Computer Science 62
Lab 4

Wednesday, February 19, 2014

The programming exercise for today's laboratory is about the Two Towers problem. Before reading this lab writeup, be sure to read about the Two Towers problem in your textbook (Java Structures) Section 8.7.

Problem Description

Your textbook gives a strategy for generating all possible subsets of the set $\{0, 1, \ldots, n-1\}$ using bit operations. There are 2^n such subsets. For example, when $n = 3$, the possible subsets are:

```
[]
[0]
[1]
[0, 1]
[2]
[0, 2]
[1, 2]
[0, 1, 2]
```

Notice that there are $2^3 = 8$ subsets listed above. How can we generate these subsets? Suppose you had n bits[1]. Each bit represents one of the elements in the set $\{0, 1, \ldots, n-1\}$. If the ith bit is 1, then we include the ith element in the subset. If the ith bit is 0, then we do not include the ith element in the subset. See Table 1 for an example.

As your book says, this suggest a strategy for generating a subset of $\{0, 1, \ldots, n-1\}$. To generate a subset,

- Let `counter` be a number between 0 and $2^n - 1$.
- Loop over the first n bits of `counter`
- Use the *arithmetic shift* operator (`<<`) and the *bitwise and* operator (`&`) as described in your textbook to check whether a bit is 1 or 0.
- If the bit is 1, include the corresponding element in the subset.

[1]The word "bits" stands for "binary digits". A bit can take on the value 0 or 1

Counter	Bit Representation	Subset
0	000	[]
1	001	[0]
2	010	[1]
3	011	[1, 0]
4	100	[2]
5	101	[2, 0]
6	110	[2, 1]
7	111	[2, 1, 0]

Table 1: An example for $n = 3$

IntegerSubsetIterator

For the first step, you are to write a class

```
public class IntegerSubsetIterator implements Iterator<ArrayList<Integer>>
```

whose constructor takes a single integer `size` and creates an object that iterates over all the subsets of $\{0, 1, \ldots, \texttt{size-1}\}$. There will be $2^{\texttt{size}}$ subsets.

The iterator will return "subsets" as ArrayLists whose elements are the elements of a subset. For example, running the code

```
IntegerSubsetIterator iter = new IntegerSubsetIterator(3);
while (iter.hasNext()){
    System.out.println(iter.next());
}
```

will produce the output:

```
[]
[0]
[1]
[0, 1]
[2]
[0, 2]
[1, 2]
[0, 1, 2]
```

Be sure that you understand what the iterator is doing before writing any code.

The `Iterator` interface requires only three methods: `hasNext`, `next`, and `remove`. The `remove` is not used here so think carefully about how to "implement" a method that performs an unsupported operation.

Two Towers

(Insert Lord of the Rings reference here :)

Now that you have created the subset iterator, you are ready to use it to solve the Two Towers problem. As Bailey describes in his book, one approach is to compute the heights of all the blocks

$$h = \sum_{i=1}^{15} \sqrt{i},$$

and then to look through all subsets of $\{1, 2, \ldots, 15\}$ to find the one whose height comes closest to, without exceeding, $h/2$.

Write a method inside of `IntegerSubsetIterator`:
`public static void twoTowersSolver(ArrayList<Double> blockSizes)`

that takes as input an ArrayList of block sizes (the easiest way is just to represent a block size by the size of one side). This method should iterate through all of the possible subsets and find the one subset whose sum is closest to $h/2$.

There is one complication: the `IntegerSubsetIterator` class generates subsets of $\{0, 1, \ldots, 14\}$ and you need to correlate these to the actual block sizes. A loop something like:

```
ArrayList<Integer> set = it.next();

int height = 0;

for( Integer i: set ){
   height += blockSizes.get(i);
}
```

should solve this problem.

Now, try out some different block sizes and print out the difference. For example:

```
public static void main(String args[]){
   ArrayList<Double> blocks = new ArrayList<Double>();

   for( int i = 1; i <= 15; i++ ){
      blocks.add(Math.sqrt(i));
   }

   System.out.println(twoTowersSolver(blocks));
}
```

Have your program print the actual half-height, the height of the best solution, and the blocks in the best solution.

Once you have a working program, experiment by increasing the number of blocks. Try values including 24, 28, and 33. Explain in the comments the results and the time it takes to get them.