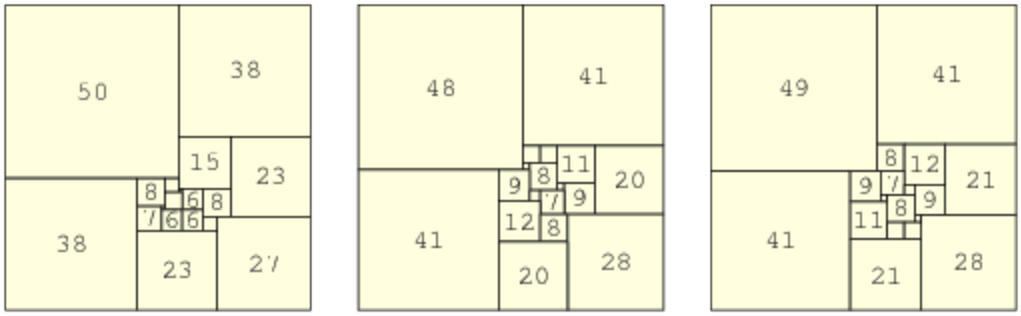
MSU CSC 325 Spring 2016
Mar. 4, 2016
Due: Wed. Mar. 23, 11:59pm

This problem is to place square tiles into a rectangular space. This problem has many real-world applications such as finding the most efficient way to pack shipping boxes into trucks, etc.

Further reading, variations and similar problems:
http://mathworld.wolfram.com/MrsPerkinssQuilt.html
http://www.squaring.net/sq/tws.html
http://math.stackexchange.com/questions/tagged/packing-problem
https://en.wikipedia.org/wiki/Packing_problems
http://mathworld.wolfram.com/SquareTiling.html

Diagram A. These are some "ideal" placements of squares (but these examples are more efficient than you will produce for this problem):



In the three examples shown, the overall shape is a square and the squares within are labelled with their edge lengths.
Source: http://mathworld.wolfram.com/MrsPerkinssQuilt.html

SUMMARY OF PROBLEM TASK: Given an overall rectangle and a set of square tiles, place the tiles into the overall rectangle using a particular algorithm described here.

Development hints: *It is not required but probably helpful during development to* ***graphically display*** *the squares as the algorithm progresses. A Python program is provided that can draw graphs similar to those above. The graphing program is described below in Appendix B.*
The algorithm to be used in this problem is a simplifying (and inefficient) technique to place tiles into the given space.

The recursive algorithm to use:
Given a bounding rectangle and a set of square tiles.

- If there are no tiles that fit within the bounding rectangle, there is no further action.
- If there at least one tile that fits within the bounding rectangle, ***choose the largest*** square tile that fits and place a corner of the tile ***at the lower left*** corner of the space. Delete that tile from the set of available tiles. There are now three rectangular areas of the bounding rectangle which do not yet have tiles. *(Perhaps these three areas have zero size and no tile may be placed in them. See examples in Diagram B).*

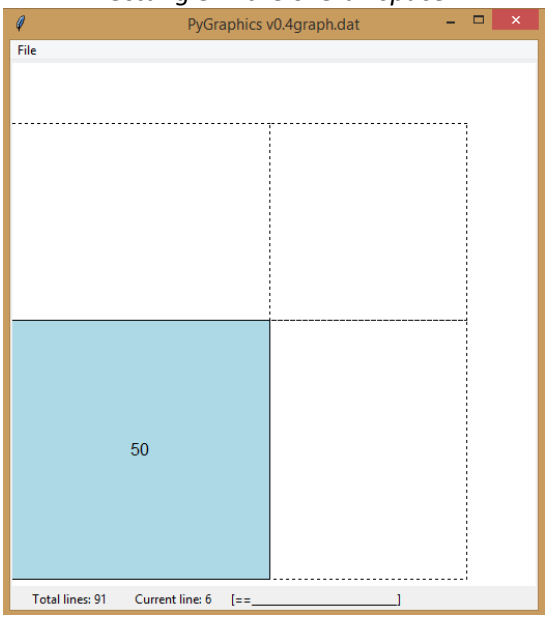
- ***Recursively and in a specific order,*** attempt to place tiles within each of the three rectangular areas of the bounding rectangle which do not yet have tiles. The order that those three rectangular areas should be attempted is illustrated by examples in Diagram C.
    - First, use the "upper left" rectangular portion of the bounding rectangle which does not yet have tiles.
    - Secondly, use the "upper right" rectangular portion of the bounding rectangle which does not yet have tiles.
    - Thirdly (lastly), use the "lower right" rectangular portion of the bounding rectangle which does not yet have tiles.

This algorithm, that you are required to use, is inefficient because square tiles are only permitted to be placed completely within each of the rectangular sub-areas. (Compare the example output diagram of Appendix to the ideal solution in Diagram A.) Improvements to the algorithm are a research topic!

Diagram B. Placement of a square within a rectangle.

Case 1. There are three non-zero-area rectangles. *Example: after placement of square of size 50 into the bounding rectangle (0, 0) --- (0, 88) --- (88, 88) --- (88, 0), there are three non-zero-area rectangles in the overall space.*

Case 2. There is only one non-zero-area rectangle. *Example: after placement of square 38 into the bounding rectangle (0, 50) --- (0, 88) --- (50, 88) --- (50, 0), there is only one non-zero-area rectangle in the overall space.*



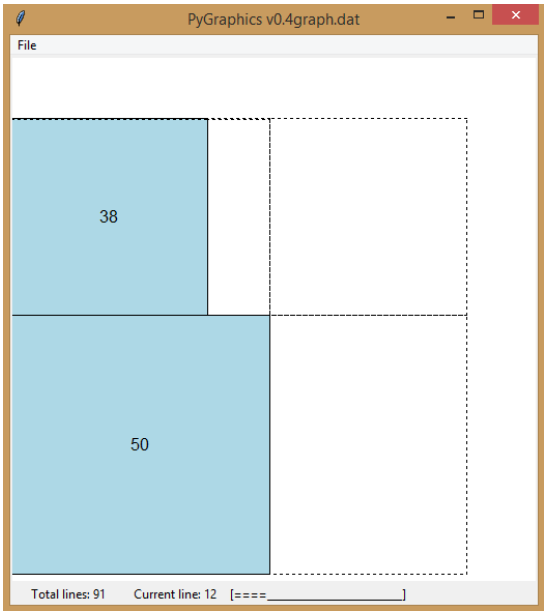
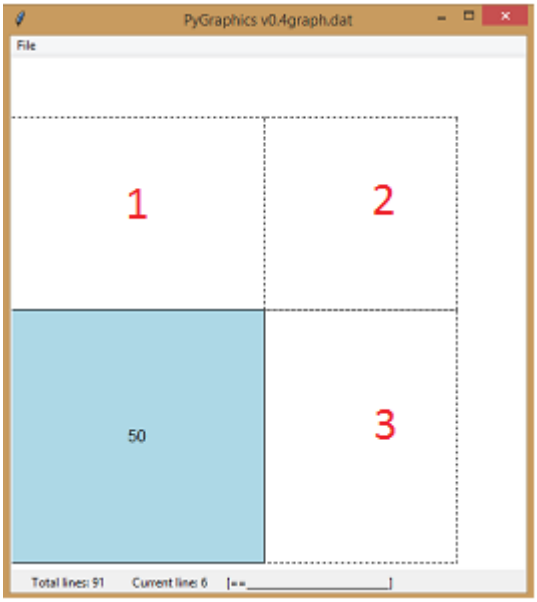


Diagram C. The required order of recursive calls.
Placement of a square into a bounding rectangle defines three rectangular portions of the bounding rectangle. Those three areas are shown in this diagram and may have zero size. Your program must attempt to place tiles into the bounding rectangle in the recursive order as described in the algorithm above and numbered below.



Input to your program: A data file is used to specify the sizes of the overall bounding rectangle and the squares to be placed within it. The data file contains only a series of integer values in the following form:

```
Width of the overall bounding rectangle
Height of the overall bounding rectangle
Number, N, of squares available for placement within the overall
   bounding rectangle
Edge length of square 1
Edge length of square 2
. . . .
Edge length of square N
```

When there are N squares available for placement within the overall bounding rectangle, the input data file contains exactly N+3 integers. The input file sequence of squares is NOT necessarily in decreasing order of size. It is your responsibility to select squares in size order for use within the algorithm. *A sample input data file, containing the data of the example used in this problem description, is provided.*

Output of your program: text output, in the order that the tiles were placed, of the identifying keyword "OUTPUT," the square's size and lower left coordinate. Shown below is the expected form of the output of the example used in this problem description. *White-space formatting is not critical. The identifying keyword "OUTPUT" should not appear in any other printed output lines.*

```
OUTPUT Square with edge 50 has lower left (0, 0)
OUTPUT Square with edge 38 has lower left (0, 50)
OUTPUT Square with edge 8 has lower left (38, 50)
OUTPUT Square with edge 8 has lower left (38, 58)
OUTPUT Square with edge 7 has lower left (38, 66)
OUTPUT Square with edge 6 has lower left (38, 73)
OUTPUT Square with edge 6 has lower left (38, 79)
OUTPUT Square with edge 1 has lower left (38, 85)
```

```
OUTPUT Square with edge 4 has lower left (46, 58)
OUTPUT Square with edge 4 has lower left (46, 62)
OUTPUT Square with edge 38 has lower left (50, 50)
OUTPUT Square with edge 27 has lower left (50, 0)
OUTPUT Square with edge 23 has lower left (50, 27)
OUTPUT Square with edge 6 has lower left (77, 27)
OUTPUT Square with edge 5 has lower left (77, 33)
```

Notes on output:

- Creating these output statements in the requested order is done by printing inside the recursive routine. No other ordering mechanism should be needed.
- The identifying keyword "OUTPUT" will help to identify any genuine output from other print statements. Please use that keyword on no other printed output statement. In fact, we would appreciate a minimum of print statements in your submitted code.

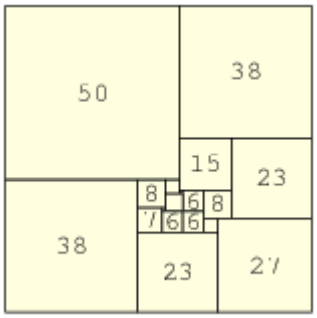
Program rules and guidelines (see also submission rules below):

1. The bounding area is a rectangle, and each edge length is a non-negative integer.
2. All tiles are squares, and each edge length is a non-negative integer.
3. No rotation of a tile to sit at an "angle" is allowed.
   That would be a variation of this problem, http://www2.stetson.edu/~efriedma/squinsqu/
4. Your program must attempt to place tiles into the bounding rectangle in the recursive order as described above.
5. Your program must read from an input file the sizes of the overall bounding rectangle and the squares to be placed within it. The form of the input file is described above.
6. Your program must create output which is a list of each placed square and the lower left coordinate at which that square was placed, with the identifying keyword **`OUTPUT`**. Example output is shown above.

Important: *Of course you may use print statements to debug your code. However,* ***please do not turn in code that causes massive amounts of output to the screen.***

Appendix A. Example step-by-step operation of the algorithm upon the example of Diagram A.

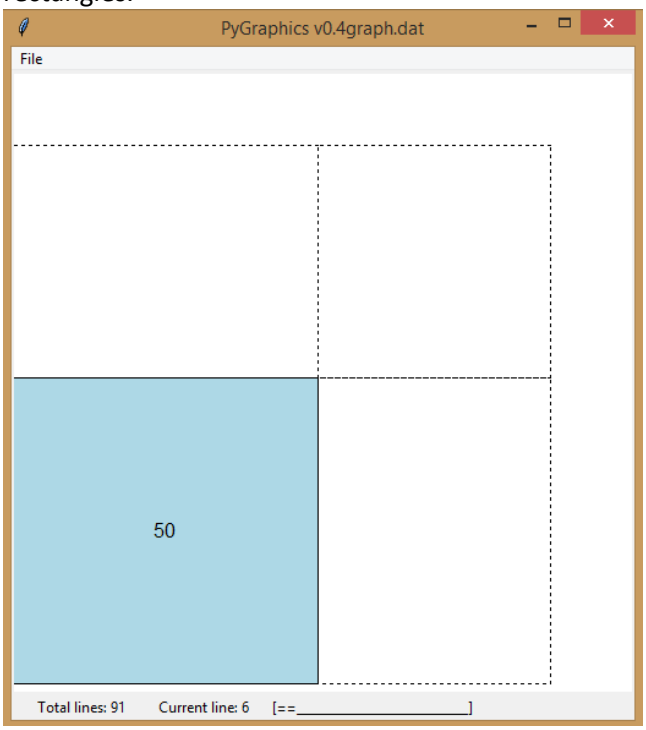
(Note: The placement of squares by your implementation will be different and less efficient than that of Diagram A. This example simply uses the same-sized squares as were used in the example of Diagram A.)



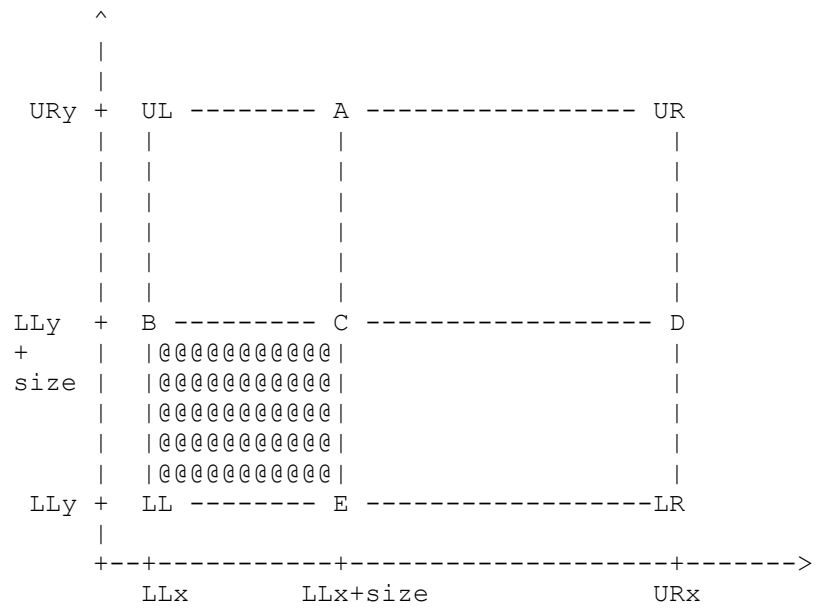
Initial example – your output results will be different!

The initial bounding area is a square whose edge lengths are 89 units and whose coordinates are lower left corner at (0,0) and upper right corner at (88,88). Seventeen squares have been placed in this bounding area, whose sorted edge lengths are 50, 38, 38, 27, 23, 23, 15, 8, 8, 7, 6, 6, 6, 5, 4, 4, and 1 respectively.

The initial bounding area is a square whose edge lengths are 89 units. Within that area, your program can place the square tile of size 50, and by the provided rules, must place that square in the lower left corner of the bounding area. The portion of the bounding area *outside* that size-50 square is in three rectangles.

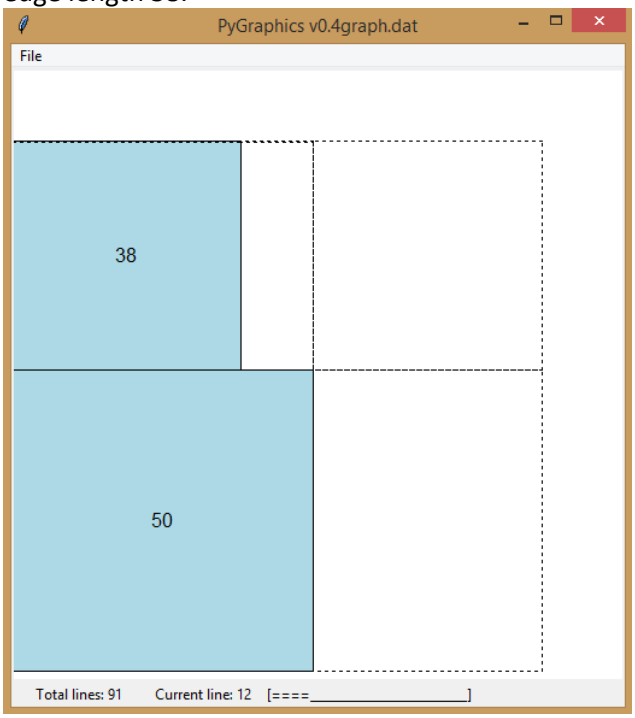


In general, the coordinates of the portions of the space are illustrated below.

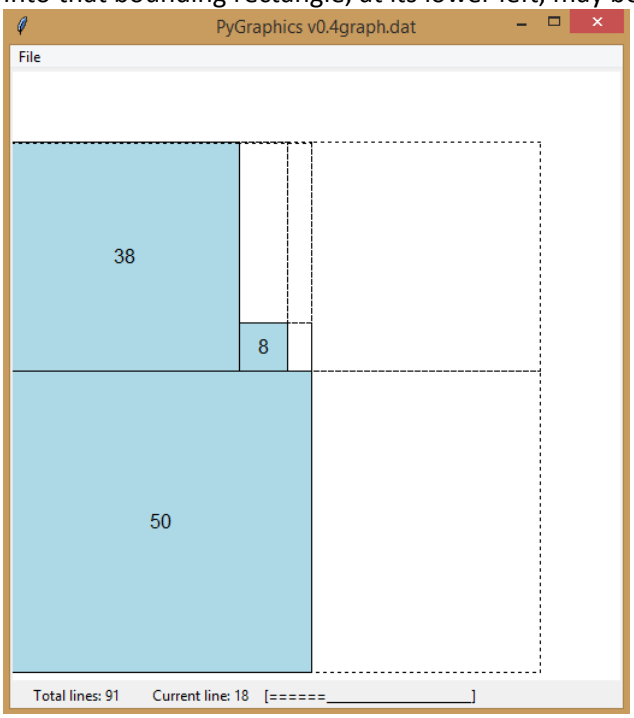


The overall bounding rectangle, using a clockwise order of vertices, is LL-UL-UR-LR. A square LL-B-C-E whose edge length is "size" is placed at the lower left of LL-UL-UR-LR. The three other rectangles, in the order used by this problem, are B-UL-A-C, C-A-UR-D, E-C-D-LR.

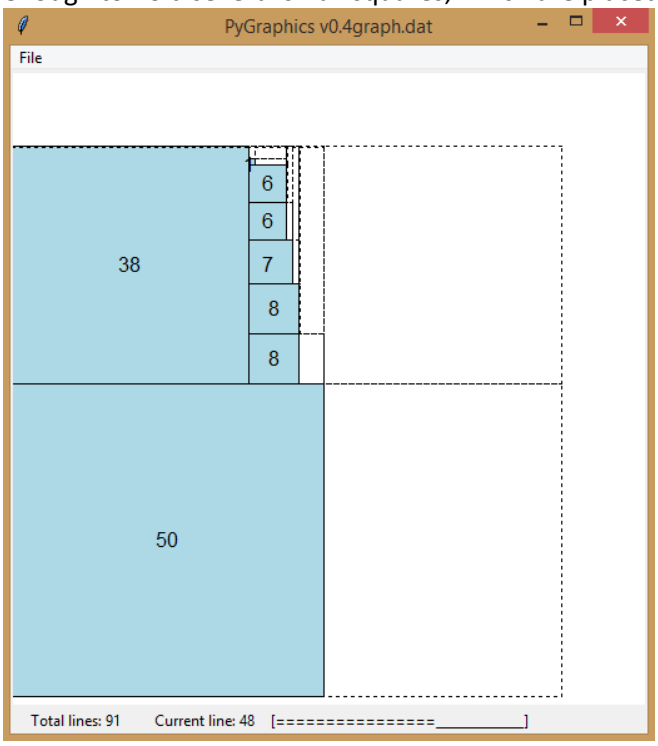
Next, your program will recurse upon the three rectangular portions not yet covered by a square. In this example, the first recursive call will be upon the bounding rectangle whose lower left is (0, 50), and whose upper right is (50, 88). Into that bounding rectangle, at its lower left, may be placed a square with edge length 38.



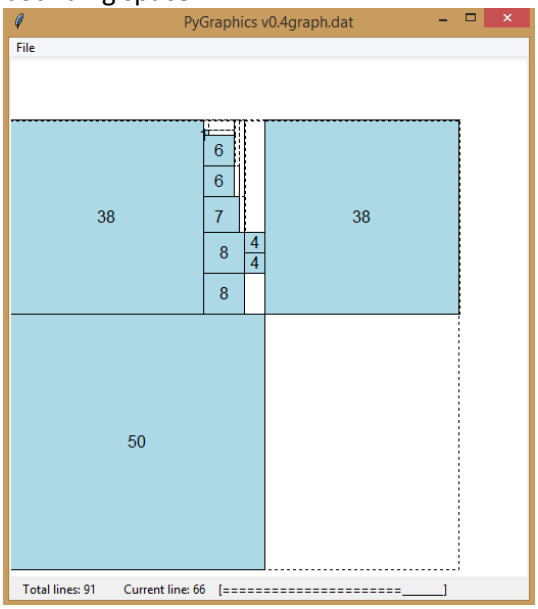
Next, your program will again recurse upon the three rectangular portions not yet covered by a square. In this example, the first two rectangular portions have size zero and so no squares may be placed. Recursion will continue to the third rectangular portion of a previous recursive call --- now the program is operating on the bounding rectangle whose lower left is (38, 50), and whose upper right is (50, 88). Into that bounding rectangle, at its lower left, may be placed a square with edge length 38.



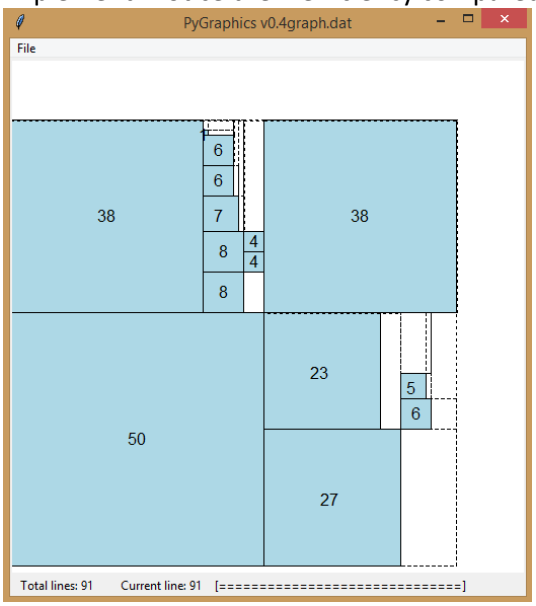
(Some steps omitted here.) The bounding area continues to be recursively divided but is still large enough to hold several small squares, which are placed in the next several recursive steps.



(Some steps omitted here.) As recursion progresses, available squares are placed within the current bounding space.



(Some steps omitted here.) This is the final placement of squares by the algorithm that you will implement. Notice the inefficiency compared to Diagram A: only 15 squares have been placed.



The squares which fit into the “ideal” packing, but were NOT placed by the inefficient algorithm you were asked to use, are two squares of size 23 and 15. The area of those two squares match the area of of the “gaps” in the diagrams above.