CpSc 2120: Algorithms and Data Structures

Instructor: Dr. Brian Dean — Fall 2021
Webpage: `http://www.cs.clemson.edu/~bcdean/` — MWF 9:05-9:55
Handout 6: Lab #4 — Powers 208

1 Finding Rotational Symmetry with Hashing

The goal of this lab is to master the technical details of updating a "sliding window hash" in an array – a technique that will be useful for the first homework assignment. We'll do this while tackling a fun geometry problem: counting rotational symmetries of 2D figures. For example, the following fractal snowflake has 6 rotational symmetries, meaning that as the figure is rotated, there will be exactly 6 points in time (including time zero) when you see picture identical to the original during a full 360 degree rotation.

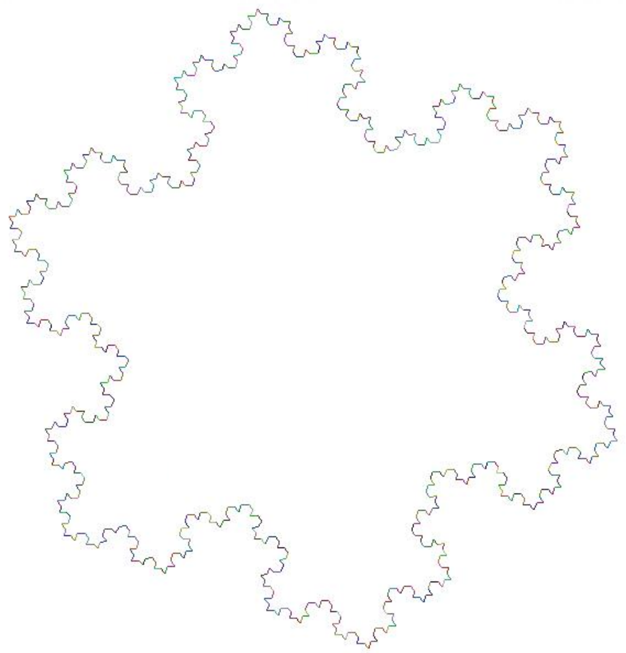

Figure 1: A fractal snowflake with 6 rotational symmetries

In the directory:

`/group/course/cpsc212/f21/lab04`

you will find several sample input files ending in `.txt`. The first line of one of these files is an integer specifying the number of remaining lines. Each of the remaining lines is a drawing command, of which there are three types:

- **F x** : Move forward by x units.
- **L x** : Turn left by x degrees.
- **R x** : Turn right by x degrees.

In the first case x will be a positive integer at most 1 million, and in the second case, x will be a positive integer at most 360. Each input file describes a 2D figure that is "closed" in the sense that the pen ends at the same point where it originally started.

You can use the `visualize` command in the lab directory to display any of these shapes on the lab machines; for example:

```
./visualize < snowflake.txt
```

The 'q' key exits the visualizer.

2 First Task: Storing the Input in an Array of Integers

If the input has k lines, you should represent it with an integer array of length $2k$. Each line supplies two of these integers; you can translate 'F', 'L', and 'R' to 0, 1, and 2. So for example, if the input contains

```
L 90
F 3
R 90
```

then your integer array would contain respectively 1, 90, 0, 3, 2, 90. Since the 2D figure ends at the same point as it starts, you should think of your array as being "circular", logically wrapping back around to the beginning once you reach the end.

3 Second Task: Counting Symmetries the Slow Way

Suppose you find that the entire array starting from index 0 is identical to the entire array if you start at some other index i (wrapping back around when you reach the end). In this case, you have found a rotational symmetry, since your figure would look the same if you started drawing it from position 0 as from position i.

To count the total number of symmetries, you could either look for the number of indices i that are matches as above. Or, even better, you could find the first such index i and divide the array length by i, since the matching indices will occur evenly spaced throughout the array.

Please write code that counts the total number of symmetries in the way we just described; that is,

```
For every index i = 1, 2, 3, ...
   Determine if the entire array starting at index 0
   matches the entire array starting at index i (wrapping around
   as appropriate).
   Stop at the first such index i that is a match.
Output the array length divided by i
```

Note that this algorithm runs in $\Theta(n^2)$ worst-case time. For example, it should run quite slowly on the `stick.txt` input, which is designed to be a bad input for this slower approach.

4 Third Task: Counting Symmetries Faster with Hashing

We can make our algorithm faster as follows:

```
Let H be a polynomial hash of the entire array, starting at index 0
For every index i = 1, 2, 3, ...
   Let H' be the updated hash for the entire array, starting at index i
   If H == H', then check to see if there is actually a match at index i.
   Stop at the first such index i that is a match.
Output the array length divided by i
```

As we discussed in class, this should run quickly since we can update the hash in only $O(1)$ time per step, and we only need to spend time checking for a match in the event of a hash collision, which as a false positive should be rare — recall from class that the probability of any two different length-n arrays colliding, using polynomial hashing modulo a prime p, is at most n/p.

Suppose our array is $A[0 \ldots n-1]$, and let p be a prime number. We'll use $p = 1,000,000,007$ here, since this is a large prime but it still allows two integers in the range $\{0, 1, 2, \ldots, p-1\}$ to be added together without overflowing an int. Let x be some number in $\{0, 1, 2, \ldots, p-1\}$. The hash of our initial array, starting from index 0, is given by

$$H = (A[0]x^{n-1} + A[1]x^{n-2} + \ldots + A[n-2]x + A[n-1]) \,\%\, p.$$

Recall that we can easily compute this with Horner's rule:

```
int H = 0
For i=0, 1, 2, ..., n-1
  H = ((long long)H * x + A[i]) % p
```

The reason for the cast to a long long in the code above is to avoid overflow. In general, for this particular hash function to behave the way we expect mathematically, we need to avoid overflow or underflow during all of our hash calculations. We can do this by the following rules:

- Any time we add two numbers x and y that are both in the range $\{0, 1, \ldots, p\}$, we reduce the result modulo p: `result = (x + y) % p`. There is no overflow since $x + y$ still fits into an int.

- Any time we subtract two numbers x and y that are both in the range $\{0, 1, \ldots, p\}$, we take `result = (p + x - y) % p` to prevent underflow. Note that $p + x - y$ still fits into an int, so it won't overflow.

- Any time we multiply two numbers x and y that are both in the range $\{0, 1, \ldots, p\}$, this could certainly overflow an int, so we first cast one of them to a long long (a 64-bit int) so the result is performed with 64-bit arithmetic and therefore will not overflow. After reducing the result modulo p, it can then fit back into an int: `result = ((long long)x * y) % p`

Now that we know the hash H of our entire array starting from some index i, how do we update H to a new value H' reflecting the hash at the next index $i+1$? As described in class, we can do this mathematically by subtracting off the term that leaves the hash window, scaling up everything by x, and adding one new term. Here, since our array is cyclic and we are hashing the entire array, the exiting element and the entering element are both $A[i]$. In the resulting formula,

$$H' = ((H - A[i]x^{n-1})x + A[i]) \,\%\, p,$$

we must use care in each of the operations (addition, subtraction, multiplication) to avoid underflow and overflow (so in code, the formula will look slightly more involved than above). We should also pre-compute $x^{n-1} \,\%\, p$ so we have this value at the ready to plug in to the formula above.

5 What to Submit

You should submit two programs, `slow.cpp` and `fast.cpp`. They should both produce the same output, but the slow program should use the slower (non-hashing) approach for counting symmetries while the faster program should use hashing. The output should be the following:

```
./a.out < square.txt
4
./a.out < stick.txt
2
./a.out < angry_triangle.txt
3
./a.out < snowflake.txt
6
```

For full credit, your `fast.cpp` code should run in less than 1 second on each of these cases.

This lab is due by 11:59pm on Sunday, September 19. No late submissions will be accepted.