work-field this method is known as *Reach* or *Push dynamic programming.*

Let us go back to the *Pull* variant. Another optimizing possibility is memorizing the highest and the lowest value achieved at a certain point, and parsing the values only in this interval [3]. 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. Thus we get to the solution given by the KNAPSACK5($n$, $Vol$, $v$, $b$) algorithm:

**Algorithm 6** KNAPSACK5($n$, $Vol$, $v$, $b$)

▷ input: $n$ – the number of the given items
▷ $Vol$ – the total volume of the knapsack
▷ $v$ – the array of the given items' volumes
▷ output: $b$ – array, where $b_{Vol}$ represents the final result

```
1:  b_0 ← true
2:  for j ← 1, Vol do
3:      b_j ← false
4:  end for
5:  max ← 0                       ▷ local: max – the sum of the first i elements
6:  for i ← 1, n do
7:      if max > Vol − v_i then
8:          max ← Vol − v_i
9:      end if
10:     for j ← max, 0, −1 do     ▷ the max value will be recalculated for each i
                                  ▷ in order to reduce the number of steps in this for cycle
11:         if b_j then
12:             b_{j+v_i} ← true
13:         end if
14:     end for
15:     max ← max + v_i
16: end for
```

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.

This method has one more advantage because of the fact that the vector from the $i^{\text{th}}$ step can be obtained by shifting to right the vector from the $(i-1)^{\text{th}}$ step with $v_i$ 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** $(a_{i-1}$ **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 [4].

Let us see some applications.

**Task 1 – Sticks 1**

Let us consider $n$ sticks ($5 \leq n \leq 1000$). The sticks have different lengths, not necessarily distinct ones ($1 \leq \mathit{length}_i \leq 100$, $i = 1, 2, \ldots, n$, the lengths are given in millimeters). The sticks must be grouped, so that those from one group form a "line" whose length should be as close as possible to the "line" formed by the sticks from the other group. The length of the line is equal to the lengths' sum of the sticks that form the line.

Determine the length of the two lines formed by the sticks arranged in the two groups, so that the difference between the length of the "line" formed by the sticks from the first group and the length of the "line" formed by the sticks from the second group is as small as possible.

*Example:* $n = 7$, $\mathit{length} = (28, 7, 11, 8, 9, 7, 27)$, the length of the first line of sticks: 48 $(= 28 + 11 + 9)$, the length of the other one: 49 $(= 7 + 8 + 7 + 27)$.

*Solution*

The analysis of the task begins with its formalizing. The students will notice that we have a vector of integer (not necessarily distinct) numbers which must be divided in two groups so that the difference between the two sums is minimal. It is possible that the students notice that apparently there is an NP-complete task, and consequently they will want to apply the *backtracking* method. We will calculate the sum of all the numbers, and then its half. It is clear that the two sums that have to be calculated should be "around" this half. But we cannot know for sure which ones these numbers will be. We build the following Boolean vector: each number from the initial *length* vector could be a partial sum, so we mark the corresponding elements of the Boolean vector with `true`. Obviously, we do not have to implement this explicitly, because the algorithm will do it, if we

set $used_0$ to `true`. We also mark in the same way any element in the vector that has the index a partial sum of the read numbers. Thus it seems that this vector should have the length equal to the sum of all the elements. Another remark leads us to realizing that we do not have to keep the whole vector. We will focus only on those elements that cover the first half, and we will calculate the second number from the total sum and the biggest number smaller or equal to the half which is marked with `true`. Because in the task's text we had soft restrictions for the limits, we will present at first the unoptimized variant of the algorithm:

**Algorithm 7** DIVIDE($n$, *length*, *half*)

▷ input: $n$ – the number of the sticks
▷ *length* – the array with the sticks' length
▷ output: *half* – the length of the first line of the sticks

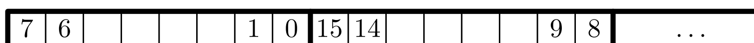
1: $sum \leftarrow 0$ ▷ local: *sum* – the total length of the sticks
2: **for** $i \leftarrow 1, n$ **do**
3: $sum \leftarrow sum + length_i$
4: **end for**
5: $half \leftarrow [sum/2]$
6: **for** $i \leftarrow 1, half$ **do**
7: $used_i \leftarrow$ `false` ▷ local: *used* – auxiliary Boolean vector
8: **end for**
9: $used_0 \leftarrow$ `true`
10: $sum \leftarrow 0$
11: **for** $i \leftarrow 1, n$ **do**
12: **for** $j \leftarrow \mathrm{Min}(sum, half), 0, -1$ **do** ▷ in order not to parse the whole
▷ Boolean vector, but only the useful part of it
13: **if** $used_j$ **then**
14: $used_{j+length_i} \leftarrow$ `true`
15: **end if**
16: **end for**
17: $sum \leftarrow sum + length_i$ ▷ it is necessary to recalculate sum
▷ in order to reduce the number of the steps
▷ we use the same idea as in the KNAPSACK5($n$, *Vol*, $v$, $b$) algorithm
18: **end for**
19: **while not** $used_{half}$ **and** $(half > 0)$ **do**
▷ searching of the existing value for *half*

**Algorithm 7** continued

20: $half \leftarrow half - 1$
21: **end while**

After the execution of this algorithm the values *half* and $sum - half$ will be printed.

In case we had had not 1000, but 2000 sticks, certain compilers would not have let us to allocate such a big *used* array ($2000 \times 100 = 200000$, from which half is 100000). In this case a vector of bits will be used. Usually we number the bits from left to right starting with the most significant one. Thus the most insignificant one (the 0 value) will be the rightmost.




This solution we present in Pascal. The vector of bits is implemented in the `bits` type, where the bits are grouped in bytes.

```
type sticks = array[1..2000] of Word;
     bits   = array[0..12500] of Byte;

var n:   Word;
    Vol: Word;
    v:   sticks;
    b:   bits;

function GetBit(var v: bits; ind: Word): Boolean;
  begin
    GetBit := v[ind shr 3] and (1 shl (ind and 7)) <> 0
               { where v[ind shr 3] was used instead of v[ind div 8] and }
                        { 1 shl (ind and 7) instead of 1 shl (ind mod 8) }
  end;

procedure SetBit(var v: bits; ind: Word);
  begin
    v[ind shr 3] := v[ind shr 3] or (1 shl (ind and 7))
  end;

function min(a, b: Word): Word;
  begin
    if a < b then min := a
    else min := b
  end;
```

```
procedure Divide(n: Word; var length: sticks; var half: Word);
                                   { input: n - the number of the sticks }
                              { length - the array of the sticks' length }
             { output: half - the length of the first line of the sticks }
  var i,j:  Word;
      sum:  Longint;       { local: sum - the total length of the sticks }
      used: bits;
  begin
    sum := 0;
    for i:=1 to n do
      Inc(sum, length[i]);
    half := sum div 2;
    FillChar(used, SizeOf(used), 0);
    SetBit(used, 0);
    sum := 0;
    for i:=1 to n do begin
      for j := min(sum, half) downto 0 do
        if GetBit(used, j) then
          SetBit(used, j+length[i]);
        Inc(sum, length[i])
    end;
    while not GetBit(used, half) and (half > 0) do
      Dec(half)
  end;
```

**Task 2 – Sums**

On a table there are $n$ ($1 \leq n \leq 100$) cards. On each card there is written one natural number between 1 and 10000. One can choose as many cards from the table as wanted. After this, the sum of the natural numbers from the chosen cards must be calculated.

Determine the number of the distinct sums that can be obtained through such selections. It is possible to select as many cards (but at least 1).

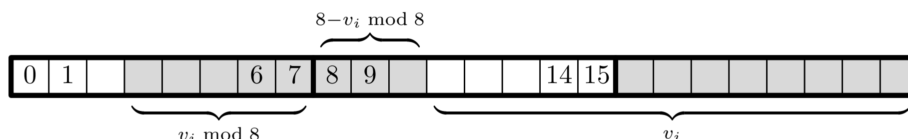
*Example:* $n = 3$, $numbers = (1, 3, 7)$, $result = 7$, that are: 1, 3, 4 $(= 1 + 3)$, 7, 8 $(= 1 + 7)$, 10 $(= 3 + 7)$, 11 $(= 1 + 3 + 7)$.

*Solution*

This task can be solved by keeping a Boolean vector of values which indicates if a certain sum can be (or cannot be) obtained. Because we have at most 100 numbers whose values are at most 10000, the maximum sum that can be obtained is 1000000, so we need 1000001 values in the vector. Thus, we must use just one bit to represent a sum, and so we reduce the necessary memory to 125001 bytes.

Initially we will consider that only the sum 0 can be obtained. Further on, for each next step we will consider one more number. We will parse the vector of the previously obtained sums (bit by bit) and, to each bit with the value 1, we will add the current number to the respective sum. The result of the operation will represent a sum which can surely be obtained, so for the next step we will have to set the corresponding bit for the new sum to value 1. Finally, we will number the value 1 bits from the whole vector and we will write down the result.

Because of the hard limits, an implementation similar to the one used in the problem $\text{Sticks}_1$ could be too slow. We present a faster solution in Pascal. To help an easier implementation, at this time we will consider the bits of a byte to be ordered from left to right.




```
type cards = array[1..100] of Word;
     bits  = array[-1..125000] of Byte;
                               { we start from -1 to avoid special cases }
          { we consider compilers, that allow us to allocate this array, }
                                { to not over-complicate the source code }
var n:   Word;
    v:   cards;
    b:   bits;
    sum: Longint;

procedure Build(n: Word; var v: cards; var new: bits; var max: Longint);
  var i, aux, aux2: Byte;
      j:            Longint;
      old:          bits;
  begin
    FillChar(old, SizeOf(old), 0);                 { set everything to 0 }
    old[0] := 128;                         { we can always get the sum 0 }
    max := 0;
    for i := 1 to n do begin
      aux := v[i] shr 3;
      aux2 := v[i] and 7;
      for j := aux to (max+v[i]) shr 3 do
        new[j] := Byte(old[j - aux - 1] shl (8 - aux2)) or
                                                  old[j - aux] shr aux2;
                                    { we need this type cast here to stay }
```

```
                           { in the bounds of a byte and avoid overflow }
      for j := aux to (max+v[i]) shr 3 do
        old[j] := old[j] or new[j];
      max := max + v[i]
    end;
    new := old
  end;

function Count(var b: bits; sum: Longint): Longint;   { here we will use }
    { a little trick for counting quickly a byte's bits that are set [4] }
  var i, ans: Longint;
      aux:    Byte;
  begin
    ans := 0;
    for i := 0 to sum shr 3 do begin
      aux := b[i];
      while aux > 0 do begin
        Inc(ans);
        aux := aux and (aux-1)
      end
    end;
    count := ans
  end;
```

## 5.3. 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 the highest index of a coin that is part of that sum, for each obtained sum, similarly to the KNAPSACK3($n$, $Vol$, $v$, $b$) algorithm presented above.

This method provides the correct result only in the case of utilizing matrices (the first variant of the algorithm—KNAPSACK1($n$, $Vol$, $v$, $a$)). 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, such as the Knapsack3($n$, $Vol$, $v$, $b$) algorithm.

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 $23 = 7+7+7+2$.

We will remind the students again the fact that now we focus on the binary variant of the task. In the case of the unbounded variant, the reconstruction of the solution can be easily done by using a memory quantity of $\Theta(V)$, with the help of a vector similar to *ind* vector from the third variant of the algorithm—Knapsack3($n$, $Vol$, $v$, $b$).

The solution based on these remarks is presented in the Knapsack6($n$, $Vol$, $v$, $parent$) algorithm. During the design of the algorithm, the teacher will take care of the students not to "forget" about the optimizations learned before.

**Algorithm 8** Knapsack6($n$, $Vol$, $v$, $b$)

▷ input: $n$ – the number of the given items
▷ $Vol$ – the total volume of the knapsack
▷ $v$ – the array of the given items' volumes
▷ output: $parent$ – two-dimensional array,
▷ needed in the reconstruction of the solution

```
1:  a_00 ← true
2:  for j ← 1, Vol do
3:      a_0j ← false
4:  end for
5:  max ← 0
6:  for i ← 1, n do
7:      if max > Vol − v_i then
8:          max ← Vol − v_i
9:      end if
10:     a_i ← a_{i−1}
11:     parent_i ← parent_{i−1}
```

**Algorithm 8** continued

```
12:     for j ← 0, max do
13:         if a_{i−1,j} then
14:             a_{i,j+v_i} ← true
15:             parent_{i,j+v_i} ← i
16:         end if
17:     end for
18:     max ← max + v_i
19: end for
```

The modality through which the reconstruction is realized is presented in the ReBuild($n$, *Vol*, $v$, *parent*) algorithm:

**Algorithm 9** Rebuild($n$, *Vol*, $v$, $b$)

```
                                        ▷ input: n – the number of the given items
                                        ▷ Vol – the total volume of the knapsack
                                        ▷ v – the array of the given items' volumes
                                        ▷ output: parent – two-dimensional array,
                                        ▷ needed in the reconstruction of the solution
1: i ← n
2: j ← Vol
3: while j > 0 do
4:     aux ← parent_{ij}
5:     Write: aux
6:     i ← aux − 1    ▷ the aux^th item will surely not be part of the next j sum
7:     j ← j − v_aux
8: end while
```

It is noticeable that this solution uses two matrices that contain $n + 1$ lines and $V + 1$ columns, so the solution needs an $\Theta(n \cdot V)$ memory space, while the execution time is also $\Theta(n \cdot V)$.

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 $([n/k])^{\text{th}}$) and constructing the next $k$ lines. Afterwards, we start from the one before last memorized line (the $([n/k] - 1)^{\text{th}}$) 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 $\Theta(n^{1/2} \cdot V)$. This idea can be treated like in the following algorithm:

**Algorithm 10** KNAPSACK7($n$, $k$, $Vol$, $v$, $parent$, $bigparent$)

▷ input: $n$ – the number of the given items, $k$ – square root of $n$
▷ $Vol$ – the total volume of the knapsack
▷ $v$ – the array of the given items' volumes
▷ output: $parent$ – two-dimensional array
▷ needed in the reconstruction of the solution
▷ it stores the lines between two consecutive lines stored in bigparent
▷ output: $bigparent$ – two-dimensional array, needed in the reconstruction
▷ of the solution; it stores every $k^{\text{th}}$ line of the solution
▷ both matrices need to have at most $\sqrt{n}+1$ lines and $Vol+1$ columns

1: $k \leftarrow \sqrt{n}$
2: **for** $i \leftarrow 1, Vol$ **do**
3: $\quad parent_{0i} \leftarrow 0$
4: **end for**
5: $parent_{00} \leftarrow -1$
6: $max \leftarrow 0$
7: **for** $i \leftarrow 1, n$ **do**
8: $\quad$ **if** $max > Vol - v_i$ **then**
9: $\quad\quad max \leftarrow Vol - v_i$
10: $\quad$ **end if**
11: $\quad$ **if** $i$ **mod** $k = 1$ **then**
12: $\quad\quad bigparent_{i \text{ div } k} \leftarrow parent_0$
13: $\quad$ **end if**
14: $\quad parent_{i \bmod k} \leftarrow parent_{(i-1) \bmod k}$
15: $\quad$ **for** $j \leftarrow 0, max$ **do**
16: $\quad\quad$ **if** $parent_{(i-1) \bmod k, j} \neq 0$ **then**
▷ we use the remark from KNAPSACK3($n$, $Vol$, $v$, $b$)
▷ algorithm to avoid using the Boolean matrix
17: $\quad\quad\quad parent_{i \bmod k, j+v_i} \leftarrow i$
18: $\quad\quad$ **end if**
19: $\quad$ **end for**

**Algorithm 10** continued

20: $max \leftarrow max + v_i$
21: **end for**

**Algorithm 11** BuildFrom(*line*, *k*, *Vol*, *v*, *parent*, *bigparent*)

▷ builds the *parent* matrix starting with the corresponding line stored in
▷ the *bigparent* matrix to the *line*$^{\text{th}}$ line
▷ input: $n$ – the number of the given items
▷ *Vol* – the total volume of the knapsack
▷ $v$ – the array of the given items' volumes
▷ output: *parent* – two-dimensional array,
▷ needed in the reconstruction of the solution

1: $parent_0 \leftarrow bigparent_{line \textbf{ div } k}$
2: **for** $i \leftarrow line \textbf{ div } k^2 + 1, line$ **do**
3: $parent_{i \textbf{ mod } k} \leftarrow parent_{(i-1) \textbf{ mod } k}$
▷ as a possible optimization, we could store for each line from bigparent
▷ the sum of the elements until that point, in order to avoid parsing the
▷ whole interval by introducing a variable with the meaning of max from
▷ the main algorithm
4: **for** $j \leftarrow 0, Vol - v_i$ **do**
5: **if** $parent_{(i-1) \textbf{ mod } k, j} \neq 0$ **then**
6: $parent_{i \textbf{ mod } k, j+v_i} \leftarrow i$
7: **end if**
8: **end for**
9: **end for**

**Algorithm 12** ReBuild($n$, $k$, *Vol*, $v$, *parent*, *bigparent*)

▷ input: $n$ – the number of the given items, $k$ the square root of $n$
▷ *Vol* – the total volume of the knapsack
▷ $v$ – the array of the given items' volumes
▷ *parent* – at the beginning: the array of last
▷ $(n \textbf{ mod } k) + 1$ lines of the solution

1: $i \leftarrow n$
2: $j \leftarrow Vol$
3: **while** $j > 0$ **do**
4: $aux \leftarrow parent_{i \textbf{ mod } k, j}$
5: **Write**: *aux*

**Algorithm 12** continued

```
6:     i ← aux − 1
7:     j ← j − v_aux
8:     BuildFrom(i, k, Vol, v, parent, bigparent)
9: end while
```

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 vectors will be constructed by using an implementation similar to the one utilized for the algorithm Knapsack3($n$, *Vol*, $v$, $b$), 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 $[n/2]$ ($j = 0, \dots, V$). For better understanding, we can think of the vector *goto*, as a "super-parent vector", similar to the vector *parent* in Knapsack6($n$, *Vol*, $v$, *parent*) algorithm, but with the modification, that for values of $i$ greater than $[n/2]$ it "points back" to line $[n/2]$ of the matrix. The construction of the *goto* vector is realized in the following way:

$$goto_j = \begin{cases} j & \text{for } i \le \left[\frac{n}{2}\right], \\ goto_{j-v_i} & \text{otherwise.} \end{cases}$$

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[3] 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: $T(n, V) = \Theta(n \cdot V) + T([n/2], goto_V) + T(n - [n/2], V - goto_V)$, where we obtain $T(n, V) = \Theta(n \cdot V)$.

[3] We consider that both the students and the reader are familiarized with the *Divide and Conquer* method [1].

The proof of this is based on the usual methods, the most comfortable being the *substitution method* or the applying of the *Master Theorem*. This step we will let for the reader to think upon ([1]).

## 6. The one-dimensional task of the knapsack

Once we analyzed this simplified variant of the task, we can present the more complex variants. In presenting the solving methods we will focus on the particularities only, because most of the solution is similar to those presented above. The text of the one-dimensional task of the knapsack was presented in the beginning of this article.

In order to solve the task we must modify the solution from the coins task in the following way: for each obtained sum $j$ we have the optimum cost $opt_j$ $(j = 0, 1, \ldots, V)$ with whose help the respective sum was obtained. Obviously, we will overwrite the $opt_j$ $(j = 0, 1, 2, \ldots, V)$ value, if we found a better value for the $j$ sum at the current step. So, we have $opt_{ij} = \mathbf{optim}(opt_{i-1,j}, opt_{i-1,j-v_i} + c_i)$, where **optim** is the optimal function specific to the task (min or max). The final solution will be given by $opt_{nV}$, for the cases (min, =) and (max, =), respectively by $\mathbf{optim}(opt_{nj} \mid j = 0, 1, 2, \ldots, V)$ for the other cases.

We notice that in the unbounded task's case we can apply a new improvement: we can eliminate the "dominant" items, that certainly do not belong to the optimal solution. For example, in the (max, $\leq$) case if an item has a lower cost and a bigger weight than another item, the first item can be eliminated.

**Task 3 – Fish**[4]

Fisherman Naum has caught $n$ ($n \leq 500000$) fish. The weight $v_i$ of each fish ($v_i \leq 35000$) is measured at the coast. Naum works for one fishing company and he has the right to take home at most $V$ ($V \leq 7000$) grams fish. He wishes to take as few as possible number of fish, but with the maximal possible weight $L$ ($\leq V$). Decide which fish will be taken home and give the number and the weight of fish to be taken home.

*Example*: $n = 10, V = 280, v = (300, 10, 30, 80, 200, 20, 20, 60, 10, 100)$. Number of fishes: 2, the weight of the fishes to be taken home: $(200, 80)$.

[4] BOI 2000, Ohrid, Macedonia (the author of the task is unknown for us)

*Solution*

The first observation we make is, that the weight of some fishes can be greater than $V$, so we can safely ignore these, because it is sure, that they will be not part of the solution. The solution presented in Pascal shows a possible implementation, of the approach explained at the end of the section 5.3. The time limit for this problem allowed solutions with $\Theta(n \cdot V)$ complexity to run in time.

```
type fishes = array[1..500000] of Word;          { again, for simplicity }
             { we suppose that the compiler lets us allocate this vector }
     vect   = array[0..7000] of Longint;

var optNew, optOld, gotoOld, gotoNew: vect;
    v:      fishes;                          { the weights of the fishes }
    n, Vol: Integer;

procedure DivideAndConquer(low, high: Longint; Vol: Word);
  var i, j: Longint;
      aux:  Word;
  begin
    if vol = 0 then
      Exit;
    if low <> high then begin
      FillChar(gotoOld, sizeof(gotoOld), 0);
      FillChar(gotoNew, sizeof(gotoNew), 0);
      FillChar(optOld, sizeof(optOld), 0);
      optOld[0] := 1;              { using -1 would not have worked here }
                 { do not forget to subtract one from the final solution }
                                     { when writing the number of items }
      optNew := optOld;
      for i := low to high do begin
        for j := v[i] to vol do
          if (optOld[j-v[i]] <> 0) and ((optNew[j] = 0) or
                              (optNew[j] > optOld[j-v[i]] + 1)) then begin
            optNew[j] := optOld[j-v[i]] + 1;
            if i <= (low+high) div 2 then
              gotoNew[j] := j
            else
              gotoNew[j] := gotoOld[j-v[i]]
          end;
        gotoOld := gotoNew;
        optOld := optNew
      end;
      i := vol;
      while (i > 0) and (optOld[i] = 0) do
```

```
      Dec(i);
    if (low = 1) and (high = n) then
      WriteLn(optOld[i]-1);                { the number of the fishes }
    aux := gotoOld[i];
            { we need to store this in an auxiliary variable, because }
       { the gotoOld vector will change after the first recursive call }
    DivideAndConquer(low, (low+high) div 2, aux);
    DivideAndConquer((low+high) div 2 + 1, high, vol-aux);
  end else
    WriteLn(v[low])                { low = high, we have one item left }
end;
```

## 7. The two-dimensional task of the knapsack

*A thief rubs a house. In the house there are $n$ items, while the $i^{th}$ item has the $v_i$, $(i = 1, 2, \ldots, n)$ volume, the $g_i$ weight and the $c_i$ cost. We know that the thief cannot carry a weight bigger than $G$ and that his knapsack has a $V$ volume. Determine the items' configuration that have room in the knapsack and that can be carried by the thief and for which the costs' sum is maximal.*

We changed the last word from "optimal" to maximal, in order to avoid the discussion of more similar cases. The difference from the precedent task is given by the introduction of the weight factor, and thus the number of task's dimension is increased. Unfortunately, this involves the increase of the complexity of the solution which this time will be $\Theta(n \cdot V \cdot G)$. So we can deduce that we will need a three-dimensional matrix in order to build the solution. The formula can be easily deduced, in analogy with the solution from above:

$$opt_{ijk} = \max(opt_{i-1,j,k}, opt_{i-1,j-v_i,k-g_i} + c_i).$$

## 8. The multidimensional task of the knapsack

We will present the general variant of the $m$-dimensional task of the knapsack, where the meaning of $n$ stays the same.

Let us consider a matrix $R$ of dimensions$n \cdot (m + 1)$ and a $Q$ vector of $m$ elements.

Determine the $X$ vector so that

$$\sum_{i=1}^{n} X_i R_{ij} \leq (\text{or} =) Q_j, \quad j = 1, 2, \ldots, m$$

and the value of

$$\sum_{i=1}^{n} X_i R_{i,m+1}$$

is optimal. The $X_i$ values depend on the task's variant. For example, we can have:

- in the case of the binary variant $0 \leq X_i \leq 1$.
- in the case of the bounded variant $l_i \leq X_i \leq u_i$.
- in the case of the discrete variant $X_i \in \mathbb{N}$, and in the case of the fractional variant $X_i \in Q$.

*Remark.* We can notice the analogy between this general task, and the ones presented before: the first $m$ columns of the matrix $R$ contain the dimensions (e.g. volumes, weights etc.) of the $n$ items, and the last $(m+1)^{\text{th}}$ column the cost of each item. The $Q$ vector represents the dimensions of the knapsack.

In the general case the task is NP-complete, but we can improve a lot the banal *backtracking*.

**Task 4 – Collection**

Alan has a lot of CD-s and DVD-s. He wants to label them with numbers, so he went to the store to buy labels formed by digits. In the store there are $n$ $(1 \leq n \leq 30)$ boxes, which are containing one-digit labels. One box contains some 0-digit labels, some 1-digit labels etc. (different packages may contain different number of each kind of labels). He wants to buy some of these boxes of labels in order to form the numbers from 1 to $k$ $(1 \leq k \leq 1000000)$ ($k$ is the number of his CD-s and DVD-s).

(1) Determine, if it is possible to buy some boxes, knowing that Alan wants to use all of the bought labels (obviously, it is not sure that this is possible).

(2) If there are such boxes, determine the minimal number of them!

(3) Determine which boxes has to buy Alan!

*Example:* $n = 4$, $k = 11$

In the next table $l$ means "label".

| | $l=0$ | $l=1$ | $l=2$ | $l=3$ | $l=4$ | $l=5$ | $l=6$ | $l=7$ | $l=8$ | $l=9$ |
|---|---|---|---|---|---|---|---|---|---|---|
| box 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| box 2 | 1 | 2 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| box 3 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| box 4 | 0 | 2 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

The answers:

(1) Yes, it is possible.

(2) There are 2 boxes which have to be bought by Alan.

(3) That are: the second and the fourth one. (Extra details: the labels will be: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11). Alan will have 1 digit 0, 4 digits 1, 1 digit 2, 3, 4, 5, 6, 7, 8 and 9.)

*Solution*

The task can be cut in two subtasks. In the first one we have to determine the vector of the numbers of the digits $i$ ($i = 0, 1, \ldots, 9$), which appear in the numbers between 1 to $k$. Let us note this vector (having 10 elements) with $A$. We can find these numbers in $\Theta(\log k)$ time, using the dynamic programming method.

Let us note the vector of boxes with $B$. The second subtask is to obtain a subset of $B$, where $\sum\limits_{i\in I} B_{ij} = A_j$, $j = 0, 1, \ldots, 9$ and $I = \{$ *the indexes of the boxes in the corresponding subset* $\}$.

We cut $B$ in two sets, $S_1$ containing $B_i$, $i = 1, 2, \ldots, [n/2]$, and $S_2 = B \setminus S_1$, with $n - [n/2]$ elements. We determine all the subsets of $S_1$, and calculate their vectorial sums. After that, we sort these subsets according to their sums. In the next step we calculate the vectorial sum of each subset of $S_2$. Let us note with $V$ a sum of this kind. We want to add to $V$ some elements from $S_1$ in order to obtain $A$. We use the binary search algorithm for $A - V$ in the stored subsets of $S_1$.

Using a sorting algorithm with $\Theta(n \log n)$ complexity, the sorting time will be $\Theta(2^{[n/2]} \cdot \log(2^{[n/2]})) = \Theta(2^{[n/2]} \cdot [n/2])$. The searching time will be $\Theta(2^{[n/2]} \cdot \log(2^{[n/2]})) = \Theta(2^{[n/2]} \cdot [n/2])$ that yields to $\Theta(2^{[n/2]} \cdot [n/2] + \log k)$ time complexity for the whole solution. To store the input data we need $\Theta(n)$ memory and the $A$ vector takes $\Theta(1)$ memory. After that we need to store only the subsets of $S_1$, which yields to memory complexity $\Theta(n) + \Theta(1) + \Theta(2^{[n/2]} \cdot n) = \Theta(n + 2^{[n/2]} \cdot n)$.

Let us see the corresponding Pascal program:

```
const mSubsets = 35000;

type vect    = array[0..9] of Longint;
     Tset    = array[1..30] of Boolean;
     Tsubset = record
                 v: vect;
                 n: Byte;
                 selected: Tset
               end;

var b: array[1..30] of vect;                  { the content of the boxes }
    n: Byte;
    k: Longint;
    a: vect;        { the number of appearence of each digit from 1 to k }
    subsets:   array[1..mSubsets] of Tsubset;
           { again, we consider, that the compiler lets us allocate this }
    nSubsets:  Word;
    ss:        Tset;              { needed by the backtracking algorithm }
    solution:  Tset;
    nSolution: Shortint;

procedure BuildA(x, pow10, last: Longint);
                              { builds the A vector, runs in log(k) time }
  var q, digit: Shortint;
  begin
    digit := x mod 10;
    if digit <> 0 then
      Inc(a[0], (x div 10)*pow10);
    for q := 1 to digit-1 do
      Inc(a[q], (x div 10+1)*pow10);
    Inc(a[digit], (x div 10)*pow10+last+1);
    for q := digit+1 to 9 do
      Inc(a[q], (x div 10)*pow10);
    if x >= 10 then
      BuildA(x div 10, pow10*10, digit*pow10+last)
  end;

procedure BuildSubsets(sp: Byte);
            { builds all the possible subsets from the first [n/2] boxes }
  var q,w: Byte;
  begin
    if sp > n div 2then begin
      Inc(nSubsets);
      FillChar(subsets[nSubsets], sizeof(subsets[nSubsets]), 0);
      subsets[nSubsets].selected := ss;
```

```
      for q := 1 to n div 2 do
        if ss[q] then begin
          Inc(subsets[nSubsets].n);
          for w := 0 to 9 do
            Inc(subsets[nSubsets].v[w], b[q][w])
        end;
    end else begin
      ss[sp] := false;
      BuildSubsets(sp+1);
      ss[sp] := true;
      BuildSubsets(sp+1)
    end;
end;

function less(var x, y:vect): Boolean;           { Operations on vectors }
  var q: Byte;
  begin
    q := 0;
    while (q < 9) and (x[q] = y[q]) do
      Inc(q);
    less := x[q]<y[q]
  end;

function Equal(var x, y: vect): Boolean;
  var q: Byte;
  begin
    q := 0;
    while (q < 9) and (x[q] = y[q]) do
      Inc(q);
    equal := x[q]=y[q]
  end;

procedure SortSubsets(down, up: Byte);
  var i, j: Byte;
      m, c: Tsubset;
  begin
    m := subsets[(down+up) div 2];
    i := down;
    j := up;
    repeat
      while less(subsets[i].v, m.v) do
        Inc(i);
      while less(m.v, subsets[j].v) do
        Dec(j);
      if i <= j then begin
```

```
        c := subsets[i];
        subsets[i] := subsets[j];
        subsets[j] := c;
        Inc(i);
        Dec(j)
      end;
    until i > j;
    if down < j then
      SortSubsets(down, j);
    if i < up then
      SortSubsets(i, up)
  end;

function BinarySearch(var dif: vect): Longint;
                                  { Returns -1 if dif could not be found }
  var down, up, m: Word;
  begin
    down := 1;
    up := nSubsets;
    while down < up do begin
      m := (down+up) div 2;
      if Equal(subsets[m].v, dif) then begin
        BinarySearch := m;
        Exit
      end else
        if less(dif, subsets[m].v) then
          up := m-1
        else
          down := m+1
    end;
    BinarySearch := -1;
  end;

procedure FindSolution(sp: Byte);
               { generates all the possible subsets using the boxes from }
                    { [n/2] + 1 to n, and binary searches for a solution }
  var sum, difference: vect;
      nSum, q, w:      Byte;
      aux:             Longint;
  begin
    if sp > n then begin
      Fillchar(sum, sizeof(sum), 0);
      nSum := 0;
      for q := n div 2 + 1 to n do
        if ss[q] then begin
```

```
          Inc(nSum);
          for w := 0 to 9 do Inc(sum[w], b[q][w])
        end;
      difference := a;
      for q := 0 to 9 do
      Dec(difference[q], sum[q]);
      aux := BinarySearch(difference);
      if (aux <> -1) and
       ((nSolution > nSum+subsets[aux].n) or (nSolution = -1)) then begin
                   { we found a better solution or this is the first one }
        nSolution := nSum + subsets[aux].n;
        Fillchar(solution, sizeof(solution), false);
        for q := 1 to n div 2 do
          solution[q] := subsets[aux].selected[q];
        for q := n div 2 + 1 to n do
          solution[q] := ss[q]
      end
    end else begin
      ss[sp] := false;
      FindSolution(sp+1);
      ss[sp] := true;
      FindSolution(sp+1)
    end
  end;

Begin
  { read data }
  FillChar(a, sizeof(a), 0);
  BuildA(k, 1, 0);
  nSubsets := 0;
  BuildSubsets(1);
  SortSubsets(1, nSubsets);
  nSolution := -1;
  FindSolution(n div 2 + 1);
  { write data }
End.
```

## 9. Special task

**Task 5 – Sticks 2**

Let us remember the task **Sticks 1** presented above. But now we have to determine the number of possibilities of the grouping of the sticks in two "lines", so that the difference between this two "lines" length to be minimal and to determine

also, the value of this minimum. We have some changes related to the restrictions: there are $n$ $(2 \leq n \leq 32)$ sticks, and their lengths are between 1 and 1000000.

*Example*: $n = 4$, $length = (1, 2, 4, 6)$. The minimal difference is 1 and there are 2 such groupings, that are: $(1 + 2 + 4 = 7)$ and 6, $(2 + 4 = 6)$ and $(1 + 6 = 7)$.

*Solution*

We will consider all the possible subsets which can be formed by the first $[n/2]$ sticks, and we will sort them corresponding to the sum of their total lengths. After that we will determine all the possible subsets which can be formed by the last $n - [n/2]$ sticks. For all the subsets $A$, obtained in such a way, we will search a subset $B$ (using binary search) in the first list of the subsets which has its value closest to $S/2 - x$, where $S$ is the total length of all the given sticks and $x$ is the total length of the sticks from subset $A$. The "candidate" group of the sticks will be obtained by uniting set $A$ and $B$. If the length of the line is closer to $S/2$ than the solution obtained before now, that means we have a new minimal difference, and (for the moment) only one way to reach it. If the difference is equal to the actual one, we will increment the number of the possibilities. At the end, we will write the value of the minimal difference and the number of the groupings which lead to it.

Let us analyse the complexity of this algorithm. For each subset $A$ (there are approximately $2^{[n/2]}$ such subsets) we make a binary search on the first list (which has $2^{[n/2]}$ elements). This yields to complexity $\Theta(2^{[n/2]} \cdot \log_2 2^{[n/2]}) = \Theta(2^{[n/2]} \cdot [n/2])$, which is substantially better than the naive $\Theta(2^n)$ backtracking.

## 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* **15**, no. 6, Agora Media, Tg. Mureş (2006), 31–35 (in Romanian).

[4] C. Silvestru-Negruşeri, Optimization of the Programs Using Bit Operations, *GInfo* **14**, no. 5, Agora Media, Tg. Mureş (2004), 32–36 (in Romanian).

[5] `http://www.tutor.ms.unimelb.edu.au/knapsack_new/`

[6] `http://www.ifors.ms.unimelb.edu.au/tutorial/knapsack/`

[7] `http://www.ace.tuiasi.ro/ro/academice/curricula/programe/ingineri/pa_209_pa1_craus.pdf`

CSABA PĂTCAŞ and KLÁRA IONESCU
BABEŞ-BOLYAI UNIVERSITY
CLUJ-NAPOCA ROMANIA

*E-mail:* patcas.csaba@gmail.com

*E-mail:* clara@cs.ubbcluj.ro

*(Received May, 2008)*