CSCI-1200 Data Structures — Fall 2011
Homework 6 — Box Packing Recursion

Your task for this homework is to write a box packing program using the techniques of recursion. Understanding the example of the non-linear word search program from Lecture 12 will be very helpful in thinking about how you will solve this problem. We strongly urge you to study and play with that program, including tracing through its behavior using a debugger or `cout` statements or both.

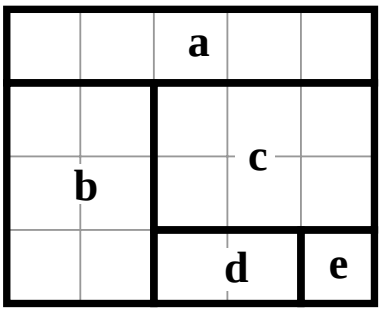
For this assignment, you will be given a two-dimensional rectangular box and a set of two-dimensional rectangular pieces to place in the box. The dimensions of the pieces are integer units, so you may think of the box as a two-dimensional grid of slots that are initially empty. When a piece is placed into the box, some of these slots are tagged as occupied. Pieces may not overlap; that is, they may not occupy the same cell in the grid. The pieces are not allowed to stick outside of the boundaries of the box, and you aren't allowed to cut the pieces! *Please carefully read the entire assignment before beginning your implementation.*

Your program should expect one command line argument, the name of the input file:

```
box_packer puzzle1.txt
```

Here's an example of the input file format:

```
5 4
5
a 5 1
b 2 3
c 3 2
d 2 1
e 1 1
```



The first line describes the dimensions of the box: the width=5 and the height=4. The next integer is the number of pieces to be packed in the box (in this example, 5 pieces). Finally, the pieces are listed, one per line. Each piece has an identifying character symbol and then a width and height. The pieces should be placed into the box in their original orientation; that is, *do not rotate the pieces.* Above is a diagram showing one solution to this particular box packing problem. This particular problem has 16 unique solutions. Your program will output (to std::cout) the number of solutions and an ASCII representation for each solution. See the example output on the course webpage. If the puzzle is impossible your program should output "`No solutions found`".

To implement this assignment, you must use recursion in your search. First you should tackle the problem of finding and outputting one legal solution to the puzzle (if one exists). Significant partial credit will be given for submissions that do this correctly. Full credit will be given to programs that find *all* of the solutions.

Once you have finished your implementation, analyze the performance of your algorithm using order notation. What important variables control the complexity of a particular problem? The height & width of the box, the number of pieces, the number of solutions, etc.? In your `README.txt` file write a concise paragraph (< 200 words) justifying your answer. If you have any notes you want the grader to read, include them in this file. For extra credit you may implement techniques to improve the performance of the basic algorithm and submit your code to the HW6 contest. Describe these improvements in your `README_contest.txt` file. If you're feeling brave (and you have some extra computer cycles), test your improved program on the larger test cases.

Do all of your work in a new folder named `hw6` inside of your Data Structures homeworks directory. Use good coding style when you design and implement your program. Be sure to make up new test cases and don't forget to comment your code! When you are finished please zip up your folder exactly as instructed for the previous assignments and submit it through the course webpage.

Box Packing Contest Rules

- Contest submissions are a separate homework submission. Contest submissions are due Saturday October 29th at 11:59pm. You may not use late days for the contest.
- Contest submissions *do not* need to use recursion.
- We will recompile (`g++ -O3 *.cpp`) and run all submitted entries on one of our machines. Programs that do not compile, or do not complete the basic tests in a reasonable amount of time with correct output will be disqualified and will not receive extra credit.
- Programs must be single-threaded.
- Programs must follow the output specifications: output to `cout`, output the number of solutions, and then output all of the solutions (in any order).
  - We will run your program by *redirecting* `cout` to a file:
    ```
    box_packer puzzle1.txt > output.txt
    ```
  - We will measure performance with the UNIX `time` command:
    ```
    time box_packer puzzle1.txt > output.txt
    ```
- Optional arguments for the contest:
  - `box_packer puzzle1.txt -single_solution`
    Only output the first solution found. Do not output the total number of solutions. If there are no solutions, print `"No solutions found"`.
  - `box_packer puzzle1.txt -rotate_pieces`
    The non-square pieces may be rotated.
- The contest will include puzzles that have no solution and puzzle that have holes (area of pieces $<$ area of box).
- Be sure to make up new test cases, and submit interesting test cases to the contest. Extra credit will be awarded for interesting test cases that are used in the contest. Caution: Don't make the test cases so difficult that your program cannot solve them in a reasonable amount of time!
- Extra credit will be awarded based on performance in the contest.
- In your `README_contest.txt` file, describe the optimizations you implemented for the contest, describe your new test cases, and summarize the performance of your program on all test cases.