CSCI 136: Fall 2017
Handout 10
12 November

Lab 10

Due 19 November

The Two Towers (Not a partner lab)

1 Prelab

Write an iterator that iterates over the characters of a `String`. Repeated calls to the iterator's `next()` method will return the first character of the string, then the second character, and so on.

More specifically, you are to complete the design of the class `CharacterIterator` and implement the following constructor and methods (you may add helper methods and class variables):

```
public class CharacterIterator extends AbstractIterator<Character> {
  public CharacterIterator(String str) { ... }
  public Character next() { ... }
  public boolean hasNext() { ... }
  public void reset() { ... }
  public Character get() { ... }
}
```

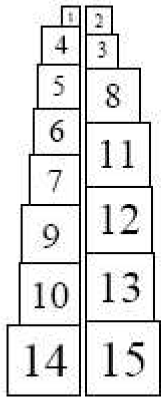
The `Character` class is the wrapper class for `char` values. You use it much like we use `Integer` for `int` values. The Java compiler will automatically convert `char` values to `Character` objects when necessary via a technique called "autoboxing."

Re-read Chapter 8 in your textbook to review Iterators. I highly recommend you complete your Prelab solutions before lab on Wednesday if possible. Submit them (in a file named CharacterIterator.java) with the rest of your code at the end of the lab.

2 Lab Program

Goal. To solve a difficult problem using Iterators. (Note: This lab is a variation of the Lab from Chapter 8, modified to use generic classes.)

Discussion. Suppose that we are given n uniquely sized cubic blocks and that each block has a face area between 1 and n. If we build two towers by stacking these blocks, how close can we get their heights? The following two towers built by stacking 15 blocks, for example, differ in height by only 129 millionth 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. Since the height of each block is the square root of its face area, this is:

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

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 (where h is the total height of all n blocks).

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

2.1 Procedure

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 the values from 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 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 (called 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:

- A Java int type is represented by 32 bits. A Java 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 `1L << i`. This works only for nonnegative, integral powers. The constant `1L` is the value one stored as a 64-bit long value. Using this constant ensures that we are using a 64-bit shift operation resulting in a `long` value instead of a 32-bit operation resulting in an `int` value.
- The “bitwise and” of two numbers can be used to determine the value of a single bit in a number’s binary representation. To retrieve bit `i` of a long integer `m`, we need only compute `m & (1L << i)`.

Armed with this information, the process of generating subsets is fairly straightforward. One strategy is the following:

1. Construct a new extension to the `AbstractIterator` class. (Note: By extending `AbstractIterator` we support both the `Iterator` and `Enumeration` interfaces.) This new class should have a constructor that takes a `Vector<E>` as its sole argument. Subsets of this `Vector` will be returned as the `Iterator` progresses.

   Name this class `SubsetIterator`, and be sure to import `structure5.*` and `java.util.Iterator` at the top of your file. Your `SubsetIterator` should be completely generic. It should know nothing about the values it is iterating over. Thus, the declaration will be:

   ```
   public class SubsetIterator<E> extends AbstractIterator<Vector<E>>
   ```

2. 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 `Iterator` progresses.
3. Write a `reset` method that resets the subset counter to 0.
4. Write a `hasNext` method that returns `true` if the current value is a reasonable representation of a subset.

5. Write a `get` method that returns a new `Vector<E>` of values that are part of the current subset. If bit `i` of the current counter is 1, element `i` of the original `Vector` is included in the resulting subset `Vector`.
6. Write a `next` method. Remember it returns the current subset *before* incrementing the counter.
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).

Incrementally test. You can now test your new `SubsetIterator` by having it print all the subsets of a `Vector` of values. For example, write a `main` method for your `SubsetIterator` that creates a `Vector<Integer>` with the first 8 `Integers` (0 through 7), creates a `SubsetIterator<Integer>` with this `Vector<Integer>`, and then prints out all subsets returned. Make sure you end up with all 256 different subsets printed.

Two Towers. To solve the two-towers problem, write a main method in a new class `TwoTowers` that inserts the values $\sqrt{1}, \sqrt{2}, ..., \sqrt{n}$ into a `Vector<Double>` object. (To compute the square root of `n`, you can use the `Math.sqrt(n)` method.) 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, the value $h/2$ is remembered. After all the subsets have been considered, print the best solution (the blocks heights are square roots, so it may be easier to print out the areas instead).

2.2 Other applications

Enumerating subsets is a powerful technique. We solved the two towers problem by picking the single best subset among all possible subsets of blocks in a `Vector<Double>`. Now we would like you to use your `SubsetIterator` to solve a very different kind of problem: list all words that appear as subsequences of a given string.

Subsequences. A subsequence s is a sequence that can be derived from another sequence t by deleting 0 or more elements from t. The ordering of the original elements is preserved in s. For example, the string "programming" has many subsequences, such as "prom", "ram", and "ring".

To solve this task, you will create a new class `SubWords`. In `SubWords`, you will:

- Implement a `main` method that expects two `Strings` as command-line arguments. The first argument is the name of a file that contains a lexicon. The second argument is a string whose subsequences you will enumerate in order to find words.
- Read the lexicon from the file (e.g., the `ospd2.txt` file from Lab 8), and store the lexicon in an appropriate data structure. You should choose either a `LexiconTrie` (from your Lab 8), an `OrderedVector`, or a `RedBlackSearchTree`.
- Convert your input string (second command-line argument) into a `Vector<String>`, where each string in the vector is a single character of the input string, and create a `SubsetIterator<String>` for that `Vector<String>`. Note: You could create a `Vector<Character>` if you prefer.
- Enumerate through all subsets using your `SubsetIterator`. Each subset represents one subsequence of your input word. For each subsequence, convert it to a `String`, and print that `String` if it is a valid word (i.e., the string is contained in your lexicon).
- In addition, count the number of subsequences that form valid words. You may count each instance of a word, even if it is a duplicate.

An example usage is:

```
$ java SubWords ospd2.txt bill
bi
ill
bill
Words found: 3
```

Optional extension. There are multiple data structures we could use to store a lexicon. Implement your `SubWords` class using a second data structure and compare the performance of the two implementations. In particular, you might compare a `LexiconTrie` to one of the classes that implement the `OrderedStructure` interface. Which is faster? Why? Which would you expect to use less memory? Submit both versions and answer these questions in your `README.txt` file for bonus credit.

A note on measuring performance. If you imagine that you will use a data structure repeatedly once it is initialized, then you might not want to count the time of construction/initialization in your performance measurement. The static method `System.currentTimeMillis()` might be useful for this task.

Thought Questions. Consider the following questions as you complete the lab:

1. What is the best solution to the 15-block problem?

2. How long does it take your program to find the answer to the 20-block problem? 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.)

   The output from the `time` program contains 3 lines:

   ```
   real 0m1.588s
   user 0m1.276s
   sys 0m0.922s
   ```

   If you read the `time` manual page, you can see the meaning of each line. The `real` line shows the "wall clock" time, which is what we care about.

   Based on the time taken to solve the 20-block problem, about how long do you expect it would take to solve the 21-block problem? What is the actual time? How about the 25-block problem? Do these agree with your expectations, given the time complexity of the problem? What about the 40- and 50-block problems? (These will take a *very* long time. Just estimate based on the run times of the smaller problems).

3. This method of exhaustively checking the subsets of blocks will not work for very large problems. Consider, for example, the problem with 50 blocks: there are 2^{50} different subsets. One approach is to repeatedly pick and evaluate random subsets of blocks (stop the computation after 1 second of elapsed time, printing the best subset found). How would you implement `randomSubset`, a new `SubsetIterator` method that returns a random subset? (Describe your strategy. You do not need to actually implement it.)

2.3 Deliverables

Complete the Prelab section (ideally) before lab on Wednesday. You may find it helpful to review the Iterators chapter (Chapter 8) of the textbook before coming to lab.

Create a lab directory called `<unix>-lab10` that includes the following:

1. Solutions to the Prelab problem in a file called `CharacterIterator.java`.

2. Your `SubsetIterator.java`, `TwoTowers.java`, and `SubWords` files. **Please do not name them anything else.**

3. Your answers to the three thought questions, included in a file called `README.txt`.

4. If you complete the optional extension, include your solution in a file called `SubWords2.java`.

As in all labs, 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. Also use comments and descriptive variable names to clarify sections of the code which may not be clear to someone trying to understand it.