Lab 7

Due 17 April

Handout 10
CSCI 136: Spring 2017
10 April

Warmup

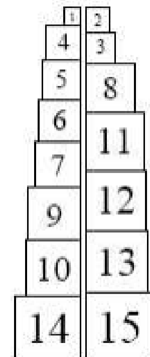
Solve the following problems *before* lab on Wednesday–they will greatly help you, and you should ask at the beginning of lab about ones that you can't solve. This lab handout and chapter 8 contain information to help you solve these. You may wish to review your Recursion lab solutions while working on this project.

1. Give an expression for the integer whose bit pattern is n zeros.
2. Give conventional-mathematical and integer-bitwise expressions for the integer whose bit pattern is n ones.
3. Given an int M that is to be treated as a bit mask (array) of n bits, write a while loop that prints the positions of the bits of M whose values are 1, counting from the *right*. For example, if M = 13 (note that decimal 13 is binary 1101), then your loop should print 0 2 3.
4. What is the difference in height for the towers that are the *second best* solution to the puzzle for $n = 15$?
5. You're going to write a program to solve the puzzle in the next section. While debugging your program, what is one to identify a probably optimal solution for $n = 15$, given the information in this handout?

1 Assignment

1.1 Goals

1. Solve a challenging puzzle exactly
2. Compare theoretical asymptotic behavior to experimental data
3. Abstract a useful iteration mechanism
4. Consider approximating the best solution for a large problem



1.2 The Puzzle

You are given n uniquely-sized cubic *blocks* whose face areas are 1, 2, ... n square centimeters. A *solution* to the puzzle is a pair of towers that collectively use all blocks. Your goal is to make the height of the towers as close as possible to each other. For example, the above figure shows a pair of towers created from 15 blocks. The heights of these towers are within 0.006% of each other–and this is only the *second best* configuration for 15 blocks!

Observations

1. Most pairs of towers will not contain equal numbers of blocks.
2. The *height* of block b is the square root of the area of one of its faces, $\sqrt{b}$.
3. Because the towers must collectively use all blocks, identifying which blocks are in *one* tower uniquely identifies a solution.
4. The total height of both towers combined is the sum of all block heights,

$$h = \sum_{b=1}^{n} \sqrt{b} = \sqrt{1} + \sqrt{2} + \cdots + \sqrt{n}.$$

5. It is therefore sufficient to find the subset of blocks that build the one tower of height closest but not exceeding $h/2$, where h is the height of all blocks.
6. The order of blocks within a tower does not matter.
7. A block is either in a tower or not in it. There are 2^n possible subsets of n blocks (including the empty set).
8. If we consider all subsets of of blocks, then we have exhaustively examined all possible towers and must have also observed the best solution.

1.3 Numeric Representation Review

We've previously used the bits of an integer as a way of representing the members of a subset. For a set of n elements, we need an n-bit integer. The Java int and Integer types each have 32 bits. If we needed more bits, we could use the 64-bit long type, or the *arbitrary* length BigInteger type.

Recall that the bit-shift expression a << b shifts the bits of integer a to the left by b bits (moving zeros in on the right side). The >> operator shifts to the right. The bitwise-and expression a & b combines each bit of a and b using a logical AND operation. For example a & 1 evaluates to 1 if the right-most bit of a is 1 (i.e., if a is odd!) and to 0 and if the right-most bit of a is 0 (i.e., if a is even). These operations allow us to treat integers as efficient arrays of boolean values. Simple addition and subtraction allow us to iterate through successive subset patterns.

Java represents real numbers using float and double types. The latter has twice the bit size (and far more than twice the precision!) of the former, so we'll use it to represent real numbers in this project. As with Integer, there are analogous Float and Double classes, and values automatically convert between the primitive types and classes. You'll need to compute square roots in this lab. In Java, Math.sqrt(x) returns a double that is the square root of x.

1.4 `TwoTowers.java`

Write a program in a file called `TwoTowers.java` that does the following, for some number of blocks n that is specified as a variable in the program*. It might be a good idea to abstract some of the parts into helper functions. *Tip: you should be using your prelab solutions, and don't need any data structures like Iterator or Vector.*

1. Compute the target height for the towers, $h/2$.
2. Iterate over all possible subsets of the n blocks, computing the height of each subset and keeping track of only the tallest subset whose height is less than or equal to $h/2$.
3. For the best subset, print:
   (a) The areas of the faces of the blocks in one of the two towers, e.g., the numbers on the blocks in the figure
   (b) The areas of the faces of the blocks in the *other* tower
   (c) The height of one of the towers
   (d) The *difference* in height between the two towers (*hint: you can compute it without measuring both towers!*)

1.5 `SubsetIterator.java`

We've seen through several labs that generating subsets is a common task, so let's abstract it. Most Java data structures can already produce Iterators that will *efficiently* iterate over their elements in order. For example java.util.Vector.iterator() returns a java.lang.Iterator object. Review chapter 8 and your lecture notes for details on Java iterators. Implement the following class that is initialized from *another* iterator and a bit mask, and then iterates over elements from the first iterator that correspond to 1's in the bit mask:

* Integer.parseInt is a handy tool if you'd like to set n through a command line argument.

```java
public class SubsetIterator<E> implements Iterator<E> {
    public SubsetIterator(Iterator<E> it, int bitMask);

    /* True if there are more elements in this subset */
    public boolean hasNext();

    /* Returns the next element and increments the iterator */
    public E next();

    /* Does nothing */
    public void remove();
}
```

For example,

```java
Vector<Integer> v = new Vector<Integer>();
v.add(1); v.add(15); v.add(20); v.add(31);

Iterator<Integer> it = new SubsetIterator<Integer>(v.iterator(), 13);
while (it.hasMore()) {
    System.out.println(it.next());
}
```

should print the numbers 1, 20, and 31. For full credit, do not read more elements of the source iterator than you need to. For example, your constructor should be brief.

1.6 Submitting your Work

Submit the following to the CS136 turnin folder:

1. `TwoTowers.java`
2. `SubsetIterator.java` (ensure that it *exactly* follows the specification)
3. Answers to the following questions, in a file named `answers.txt`. **Treat this like a paper and plan to spend at least an hour on this step.**
   (a) What is the difference in the height of the towers for the best solution for $n = 15$?
   (b) Draw a graph of the time to solve the tower problem vs. the value of n depicting the domain $n = 5$ through $n = 25$. Save it in `graph.png`[†], listing your source data in `answers.txt`. You may time programs with the Unix `time` command, as in the following:

       `time java -Xint TwoTowers`

       (The "-Xint" flag turns off some optimizations in the JVM and will give you more reliable results.) You don't have to look at every value–choose a few to test based on your understanding of the time complexity of your algorithm. Explain why the curve has the shape that it does.
   (c) Estimate the time to solve the 40- and 50-block problems, using mathematical justification. Do not attempt to actually run these.
   (d) Describe how you would redesign TwoTowers to solve the n-block problem using SubsetIterator and Vector. What are the benefits and drawbacks of that solution compared to your original design?

As always, put your name at the top of all of your files. You will be graded on **design**, documentation, style, and correctness. Be sure to document your program with appropriate comments, including a general description at the top of the file, a description of each method with pre- and post-conditions where appropriate. Use comments and descriptive variable names to clarify sections of the code.

[†] You can take a screenshot of any Mac program with command-shift-4, and can also draw the graph on paper by hand and take a photograph to move it onto the Mac.