CS61B Lecture 2

Friday, January 24, 2020

Announcements

- Please make sure you have obtained a Unix account.
- Lab #1 is due Wednesday (end of Wednesday at midnight). Usually, labs are due Friday midnight of the week they occur. It is especially important to set up your central repository.
- If you decide not to take this course after all, please tell CalCentral ASAP, so that we can adjust the waiting list accordingly.
- HW #0 will be up this evening, due next Friday at midnight. While you get credit for any submission, we strongly suggest that you give the problems a serious try.
- We strongly discourage taking this course P/NP (or S/U).

Let's write a program

To understand Java, let's write a sample program called `Primes` that takes in a value `U`, which prints every prime number through to `U`. For example:

```
Input: java Primes 101
Output: 2 3 5 7 11 13 17 19 23 29 31 37 41 43
        47 53 59 61 67 71 73 79 83 89 97 101
```

Useful facts:

- prime number is an integer greater than 1 that has no divisors smaller than itself other than 1. (Alternatively: p > 1 is prime if and only if (iff) `gcd(p, x)` = 1 for all 0 < x < p.)
- $k \leq \sqrt{N}$ iff $N/k >= \sqrt N$, for $N, k > 0$
- If k divides N, then N/k divides N.

Plan

There are various ways of writing a program. One of them is called **top-down**, where we look at the program as a whole, figure out some pieces into which it may be broken (nice, clean pieces that don't interact with each other too much), and recursively solve each piece.

Our program is going to be in a class called `Primes`, which has a `main` method, a `printPrimes` method which prints the primes up t a `limit` and returns `void`, and an `isPrime` method that figures out if a number is prime.

`isPrime`: Testing for primes

Let's start with the simplest part of the code, which is the `isPrime` method:

```
private static boolean isPrime(int x) {
    if (x <= 1) {
        return false;
    } else {
        return !isDivisible(x, 2); // "!" means "not"
    }
}
```

This part of the code seems reasonable enough, but well, we now need to write the `isDivisible` function (which is an abstraction), as we learnt in 61A.

```
/** True iff X is divisible by any positive number >=K and < X,
 *  given K > 1. */
private static boolean isDivisible(int x, int k) {
    if (k >= x) { // a "guard"
        return false;
    } else if (x % k == 0) { // "%" means "remainder"
        return true;
    } else { // if (k < x && x % k != 0)
        return isDivisible(x, k+1);
    }
}
```

Thinking recursively

Understand and check `isDivisible(13,2)` by tracing one level. You don't trace recursive calls down to the very bottom, just one level is enough. Let's see what we mean:

- Call assigns `x=13`, `k=2`
- Body has form `if (k >= x) S1 else S2`.
- Since $2 < 13$, we evaluate the first `else`.
- Check if 13 mod $2 = 0$; it's not.
- Left with `isDivisible(13,3)`.
- Rather than tracing it, instead use the comment:
- Since 13 is not divisible by any integer in the range 3,12 (and 3 > 1), `isDivisible(13,3)` must be false, and we're done!
- Sounds like that last step begs the question. Why doesn't it?

The lesson here is that comments aid understanding. Make them count!

Iteration

`isDivisible` is tail recursive, and so instead of thinking recursively, you can think of it as an iterative process that creates the body of the recursion.

Usually, an "Algol family" production language have special syntax for iteration. Here are four equivalent versions of `isDivisible`:

> **Algol**
> Algol was a programming language is the late 50s and early 60s, from which Java is derived from, regarded by many as the first modern programming language.

```
/* Version 1 */
if (k >= x) {
    return false;
```

```
} else if (x % k == 0) {
    return true;
} else {
    return isDivisible(x, k+1)
}

/* Version 2 */
while (k < x) { // !(k >= x)
    if (x % k == 0) {
        return true;
    }
    k = k+1; // or k += 1, or (yuch) k++
    }
return false;

/* Version 3*/
int k1 = k;
while (k1 < x) {
    if (x % k1 == 0){
        return true;
    }
    k1 += 1 ;
}
return false;

/* Version 4*/
for ( int k1 = k ; k1 < x ; k1 += 1 ) {
    if (x % k1 == 0) {
        return true;
    }
}
return false;
```

Using Facts About Primes

We haven't used the Useful Facts from an earlier slide. Since we only have to check for divisors up to the square root, let's reimplement the iterative version of `isDivisible` to make it more efficient:

```
/** True iff X is divisible by some number >=K and < X,
 *  given that K > 1, and that X is not divisible by
 *  any number > 1 and < K. */
private static boolean isDivisible (int x, int k){

    // !! NOTICE !!
    int limit = (int) Math.round(Math.sqrt(x));
    // !! NOTICE !!

    for (int k1 = k; k1 <= limit; k1 += 1){
        if (x % k1 == 0){
            return true;
        }
        return false;
    }
```

This makes the program run in square root N time instead of linear time, which makes it considerably faster, especially when checking large numbers of primes.

> **Cautionary Aside: Floating Point**
> See the notice on the previous piece of code. Why is that there? Well, floating point operations yield to the corresponding mathematical operations, you might ask the following about `(int) Math.round(Math.sqrt(x))`:
>
> - Is it always at least $⌊\sqrt{x}⌋$? ($⌊z⌋$ means "the largest integer $≤ z$.") If not, we might miss testing $\sqrt{x} when {x} is a perfect square.
>
> As it happens, the answer is "yes" for IEEE floating-point square roots. This is just an example of the sort of detail that must be checked in edge cases.

`printPrimes`: Simplified solution

Here is a very quick overview of `printPrimes`:

```
/** Print all primes up to and including LIMIT. */
private static void printPrimes(int limit){
    for (int p = 2; p <= limit; p += 1){
        if (isPrime(p)){
            System.out.print(p + " ");
        }
    }
System.out.println();
}
```

There is just one problem with this: it doesn't neatly create line breaks. What if we asked it to print up to 8589935681?

Let's create a neat solution that automatically wraps every 10 numbers (it's not perfect, but oh well):

```
private static void printPrimes(int limit){
    int np;
    np = 0;
    for (int p = 2; p <= limit; p += 1){
        if (isPrime(p)){
            System.out.print(p + " ");
            np += 1;
            if (np % 10 == 0){
                System.out.println();
            }
        }
    }
    if (np % 10 != 0){
        System.out.println();
    }
```

And there's our program! We have just written our first Java program with the top-down method.