Lab 5: The Two Towers[1]
CSCI 2101 – Fall 2018

Due: Tuesday, October 30, 11:59 pm
Collaboration Policy: Level 1
Group Policy: Individual

In this week's lab, you will use `Iterators` to solve a difficult problem. You will also gain experience measuring the execution speed of programs and an appreciation for the impact of algorithmic complexity on real-world running times.

Note that the amount of code you will need to write in this week's lab is less than in past labs, but the code is harder to wrap your head around. Plan accordingly!

1 The Two Towers

Suppose that you are given n uniquely sized cubic blocks, where each block has a face area (*not* side length) of 1 to n. In other words, each block k has a face area of k and a side length of $\sqrt{k}$. Your goal is simple: you want to use all n blocks to build two towers such that the heights of the towers are as close as possible.

For example, consider the case of $n = 15$. Below is a possible stacking of the 15 blocks into two towers, and in this particular stacking (assuming each unit is one-tenth of an inch), the heights of the towers differ by only 129 millionths of an inch.

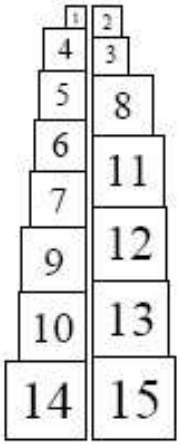

Figure 1: A possible stacking of $n = 15$ blocks.

You might be surprised to hear that the stacking configuration depicted in Figure 1 is actually only the *second-best* solution for the $n = 15$ case! This week, you will write a program that can find the best solution to this problem (for any n, with a few caveats) by exhaustively checking every possible pair of tower configurations.

[1] Adapted from a lab provided with *Java Structures*, D. Bailey

2 Program Interface

The basic operation of your program will be quite simple. On startup, the program should prompt the user for a desired number of blocks (i.e., the value of n):

```
Enter number of blocks:
```

Once a number of blocks is entered (you can assume $n \geq 2$), the program should solve the two towers problem by trying every configuration and then printing the following pieces of information:

1. The optimal tower height (i.e., the height of each tower if they were exactly equal in height).
2. In the best possible solution, the subset of blocks making up the *shorter* tower (the taller tower would simply be the rest of the blocks).
3. The height of the (shorter) tower represented by the above subset.
4. The difference between the height of the best shorter tower and the optimal tower height.
5. The clock time (i.e., actual real-world time) taken to solve the problem, in milliseconds. Note that this duration may vary from run to run or machine to machine.

Below is example output for $n = 10$:

```
Enter number of blocks: 10
Target (optimal) height: 11.23413909310205
Best subset: 1 4 5 8 10
Best height: 11.22677276241436
Distance from optimal: 0.0073663306876898815
Solve duration: 3 ms
```

3 Problem Analysis

To start, we can easily determine the total height of *all* the blocks by just summing their individual heights, i.e., the total height h is:

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

Thus, if we were able to produce two towers of exactly equal height, the height of each tower would be $h/2$. Now consider the best possible two tower configuration. The *shorter* tower of this configuration would have a tower height as close to $h/2$ as possible (ideally equalling it), but *without* exceeding it. Given a particular subset of blocks, it is easy to calculate the height of the corresponding tower, so we can solve the problem optimally by enumerating every possible subset of the n blocks (to find the subset with height closest to but no greater than $h/2$). In other words, our challenge is to find a way to enumerate every possible subset of blocks.

In more concrete terms, imagine that we have a list of n values. In the context of the two towers problem, each value corresponds to a particular block, but this analysis would equally apply

to any list (storing arbitrary values). The elements of the list are accessed by indices from 0 to $n-1$. If we can generate a subset of *indices*, then the elements at those indices gives us an actual subset of list elements (blocks or anything else). Thus, what we really need is a way to systematically generate all possible subsets of the n list indices. To think about how we can represent a subset of the n indices, we're going to use a few tricks based on the way computers represent numbers.

3.1 Binary Numbers

All computers represent information in **binary** (i.e., base 2) instead of the base 10 numbering system that humans generally use. A binary number is comprised of only 0's and 1's, and is built using powers of 2 instead of powers of 10. For example, the binary number 1101 is equal to $1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 13$ in base 10. Each 0 or 1 in a binary number is referred to as a *bit*. Although we normally don't worry about the fact that computers use binary, all information is internally represented as a series of bits.

We can represent a subset of n list indices using an n-bit binary number, in which each bit represents the presence (1 bit) or absence (0 bit) of the corresponding index. Bit 0 would be the right-most bit, while bit $n-1$ would be the left-most bit. For example, the 4-bit binary number 1101 (which is 13 in base 10) would represent the index subset $[0, 2, 3]$ (all indices included except bit 1). Similarly, the binary number 0101 (5 in base 10) would represent the index subset $[0, 2]$. Note that if we considered every 4-bit binary number starting with 0000 (the subset containing nothing) and ending with 1111 (the subset containing all four indices), we would have considered every possible subset along the way. Traversing subsets represented in this way is easy, since we're just counting from 0 (binary 0000) to 15 (binary 1111). Note that for an n-bit binary number, the smallest value is 0 (all bits are 0) and the largest value is $2^n - 1$ (all bits are 1).

Of course, if we want to represent subsets of n indices in this way, then we need to have numbers that are represented using at least n bits (since computers can only use a finite number of bits to represent a number). In Java, an `int` is represented using 32 bits, and a `long` (the larger integer type) is represented using 64 bits. For maximum flexibility, we can choose to represent the subset using a `long`, and should thus be able to consider the subset problem up to $n = 64$.

3.2 Binary Operations

Following the above, you will represent an index subset using a `long` (a single number), but will need to extract the actual indices included in the subset by working with the binary representation of the number. Doing so will require a few operators we haven't seen before. We needn't worry about exactly how these operators work, but there are two specific tricks we'll need:

- The *arithmetic shift* operator `<<` can be used to quickly compute powers of 2. In particular, the value 2^i can be computed by shifting one i places to the left. For example, the expression `1L << 7` computes the value 2^7 (the `L` just indicates that the 1 is a `long` rather than an `int`).

- The *bitwise and* operator `&` can be used to determine the value of a single bit in a number's binary representation. In particular, if `m` is some `long`, we can check bit i of `m` by computing the expression `m & (1L << i)`. This expression will have a value of 0 if bit i was 0, or some nonzero value (specifically, some power of 2) if bit i was 1. Note that the operator `&` is completely different from the *logical and* operator `&&` that you're used to for combining boolean expressions (don't mix them up!).

4 Program Design

Armed with our understanding of how to work with index subsets represented using single numbers, we can go about writing a program to solve the two towers problem.

4.1 Subset Iterator

First we need to build an `Iterator` that iterates through all element subsets of a given list. For example, if we're iterating over subsets of the list $[a, b, c]$, then the iterator should go about giving us all eight possible subsets (in no particular order): $[], [a], [b], [c], [a, b], [a, c], [b, c], [a, b, c]$.

Name your new class `SubsetIterator`. For maximum generality, the list that you're producing subsets from could store any type of object, so let's call that generic type `T`. In other words, each element (i.e., subset) that the iterator produces will be of type `List<T>` (a list containing all elements within that subset). Thus, the generic type of the `Iterator` interface will be `List<T>`.

Putting this all together, the declaration of your iterator will be the following:

```
public class SubsetIterator<T> implements Iterator<List<T>>
```

The constructor of the class should be given the list of elements that you are going to iterate over (i.e., generate subsets from). In order to implement the `Iterator` interface, you will then need to write the two core methods: `hasNext`, which says whether there are any remaining elements (subsets) to iterate over, and `next`, which actually produces and returns the next subset.

Internally, your iterator will need to keep track of the current subset using a `long` as detailed in Section 3. This value will increase from 0 (representing the first subset, containing nothing) all the way to $2^n - 1$ (representing the last subset, containing everything) as the iterator progresses.

Note that the `Iterator` interface also defines a third method `remove` (which removes the last element iterated over from the underlying collection), but this is an optional operation and there's no need for it here.

Once your iterator is implemented, test it by writing a `main` method in `SubsetIterator` with some test code. For example, you could create an `ArrayList` of characters and put the characters 'a' through 'e' in it. Then, create a `SubsetIterator` for this list and use it to print out all possible subsets of the list. If your iterator works correctly, it should produce the $2^5 = 32$ different subsets of the 5 characters in the list.

4.2 Two Towers with Iterators

At this point, you should have a functioning `SubsetIterator` class. Now you can use this class to solve the two towers problem. Create a new class named `TwoTowers`, which will just be a container for your (final) `main` method. This method should actually run the program as detailed in Section 2. To solve the two towers problem, create a list holding the heights of the n blocks (i.e., the values $\sqrt{1}, \sqrt{2}, \sqrt{3}, ...$) and use a `SubsetIterator` to iterate through the subsets of this list. For each subset, you just need to sum the values to find the height of the corresponding tower, and then pick the tallest tower (i.e., across all subsets) that's no taller than $h/2$. The subset corresponding to this tower is the best (shorter) tower for the given n. Once you have found the best subset, you can print out all the information specified in Section 2.

5 Implementation Tips

Here are some implementation pointers as you write your program.

5.1 Program Decomposition

Programming is all about breaking hard problems down into simpler pieces, solving those pieces on their own, then putting things back together. When you're writing the `SubsetIterator` class, keep that idea in mind – all you're doing is writing an iterator to produce subsets of a list, nothing more. In particular, nothing in the `SubsetIterator` class should have anything to do with the two towers problem specifically. Approach your code accordingly – you're solving the subset iteration problem first, and only afterwards actually considering the two towers problem.

Remember that you'll need to import `java.util.Iterator` within `SubsetIterator` in order to use the `Iterator` interface.

5.2 Binary Numbers

Beginners often get confused by thinking about binary numbers and think that there's something different about storing a number "in binary". There isn't – remember that *every* number in a computer program is stored in binary, regardless of how you specified the number in your program. For example, if I wrote `int i = 60`, even though I specified the value in decimal, the computer is still storing the value 60 in binary (which is 111100). The only significance of binary here is that we are using the binary representation to represent a subset (as described in Section 3) and can work with the individual bits of the number as detailed there.

5.3 Timing Code

Whenever you want to measure how much actual time is taken to run some code (i.e., wall clock time), you should write code to measure it – manually controlling a stopwatch or similar is cumbersome and much less precise than having the computer do your timing for you.

Measuring the time taken to execute code in Java is very simple – we just record the time before starting to run the relevant code, record the time after running the relevant code, then subtract the times to obtain the duration. The standard method to record the current time is `System.currentTimeMillis()`, which returns the current time represented as a `long`:

```
long startTime = System.currentTimeMillis();
doSomething(); // do something that we want to time
long duration = System.currentTimeMillis() - startTime;
System.out.println("Time to execute doSomething: " + duration + " ms");
```

If you're curious, the way the current time is represented as a `long` is that it's just the number of milliseconds that have passed since midnight on January 1, 1970. That particular date is completely arbitrary, but since everyone agrees on it as "time zero", we can use a single number like the one returned by `System.currentTimeMillis` as a specific timestamp. This approach is how most computers store dates and times.

5.4 Math Functions

There are a couple of useful math methods that you may wish to use in your program:

- You can use the `Math.sqrt` method to compute square roots. Note that `Math.sqrt` returns a `double`, which is the type you should use whenever you need to represent fractional values.
- You can use `Math.round` to round a `double` to the nearest `int` (for instance, if converting a side length like $\sqrt{5}$ back to its respective block number).
- There is a `Math.pow` method that can be used to compute powers, but you don't need to use it here. You should be able to use the shifting trick described in Section 3 whenever you need to raise a number to a power (which is also much faster than calling `Math.pow`).

6 Thought Questions

Once you have completed your program, use it to answer the following questions. Write your answers in the program `README` file.

1. What is the best solution to the 15-block problem?
2. Solve the 20-block, 21-block, and 22-block problems and record the time taken to solve each problem. You might want to run each test two or three times and average the results to get better measurements. What do you notice about the runtimes? Why does this result make sense given the design of the program?
3. Based on your empirical results from the previous question and your understanding of the time complexity, estimate how long it would take to solve the 50-block problem (you won't want to actually run this). Give your answer in some reasonable time unit (which shouldn't be milliseconds for this question).

7 Evaluation

As usual, your completed program will be graded on correctness, design, and style, as well as your thought question answers included in your `README`. Make sure that your program is documented appropriately and is in compliance with the coding and style guide. Lastly, make sure that your name is included in both your Java files as well as your `README`.

8 Submitting Your Program

Submit your program on Blackboard in the usual way. Remember to create a zip file named with your username and lab number, e.g., `sbowdoin-lab5.zip`, and upload that file.