Lab 12

Due: 22 April

Handout 12
CSC 062: Spring, 2007
13 April

Short Answer Questions

Write (on paper) an iterator that iterates over the characters of a C++ string. You should assume that your iterator class lives inside of the string class. That is, you may assume you have a structure like

```
class string {
private:
  vector<char> letters;

public:
  string(...): ... { ... }

  class Iterator : public std::iterator<std::forward_iterator_tag, char > {

  public:
     Iterator(vector<char> *stringVect, long posn = 0) : ... {...}

     ~Iterator() {}

     // The assignment operator
     Iterator& operator=(const Iterator& other) {...}

     bool operator==(const Iterator& other) {...}

     bool operator!=(const Iterator& other) {...}

     Iterator operator++() {...}

     Iterator operator++(int) {...}

     char operator*() {...}

  private:
     ...
  }

  Iterator begin() {...}

  Iterator end() {...}
}
```

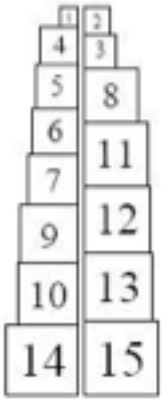
Here we assume that a string is represented as a vector of characters. (You can also use an array of characters if you prefer, though you will also need to keep track of its length.) Please remember that nested classes in C++, unlike those in Java, cannot see the private members of their outer class. Thus the Iterator class cannot get access to the private member `letters` in the string class. Hence it will need to be passed in.

Hand in your solution at the beginning of lab on Tuesday.

Lab Program

The following problem is an example of a very difficult collection of problems known as the *NP-complete* problems. They are all quite time-consuming to solve, even on very fast computers, and this one is no exception.

Suppose that we are given n uniquely sized cubic blocks and that each block has a face area between 1 and n. (In particular, each edge of a block has length $\sqrt{i}$ if its face has area i.) 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 millions 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.

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 values 0 to $n-1$. We can see this by imagining that a coin is tossed n timesonce for each valueand the value is added to the subset if the coin flip shows a head. Since there are $2 \times 2 \times \times 2 = 2^n$ 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 (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 in Java. It varies between C++ implementations, but seems to be 32 bits on the Macs. 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. (Even better would be to use unsigned longs, but we won't bother.) That would work in Java, but in C++ on the Macs, longs are also 32 bits, alas! (But we'll use them anyway!)

- 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 C++ and 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 numbers 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 line of attack is the following:

1. Write a main program that will enter the block edge sizes into a pointer to a vector and then print them out. You may use a constant to define the number of blocks. Define a template to overload "<<" to print out all the elements of a vector. For example if a vector contains the edge sizes of the first 5 blocks then it should print out something like `[1 1.41421 1.73205 2 2.23607 ]`.

The `sqrt` function is available by `#include`ing "math.h".

2. Construct a new class `Subsequences`, which is to represent all of the subsequences (think subsets, but kept in the order they appear in the original sequence) generated from a vector. Originally, you can define this just to hold subsequences built out of doubles, but eventually you will make it into a template.

Strangely, this class will only need two methods, `begin` and `end`, which return iterators. Define a nested class to define class `Iterator`. The idea is that it will use an integer, `setRep` between 0 and $1L << size$, where size is the length of the original vector, to determine which subset should be returned next. The constructor for the iterator should take the vector representing the complete sequence and an long representing where the iteration should start. (The default starting position would be 0, while one past the end would be represented by $1L << size$.)

Make sure that the "*" operator returns the subsequence corresponding to `setRep`, while "++" moves the iterator along to the encoding of the next subsequence. Methods "==" and "!=" should also be written so that you can terminate a for loop.

Write the `begin` and `end` methods of `Subsequences` to return iterators initialized to the first and "one past the end" positions. Add code to the main program to obtain the iterator from `begin()` and print out the vector obtained from it using "*". When that works, write a `for` loop to print out all of the subsequences of the vector of sizes of blocks. (Use a small number of blocks as there are lots of subsequences.) Make sure you get all of the subsets back

3. If all of that works correctly, you should be able to write a program to find the best partition of the set of blocks so that the two towers are as even as possible. Be sure to print out the elements in each tower – the block numbers, not their sizes – (you will likely find it handy to write some auxiliary functions helpful in doing this), and to print out the size of each tower (and the ideal size of the towers if they matched exactly).

My output looks like:

```
The first tower contains [2 4 5 6 10 12 13 14 17 21 22 ]
with height 35.4695, compared with ideal of 35.4695
The second tower contains [1 3 7 8 9 11 15 16 18 19 20 ] with height 35.4695
```

Do not succumb to the temptation to create a vector of all of the subsequences and just output them one at a time with the "*" operator. Even with only 20 blocks, there are over a million subsequences. Keeping them all around is a tremendous waste of space!

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 ./a.out
   ```

   assuming `a.out` is your executable.

   Interestingly, I also wrote this program in Java. Originally, my C++ version ran slower, but after a lot of work trying to avoid creating new objects in the heap – and copying – I got it so my C++ program ran nearly twice as fast as Java. The lesson to me is that with work, one can really improve the performance of C++ programs. With Java, there is not as much that you can do, but it does pretty well without worrying much about efficiency.

   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?

What to hand in

Create and submit a folder that includes the following before midnight on the due date:

1. Your program files.

2. A make file that compiles everything and generates an executable. It should also have a "clean" option so that "make clean" removes all ".o" and executables in the directory.

3. A README file that includes your answers to the three thought questions.

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.

Credits: *This program is adapted from one in Bailey's **Java Structures**, as modified by Steve Freund.*