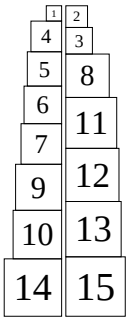
**Computer Science CS136 (Fall 2021)**

*Duane Bailey & Aaron Williams*

Laboratory 6

*The Two Towers Problem*

Suppose that we are given $n$ uniquely sized cubic blocks and that each block has a face area between 1 and $n$. Build two towers by stacking these blocks. How close can we get the heights of the two towers? The following two towers built by stacking 15 blocks, for example, differ in height by only 129 millionths of an inch (each unit is one-tenth of an inch):




Still, this stacking is only the *second-best* solution! To find the best stacking, we could consider all the possible configurations.

We *do* know one thing: the total height of the two towers is computed by summing the heights of all the blocks:

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

If we consider all the *subsets* of the $n$ blocks, we can think of the subset as the set of blocks that make up, say, the left tower. We need only keep track of that subset that comes closest to $h/2$ without exceeding it.

In this lab, we will represent a set of $n$ distinct objects by a `Vector`, and we will construct an `Iterator` that returns each of the $2^n$ subsets.

**Discussion.** The trick to understanding how to generate a subset of $n$ values from a `Vector` is to first consider how to generate a subset of indices of elements from 0 to $n-1$. Once this simpler problem is solved, we can use the indices to help us build a `Vector` (or subset) of values identified by the indices.

There are exactly $2^n$ subsets of values 0 to $n-1$. We can see this by imagining that a coin is tossed $n$ times—once for each value—and the value is added to the subset if the coin flip shows a head. Since there are $2 \times 2 \times \cdots \times 2 = 2^n$ different sequences of coin tosses, there are $2^n$ different sets.

We can also think of the coin tosses as determining the place values for $n$ different digits in a binary number. The $2^n$ different sequences generate binary numbers in the range 0 through $2^n - 1$. Given this, we can see a line of attack: count from 0 to $2^n - 1$ and use the binary digits (*bits*) of the number to determine which of the original values of the `Vector` are to be included in a subset.

Computer scientists work with binary numbers frequently, so there are a number of useful things to remember:

- An `int` type is represented by 32 bits. A `long` is represented by 64 bits. For maximum flexibility, it would be useful to use `long` integers to represent sets of up to 64 elements.

- The *arithmetic shift* operator (<<) can be used to quickly compute powers of 2. The value $2^i$ can be computed by shifting a unit bit (1) `i` places to the left. In Java we write this `1<<i`. This works only for nonnegative, integral powers. (For long integers, use `1L<<i`.)
- The *bitwise and* of two integers can be used to determine the value of a single bit in a number's binary representation. To retrieve bit `i` of an integer `m` we need only compute `m & (1<<i)`. The result is either 0, if bit `i` is not set, or non-zero if it is set.

**This Week's Tasks.** Armed with this information, the process of generating subsets is fairly straightforward.

1. Clone your `lab6` repository:

   ```
   cd ~/cs136
   git clone https://evolene.cs.williams.edu/cs136-labs/22xyz3/lab6.git
   ```

   replacing 22xyz3 with your CS username. This will create the directory `~/cs136/lab6`. Here, we'll focus on modifying files describing two iterators: the `SubsetIterator` and `RulerIterator`.

2. Construct a new extension to the `AbstractIterator` class called `SubsetIterator`. (By extending the `AbstractIterator` we support the `Iterable` and `Iterator` interfaces.) This new class should have a constructor that takes a `Vector` of objects as its sole argument. The objects represent the universe of values used in this problem. Subsets of this `Vector` will be returned as the result of calling the `Iterator`'s `next` method. Think carefully about how this type is parameterized.

3. Internally, a `long` value is used to represent the current subset. This value increases from 0 (the empty set) to $2^n - 1$ (the entire set of values) as the `SubsetIterator` progresses. Write a `reset` method that resets the subset counter to its initial state, 0.

4. Write a `hasNext` method that returns `true` if the *current* value is a reasonable representation of a subset. You should be able to call `hasNext` without changing the state of the iterator.

5. Write a `get` method that constructs a `Vector` of values that are part of the current subset. If bit `i` of the current counter is 1, element `i` of the `Vector` is included in the resulting subset `Vector`. This is the workhorse method of the `SubsetIterator` class. Take your time and think about how you might test if this method works.

6. Write a `next` method. It returns a `Vector` of values in the current subset *before* incrementing the counter. As with many of the `AbstractIterator` classes we have written, you will want to call `get` at some point, just to leverage all the hard word you've already done.

7. For an `Iterator` you would normally have to write a `remove` method. If you extend the `AbstractIterator` class, this method is provided and will do nothing (this is reasonable).

You can now test your new `SubsetIterator` by having it print all the subsets of a `Vector` of values. Remember to keep the `Vector` small. If the original values are all distinct, the subsets should all differ by value as well.

To solve the two-towers problem, write a `main` method in the `SubsetIterator` class that inserts the values $\sqrt{1}, \sqrt{2}, \ldots, \sqrt{n}$ (use `Math.sqrt`) as `Double` objects into a `Vector`. A `SubsetIterator` is then used

to construct $2^n$ subsets of these values. The values of each subset are summed, and the sum that comes closest to, but does not exceed,[1] the value $h/2$ is remembered. After all the subsets have been considered, have your program print the best solution it encountered along the way.

Find the best solution to the 15-block problem and include that solution in comments near your `main` method.

If you've made it this far, then you've earned 9 points. And: you've solved a very challenging problem...well done!

**Extension.**

Our strategy for solving the Two Towers Problem can be categorized as an *exhaustive search* or as a *generate and test algorithm*, meaning that we create every possible solution to the problem, evaluate it, and keep the best one. This approach is often dismissively described as *brute force*, with the implication that raw computing power is being leveraged rather than creativity or intelligence. In reality, exhaustive search can be interesting and challenging, especially if your goal is to do it as efficiently as possible.

Let's begin by considering the amount of time we currently spend *processing*—that is *generating* and *evaluating*—each possible solution to an $n$-block Two Towers Problem:

1. We spend $O(n)$-time *generating* each possible solution. It takes $n$ operations to convert an integer into a subset of the $n$ blocks.

2. We spend an additional $O(n)$-time *evaluating* each possible solution: we must iterate over as many as $n$ elements of the subset to compute the height of the associated tower. (With care you can see that these subsets contain, on average, $\frac{n}{2}$, elements.)

As a result, we spend a total of $O(n + n)$-time, or simply $O(n)$-time, to process each possible solution.

In this extension, we will focus on the following goal:

*Process each possible solution in* $O(1)$*-time.*

In other words, we will aim to generate the next subset in $O(1)$-time, and evaluate it in $O(1)$-time. To illustrate what needs to be done, and what needs to be avoided, consider the following table. It contains three successive steps using the `SubsetIterator` strategy to solve the 6-block problem. They correspond to the $n = 6$ bit integers, 14, 15, and 16:

| integer | binary string | subset | height calculation |
|---|---|---|---|
| 14 | 001110 | $\{4, 3, 2\}$ | $\sqrt{4} + \sqrt{3} + \sqrt{2}$ |
| 15 | 001111 | $\{4, 3, 2, 1\}$ | $\sqrt{4} + \sqrt{3} + \sqrt{2} + \sqrt{1}$ |
| 16 | 010000 | $\{5\}$ | $\sqrt{5}$ |

First consider the integers 14 and 15. Notice that the binary strings, subsets, and height calculations are all similar to each other for these two integers. Specifically, the two binary strings differ in one bit, which means that the subsets differ in one element, and so the height calculations differ in one value. As a result, we can transition from the first row of the table to the second row of the table in $O(1)$-time. We would simply *modify* the binary string by complementing one bit, and *update* the height calculation by adding

[1] The reason we're only interested in subsets that *do not exceed* $h/2$ is because each configuration is represented twice: once with the subset of blocks on the *left*, and another with the same subset on the `right`. Without loss of generality, we're interested in solutions where the left tower is shorter than the right.

one value. There are still issues to be sorted out — for example, we don't want to spend $O(n)$-time figuring out that we can perform the transition in $O(1)$-time — but hopefully the basic idea is clear.

The most pressing issue is that our current strategy doesn't always have such smooth transitions. For example, last two rows of the table illustrate that successive integers can yield wildly different binary strings, subsets, and height calculations. More generally, our current strategy has many instances in which $O(n)$ changes are needed. In particular, the integer $2^{n-1} - 1$ has binary representation $011\ldots1$, while the next integer $2^{n-1}$ has binary representation $100\ldots0$, and so literally everything has changed. This extreme example only occurs once, but smaller "roll-overs" requiring $O(n)$-time occur frequently.

For the reasons outlined above, we'll need a new strategy for generating the possible solutions. Our new strategy is based on a *Gray code*,[2] and the *ruler sequence*[3] that generates it. The Gray code is an ordering of the $n$-bit binary strings in which successive strings differ in only a single bit, and the ruler sequence provides the bit indices that are changed (with 0-based indices). Below are illustrations for $n = 3$.

| ruler sequence | | 0 | 1 | 0 | 2 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Gray code ($b_2b_1b_0$) | 000 | 001 | 011 | 010 | 110 | 111 | 101 | 100 |

This order of binary strings may seem strange, but you should be able to verify that every string is included, and that each is obtained from the previous by changing the bit specified by the ruler sequence. For example, 010 is followed by 110 by changing bit $b_2$.

We have provided a file `RulerIterator.java` that includes an iterator for generating each value in the ruler sequence in $O(1)$-time. As with your testing of `SubsetIterator`, use the `main` method of the `RulerIterator` to identify the best configuration of blocks for the 15-block Two Towers Problem:

1. Do not write any new classes. Instead, use the indices returned by the `RulerIterator` to *modify* your current subset in constant ($O(1)$) time (i.e. without loops).

2. Each time you modify the subset, make sure you *update* the current height. Do not *recompute* the height from scratch, but carefully update it based on the value returned from the `RulerIterator`. Again: no loops!

3. In the spirit of simplicity, feel free to keep track of only the minimum difference in height seen so far. Each time you update this minimum difference, neatly print the difference and the corresponding subset of block numbers. As your program searches the space, you should see better and better solutions. The last one printed will be the best.

4. Verify that the `RulerIterator` approach gives the same answer as the `SubsetIterator` approach.

If you've made it this far, congratuations: you've solved this challenging problem in *two* interesting ways!

**Submitting Your Work.** To get credit for this week's lab make sure that you've added, committed, and pushed your `SubsetIterator.java` and `honorcode.txt` (containing your signature). If you did any work on the extension, make sure you add, commit, and push your changes to `RulerIterator.java`. Recall that `git status` and `git push` will help to verify that everything is up-to-date.

⋆

[2] The invention of Gray codes is attributed to Frank Gray and his colleagues at Bell Labs in the 1940's.

[3] You can read more about the ruler sequence at the Online Encyclopedia of Integer Sequences: `https://oeis.org/A007814`.