CS358: Applied Algorithms

Assignment 1: Two Towers Revisited (due 2/19/2020)

Instructor: Sam McCauley

Instructions

All submissions are to be done through github. This process is detailed in the handout "Handing In Assignments" on the course website. Answers to the questions below should be submitted by editing this document. All places where you are expected to fill in solution are marked in comments with "FILL IN."

Please contact me at sam@cs.williams.edu if you have any questions or find any problems with the assignment materials.

This assignment can be done in pairs. **If two people are working on this assignment, please include both of your names in the source code here:**

Problem Description

INPUT: The input to the problem will be an array of at most 64 signed 64-bit integers (this array will be given as a pointer of type `int64_t*`), along with an integer representing how many elements there are in the array.

OUTPUT: The output is a single signed 64-bit integer whose bits represent a subset of the blocks in the array. For example, the subset consisting of only the first element in the array would be represented by the integer 1; the subset consisting of the first, second, and fourth elements of the array would be represented by 11.

GOAL: The input represents a set of blocks; the ith integer in the input represents the "area" of the ith block. The height of a block is the square root of its area.

The goal is to partition the blocks into two sets (which we call towers), such that the height of the towers is as close as possible.

Testing Parameters

The `main()` method of the testing program (in `test.c`) takes two arguments, each of which is a file containing instances of the two towers problem. An instance of the problem is represented by a sequence of integers (separated by spaces) on one line, and the intended solution (represented as a decimal number) on the second line. You can test your program by first running `make`, and then running `./test.out testData.txt timeData.txt`.

- For all instances, the best solution is guaranteed to be at least .0001 larger than the second-best solution (i.e. the smaller tower in the optimal solution is .0001 larger than any other tower smaller than half the height). This guarantee is to help rule out issues with floating point errors. This also means that the smaller tower is strictly smaller than the larger tower—you do not need to worry about tiebraking.
- These instances were tested using 64-bit `double`s. Higher-precision numbers such as `long double`s are acceptable (and are available on the lab computers), but are not necessary to obtain a correct solution. 32-bit `float`s are likely not going to be sufficient unless handled very carefully.
- All input block areas will be between 1 and 9223372036854775807 (this is the largest number that fits in a signed 64 bit integer). For the timed testing case, these inputs were (essentially) randomly generated. This means that you can assume that the inputs are approximately randomly distributed. However, your algorithm should work for any input.
- Two solutions with running times within .1 seconds of each other will be considered tied for the purposes of this assignment.

Questions

Problem 1. Please describe your implementation at a high level. What techniques and data structures did you use to implement meet in the middle? What optimizations did you use (if any)?[1]

Solution.

Problem 2. Can meet in the middle be modified to efficiently return the k best solutions, rather than just the best? Describe how this modification could be made and analyze its running time.

Solution.

Problem 3. Let's say that two towers are **balanced** if the width of the bottom block of each towers differs by at most 1. Describe how meet in the middle can find the two balanced towers that are closest in height if it exists. What is the running time of your approach?

Solution.

[1]It's OK if you don't use any optimizations because this is the first assignment, but the expectations will increase as time goes on.

Tips and tricks

- Remember to create a correct program before worrying about creating a fast one!
- You need to iterate through all possible subsets. One way to do this is to store each subset as an integer, and increment it to obtain a new subset. If you are unfamiliar with this technique, it may be useful to read the version of this lab given in CSCI 136, which can be found here for example: https://williams-cs.github.io/cs136-f19-www/labs/two-towers.html.
- Since the problem asks for the items that make up the solution, your meet in the middle approach will need to keep track of both the cost of a subset, and the items in the subset. There are many ways to do this. One easy way may be to keep track of a subset using a 64-bit integer, and to keep track of a partial solution using a struct containing both the cost of the subset and the integer representing its items. Other solutions are also possible, such as storing two arrays (one for costs and one for subsets), or storing a hash table keeping track of which solution corresponds to which cost.