Lab 5: The Two Towers[1]
CSCI 2101 – Fall 2021

Due (Section A): Tuesday, November 2, 11:59 pm
Due (Section B): Wednesday, November 3, 11:59 pm
Collaboration Policy: Level 1 (review full policy for details)
Group Policy: Individual

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

Note that while your program this week will be relatively short, the code is somewhat tricky to grasp conceptually. Also note that unlike the last several labs, this lab must be completed individually. 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 square units. In other words, each block k has a face area of k square units and a side length of $\sqrt{k}$ units. 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, suppose that $n = 15$. Below is a possible stacking of the 15 blocks into two towers. In this particular stacking, the heights of the towers differ by only 129 millionths of a unit.

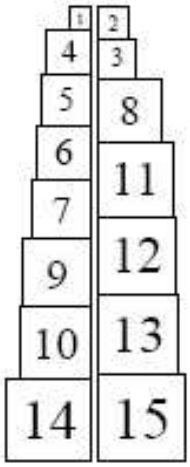

Figure 1: A possible stacking of $n = 15$ blocks.

You might be surprised to learn 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 practical 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 the block count is entered (which you can assume is at least 2), the program should solve the two towers problem by checking every configuration and then printing the following 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 tower configuration, 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. This height should be close to, but not larger than, the optimal tower height.
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 determine the total height of *all* the blocks by just summing their individual heights. The total height h of all the blocks (which will be divided between the two towers) is:

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

If it were possible to produce two towers of exactly equal heights, then the height of each tower would be precisely $h/2$. Now consider the best *feasible* two tower configuration. Of this configuration, one tower is shorter and one tower is taller (unless it's perfect, in which case they're the same). The *shorter* tower of this configuration would have a height of $(h/2 - \epsilon)$, where ϵ is as small as possible but nonnegative. Thus, one way to find this optimal configuration is to enumerate every possible subset of n blocks, looking for the subset with height closest to but no greater than $h/2$. This subset would correspond to the shorter tower in the optimal configuration, while the rest of the n blocks (i.e., those not in the subset) would correspond to the taller tower.

In more concrete terms, imagine that we have a `List` of n values, each of which corresponds to a block – the values themselves might be, e.g., the height of each block ($\sqrt{1}$, $\sqrt{2}$, etc). The elements of the list are stored from indices 0 to $n-1$. A particular *subset* of blocks (i.e., a particular tower configuration) can be represented as a subset of those n indices. In other words, rather than thinking about enumerating subsets of blocks, we can instead think about enumerating subsets of list indices (which then correspond to the specific list values at those indices). Thus, what we really need is a way to systematically generate all possible subsets of the n list indices.

Note that the subset-generation problem we're describing isn't at all specific to the two towers problem. In the specific context of the two towers problem, each subset corresponds to a group of blocks, but the list over which we're enumerating subsets of indices could potentially be storing any kind of data item. Thus, what we're really doing here is tackling a more general problem (how to enumerate subsets of list indices), and then applying that general solution to the specific scenario of the two towers.

To think about how we can represent (and then enumerate) subsets of the n list indices, we're going to use a few tricks based on the way that computers represent numbers, as described below.

3.1 Binary Numbers

All computers represent information in **binary** (base 2) instead of the decimal (base 10) numbering system that humans generally use. A decimal number consists of digits from 0 to 9, and each digit represents a power of 10. For example, the decimal number 382 is deconstructed as $3 * 10^2 + 8 * 10^1 + 2 * 10^0 = 382$. A binary number, on the other hand, 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 (that's "one-one-zero-one") is equal to $1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 13$ in decimal. Each 0 or 1 in a binary number is referred to as a **bit**. Note that for an n-bit binary number, the smallest decimal value is 0 (the n-bit number consisting of all 0 bits, e.g., 0000 = 0) and the largest decimal value is $2^n - 1$ (the n-bit number consisting of all 1 bits, e.g., 1111 = 15).

Returning to the subset enumeration problem, we can be clever by noting that we can represent a subset of n list indices using an n-bit binary number, in which each bit represents the exclusion or inclusion of the corresponding index as a 0 or 1, respectively. The right-most bit would correspond to the inclusion or exclusion of index 0, while the left-most bit would correspond to the inclusion or exclusion of index $n-1$. For example, the 4-bit binary number 1101 would represent the index subset $[0, 2, 3]$, while the binary number 0101 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. When representing the subsets in this way, enumerating all possible subsets is easy, since all we're actually doing is counting from 0 (binary 0000 in 4 bits) up to $2^n - 1$ (binary 1111 = decimal 15 in 4 bits).

One notable limitation of this approach is that if we want to represent a subset of n indices, then we need to use a number that is internally stored using at least n bits. Computers can only use a finite number of bits to represent a number, with the number of bits used depending on the data type. In Java, an `int` is represented using 32 bits, while 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 represent a subset of up to $n = 64$ indices.

3.2 Binary Operations

In order to use the above approach, we'll need to work with the actual binary representations of numbers in Java, which will require a few operators that we haven't seen before. We needn't worry too much about the details of how these operators work, but we'll need to know a few useful tricks:

- The *left shift* operator "`<<`" can be used to quickly compute any power 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 means that the value is a `long`, not 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))`. If bit i of m is a 0, then this expression will evaluate to 0, while if bit i is a 1, then this expression will evaluate to some nonzero value (specifically, the value 2^i). Note that the operator `&` is completely different from the *logical and* operator "`&&`" that you're familiar with for combining boolean expressions (don't mix them up!).

4 Program Design

Armed with our understanding of how to represent an index subset using a single number, 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 (not just integers, as in the above example), 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>`. 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 (an iterator that returns a `List<T>` object every time `next` is called):

```
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 (the first subset, containing nothing) all the way to $2^n - 1$ (the last subset, containing everything) as the iterator progresses.

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 integers and add the values from 1 to 7 to 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, your code should print out $2^7 = 128$ different subsets of the 7 integers in the list. Make sure this works before moving on.

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 (real) `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

As usual, here are some implementation pointers as you write your program.

5.1 Subset Enumeration

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 to import `java.util.Iterator` in `SubsetIterator` to use the `Iterator` interface.

5.2 Timing Code

The final piece of output specified in Section 2 is the amount of time (i.e., wall clock time) taken to solve the problem. Whenever you want to measure how much real-world time is taken to run a piece of code, you should include code to have the computer do the timing for you, which is much more precise than using a stopwatch or any other manual method.

Measuring the time taken to execute code is very simple – just record the current time before starting to run the relevant code, record the current time after running the relevant code, then subtract the two times to obtain the duration spent running the code. The standard method to record the current time in Java is `System.currentTimeMillis()`, as demonstrated below:

```
long startTime = System.currentTimeMillis();
doSomething(); // run some code that we want to time
long duration = System.currentTimeMillis() - startTime;
System.out.println("Time executing doSomething was " + 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 (a very large number). 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.

5.3 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 instead 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`, although can only be used if computing a power of 2).

6 Thought Questions

Once you have completed your program, use it to answer the following questions. Write your answers in the program `README` file. If your IDE doesn't automatically create this file, create it yourself – this file should be a plain text file (not a Word document or other file format), which you can create in TextEdit on a Mac or Notepad on a PC.

1. What is the best solution to the 15-block problem?
2. Solve the 20-block, 21-block, and 22-block problems and list the times 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 of the program, estimate how long it would take to solve the 50-block problem (you won't want to actually run this). Specify your answer in some reasonable time unit (hint: it shouldn't be milliseconds for this scenario)!

7 Evaluation

Your lab submission will be evaluated both on your answers to the questions above (in your `README` file) and on your actual program. As usual, your program will be evaluated for correctness, design, and style. 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 all of 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 (which should contain both your Java files and your `README`).