Eric Roberts
CS106B

Handout #22
January 28, 2013

Section Handout #3—Recursion and Big O

The purpose of this section is to give you some additional practice solving recursive problems and to test your understanding of complexity classes and Big-O notation.

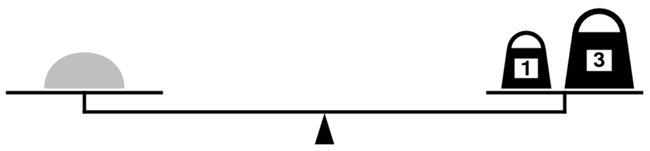
Problem 1. Weights and balances (Chapter 8, exercise 6, page 378)

I am the only child of parents who weighed, measured, and priced everything; for whom what could not be weighed, measured, and priced had no existence.

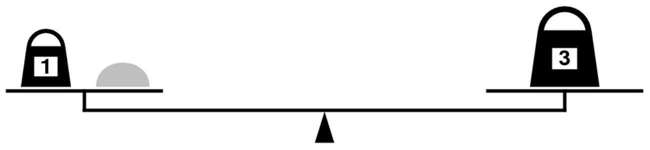
—Charles Dickens, *Little Dorrit,* 1857

In Dickens's time, merchants measured many commodities using weights and a two-pan balance—a practice that continues in many parts of the world today. If you are using a limited set of weights, however, you can only measure certain quantities accurately.

For example, suppose that you have only two weights: a 1-ounce weight and a 3-ounce weight. With these you can easily measure out 4 ounces, as shown:



It is somewhat more interesting to discover that you can also measure out 2 ounces by shifting the 1-ounce weight to the other side, as follows:



Write a recursive function

```
bool isMeasurable(int target, Vector<int> & weights)
```

that determines whether it is possible to measure out the desired target amount with a given set of weights, which is stored in the vector `weights`.

As an example, suppose that the variable `sampleWeights` has been initialized as follows:

```
Vector<int> sampleWeights;
sampleWeights += 1, 3;
```

Given these values, the function call

```
isMeasurable(2, sampleWeights)
```

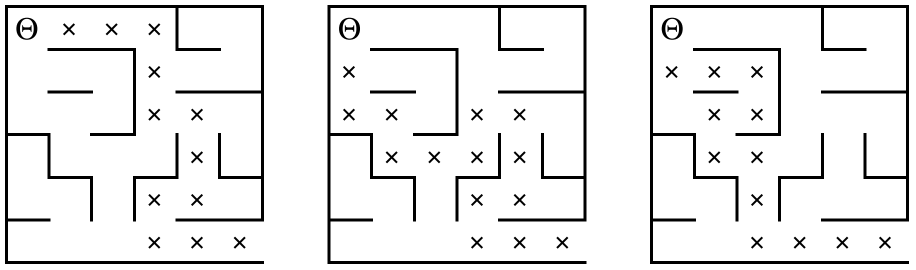
should return **`true`** because it is possible to measure out 2 ounces using the sample weight set as illustrated in the preceding diagram. On the other hand, calling

```
isMeasurable(5, sampleWeights)
```

should return **`false`** because it is impossible to use the 1- and 3-ounce weights to add up to 5 ounces.

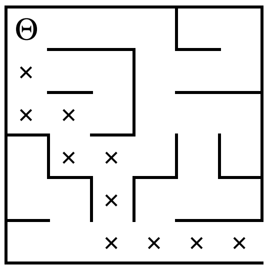
Problem 2. Shortest path (Chapter 9, exercise 1, page 417)

In many mazes, there are multiple paths. For example, the diagrams below show three solutions for the same maze:



length = 13 length = 15 length = 13

None of these solutions, however, is optimal. The shortest path through the maze has a path length of 11:



Write a function

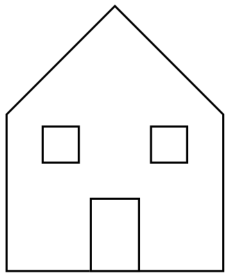
```
int shortestPathLength(Maze & maze, Point start);
```

that returns the length of the shortest path in the maze from the specified position to any exit. If there is no solution, **`shortestPathLength`** should return –1.

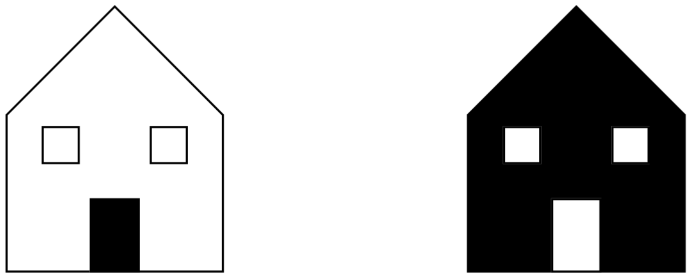
Problem 3. Filling a region (Chapter 9, exercise 4, page 419)

Most drawing programs for personal computers make it possible to fill an enclosed region on the screen with a solid color. Typically, you invoke this operation by selecting a paint-bucket tool and then clicking the mouse, with the cursor somewhere in your drawing. When you do, the paint spreads to every part of the picture it can reach without going through a line.

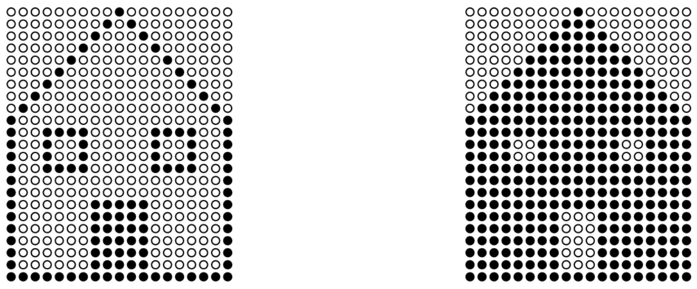
For example, suppose you have just drawn the following picture of a house:



If you select the paint bucket and click inside the door, the drawing program fills the area bounded by the door frame as shown at the left side of the following diagram. If you instead click somewhere on the front wall of the house, the program fills the entire wall space except for the windows and doors, as shown on the right:



In order to understand how this process works, it is important to understand that the screen of the computer is actually broken down into an array of tiny dots called ***pixels***. On a monochrome display, pixels can be either white or black. The paint-fill operation consists of painting black the starting pixel (i.e., the pixel you click while using the paint-bucket tool) along with any pixels connected to that starting point by an unbroken chain of white pixels. Thus, the patterns of pixels on the screen representing the preceding two diagrams would look like this:



It is easy to represent a pixel grid using the type **`Grid<bool>`**. White pixels in the grid have the value **`false`**, and black pixels have the value **`true`**. Given this representation, write a function

```
void fillRegion(Grid<bool> &pixels, int row, int col)
```

that simulates the operation of the paint-bucket tool by painting in black all white pixels reachable from the specified row and column without crossing an existing black pixel.

Problem 4. Generating multiword anagrams

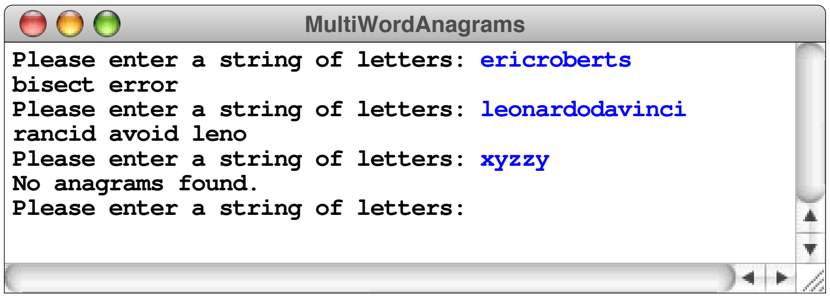
Even before we got to recursion, I showed how it was possible to use the map data structures to find all *anagrams* of a given word, where an anagram is simply a string with the same letters in a different order. Recursion makes it possible to do much more with the anagram idea. In particular, you can write programs that extend the idea of anagrams to multiple words. Dan Brown used several multiword anagrams as clues in his 2003 best-selling thriller *The Da Vinci Code,* so that (if you ignore spaces) you discover that

```
   o draconian devil   →   leonardo da vinci
        oh lame saint   →   the mona lisa
so dark the con of man   →   madonna of the rocks
```

Your job in this problem is to write a function

```
bool findAnagram(string letters, Lexicon & english,
                 Vector<string> & words);
```

that takes a string of letters with all spaces and punctuation removed, the well-worn lexicon of English words, and a vector to hold the words of the generated anagram. As soon as `findAnagram` discovers a series of English words that are an anagram of the specified letters, it should return `true` with those words stored in the vector. If no anagrams exist, `findAnagram` should return `false`. To avoid returning anagrams that consist of boring short words, you should also include a constant in your implementation setting the minimum size of a word. The following sample run was generated with a minimum word length of four:



```
Please enter a string of letters: ericroberts
bisect error
Please enter a string of letters: leonardodavinci
rancid avoid leno
Please enter a string of letters: xyzzy
No anagrams found.
Please enter a string of letters:
```

Problem 5. Recursion and big-O

For each of the following functions, determine its computational complexity expressed in Big-O notation, where N is simply the value of `n` in the first three problems and the length of the string `str` in part (d). Briefly justify your answer. In determining your answers, you may assume that all arithmetic operations run in constant time. In part (d), you should assume that the `length` method runs in constant time and that the `substr` method and the concatenation operator run in linear time with respect to the lengths of their arguments.

(a)

```
int mystery1(int n) {
   int sum = 0;
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < i; j++) {
         sum += i * j;
      }
   }
   return sum;
}
```

(b)

```
int mystery2(int n) {
   int sum = 0;
   for (int i = 0; i < 10; i++) {
      for (int j = 0; j < i; j++) {
         sum += j * n;
      }
   }
   return sum;
}
```

(c)

```
int mystery3(int n) {
   if ( n <= 1 ) return 1;
   return mystery3( n / 2 ) + 1;
}
```

(d)

```
string mystery4(string str) {
   int len = str.length();
   if (len < 2) {
      return str;
   } else {
      string head = str.substr(0, len / 2);
      string tail = str.substr(len / 2);
      return mystery4(tail) + mystery4(head);
   }
}
```

What is the value of `mystery4("PUMPKIN")`?