Technische Universiteit
Eindhoven
University of Technology

DBL Algorithms (2IO90)
Problem Description

Rectangle Packing

Mark de Berg

1 Description of the problem

In a packing problem we are given a collection of objects that we wish to pack into some kind of container. An application of such a problem is when we want to place a set of boxes into a moving van, or a set of suitcases into a car trunk. There are also applications that give rise to 2-dimensional packing problems. For example, suppose we have a large piece of leather from which we want to cut out certain shapes to make leather bags. Then finding a good layout for the shapes on the given piece of leather can be viewed as a packing problem.

In this DBL project we consider two variants of a 2-dimensional packing problem where the goal is to place a set S of n rectangles of various sizes (with edges parallel to the x- and y-axis) into a minimum-area rectangular container (with edges parallel to the x- and y-axis). In such a placement, the rectangles should not overlap with each other, that is, they should have disjoint interiors. The difference between the two variants is the type of containers we can use.

- In the first variant we are allowed any type of rectangle as container. The problem can be then stated as follows: place the rectangles such that their *bounding box* has minimum area. (The bounding box of a set of objects is the smallest axis-aligned rectangle containing all the objects.)
- In the second variant the height of the container, H, is specified. In other words, we want to place the rectangles into a rectangle of height H whose width is minimized.

For both variants, we will consider two versions: one where we are allowed to rotate the input rectangles by 90 degrees, and one where rotations are not allowed. See Fig. 1 for an example.

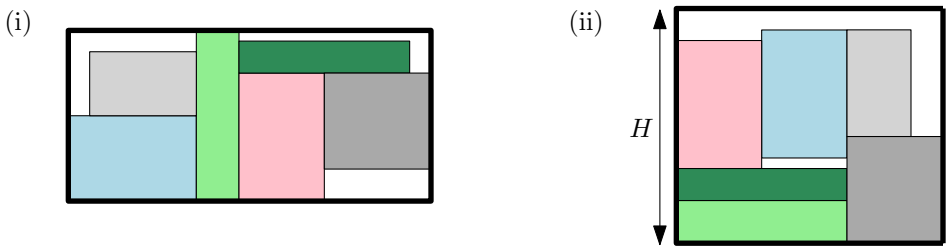


Figure 1: (i) A packing of a set of rectangles into a rectangular container. (ii) A packing of the same set of rectangles into a strip of height $H = 22$; some rectangles have been rotated.

2 The project

In this project you have to develop software for the variants of packing problem discussed above. Your program should be implemented in Java. It should read an input file that describes a problem instance from standard input (System.in) and write the output to standard output (System.out). You should submit your program using Canvas.

The program will be evaluated by running it on a set of test instances. For each of the four problem settings (arbitrary container/fixed-height container, rotations/no rotations) there will be instances with the following values for n, the number of rectangles:

$$n = 3, 5, 10, 25, 10^4.$$

For each of these values of n there will be three different instances for every problem setting. Thus the total number of test instances is $4 \times 5 \times 3 = 60$. Your program should terminate within five minutes on any of the problem instances, otherwise the output for that instance is not taken into account.

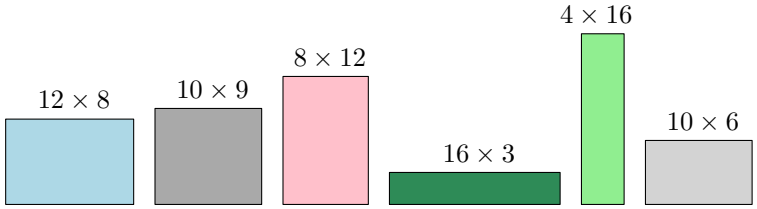
The most important criterion for grading the software will be the quality of the output, that is, the area of the container of your packing (assuming your algorithm computes a valid solution within five minutes). The running time is another, less important criterion; it will influence the grade by at most 0.5 points.

2.1 Input format

The input file describing a problem instance has the following structure; see also Fig. 2.

`container height:` *variant* $[H]$
`rotations allowed:` *version*
`number of rectangles:` n
w_1 h_1
$\vdots$
w_n h_n

Here the parameter *variant* $\in \{$`free`, `fixed`$\}$ in the first line determines the variant that needs to be solved, and H is a positive integer that indicates the height of the container in case *variant* = `fixed`. (The square brackets indicate that the parameter H is optional: H is not specified when *variant* = `free`.) The parameter *version* $\in \{$`yes`, `no`$\}$ in the second line specifies whether it is allowed to rotate the input rectangles, and the parameter n in the third line specifies the number of rectangles in the input. The rest of the input file consists of n



(i)
```
container height:  free
rotations allowed:  no
number of rectangles:  6
12 8
10 9
8 12
16 3
4 16
10 6
```

(ii)
```
container height:  fixed 22
rotations allowed:  yes
number of rectangles:  6
12 8
10 9
8 12
16 3
4 16
10 6
```

Figure 2: (i) Input file corresponding to the shown set of rectangles when the container height is free and rotations are not allowed. (ii) Input file corresponding to the same set of rectangles when the container height is fixed at $H = 22$ and rotations are allowed.

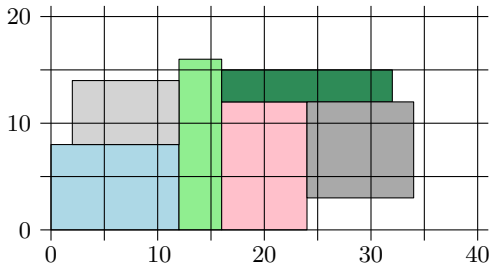
lines, where the i-th line specifies the width w_i and height h_i of the i-th rectangle. The width and height are positive integers in the range $1, \ldots, 10^4$. When *variant* = `fixed`, the height of the rectangles will never exceed H so that there is always a valid packing.

2.2 Output format

The output file has the following structure; see also Fig. 3

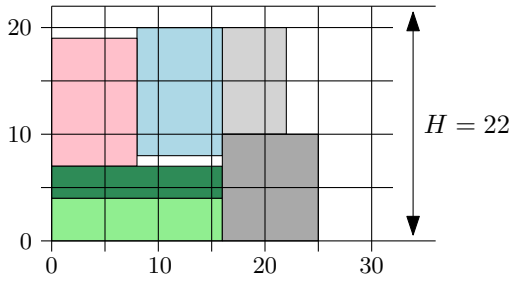
`container height:` *variant* $[H]$
`rotations allowed:` *version*
`number of rectangles:` n
w_1 h_1
⋮
w_n h_n
`placement of rectangles`
$[rotated_1]$ x_1 y_1
⋮
$[rotated_n]$ x_n y_n

(i)




```
container height:  free
rotations allowed:  no
number of rectangles:  6
12 8
10 9
8 12
16 3
4 16
10 6
placement of rectangles
0 0
24 3
16 0
16 12
7 0
2 8
```

(ii)




```
container height:  fixed 22
rotations allowed:  yes
number of rectangles:  6
12 8
10 9
8 12
16 3
4 16
10 6
placement of rectangles
yes 8 8
yes 11 0
no 0 7
no 0 4
yes 0 0
yes 11 10
```

Figure 3: (i) A possible solution for the input in Fig. 2(i) and the corresponding output file. (ii) A possible solution for the input in Fig. 2(ii) and the corresponding output file.

The first part is simply a copy of the input file. After that there should be a line with the text `placement of rectangles`, followed by a specification of the placement of each of the n rectangles. More precisely, the i-th line of the placement specification has the form

$$[rotated_i]\ x_i\ y_i$$

where

- $rotated_i \in \{\texttt{yes},\texttt{no}\}$ is a parameter that specifies, in case rotations are allowed, whether the i-th rectangle has been rotated;
- x_i and y_i are non-negative integers indicating the x- and y-coordinate of the lower-left corner of the placement of the i-the rectangle.

Note that for rotated rectangles, x_i and y_i specify the lower-left corner *after* rotation. Thus the line

$$\texttt{no}\ x_i\ y_j$$

means that the i-th rectangle will be occupy the region $[x_i, x_i + w_i] \times [y_i, y_i + h_i]$, while the line

$$\texttt{yes}\ x_i\ y_j$$

means that the i-th rectangle will be occupy the region $[x_i, x_i + h_i] \times [y_i, y_i + w_i]$. Also note that the order in which you specify the placement of each rectangle in the second part of the output file should correspond to the order in which the widths and heights of the rectangles are specified in the first part (which should be the same as in the input file).

The lower-left corner of the container in your packing should always be the point $(0, 0)$. This means that your output should be such that $\min_{1 \leqslant i \leqslant n} x_i = \min_{1 \leqslant i \leqslant n} y_i = 0$.

3 Hints for solving the problem

Modelling and solving the problem. When faced with an algorithmic problem the first step is often to formulate it in precise mathematical terms: what is the input to the problem, what are the requirements on the output—in other words, what is a valid solution—and what determines the quality of a solution? In the packing problem you are given for this DBL-project this is already given. Thus you can immediately start thinking about algorithms to solve the problem.

It is allowed to use algorithms from the literature, provided you properly cite the relevant papers describing the algorithms, but you can also design your own algorithm. In case an algorithm from the literature does not solve the exact same problem as you need to solve, you can perhaps adapt the algorithm. Note that algorithms from the literature are sometimes hard to implement, and that they not always give high-quality solutions in practice. Hence, make a careful choice about which algorithm(s) you are going to implement.

In this DBL-project there are two different variants of the problem, each with two versions. It may be a good idea to see if you can use an algorithm for one variant as a subroutine in the algorithm for another variant. Remember that the number of rectangles, n, in the input instances used in the evaluation varies a lot: there will be instances that are quite small ($n = 3$ and $n = 5$) and instances that are quite large ($n = 10,000$). It is advisable to develop separate

algorithms for these cases: an algorithm that is guaranteed to compute an optimal solution when n is small (but is too slow to work on large problem instances), and an algorithm that can still compute a fairly good (but probably non-optimal) solution for large values of n.

Implementing algorithms. Before you start implementing your algorithms, it is good to make a design in which you establish which classes and methods need to be implemented, and what functionality they should offer. This makes it easier to divide implementation tasks among the team members. It is wise to first implement a simple, possibly less efficient, version of your algorithm. Once this version works properly, you can start replacing certain modules by ones that are more efficient or compute better solutions. It is not allowed to use other people's code (not from within the course and not from outside the course) without prior permission. This also applies to libraries that are not included in Java.

Evaluating the algorithms. Evaluating your algorithms should be done theoretically and experimentally. A theoretical analysis involves analyzing the running time, proving that the algorithm correctly solves the problem and/or proving certain properties of the solutions is computes. An experimental evaluation involves investigating the quality of the computed solutions, the actual running time in practice, etcetera. For the experimental evaluation, which is one of the most important aspects of this DBL project, you should carefully generate different types of data sets (different aspect ratios of the input rectangles, rectangles that differ a lot in size or have similar sizes, and so on). It is useful to generate (some of) the data sets in such a way that you know the area of an optimal packing. Often the evaluation of your algorithm leads to ideas to improve the algorithms. For this a GUI that allows you to easily run the algorithm on different test sets and see the resulting output is quite useful. A tool that generates a picture of the output is also useful for generating figures to be used in the final report.