3 Instruction – Arrays

Preliminaries

Remember:

- Start every file you write with a line of comment with your name(s) and the date.
- Work in pairs, if possible.
- Keep your code neat from the beginning on, with proper indentation, etc.
- Have each exercise checked by one of your instructors.

Furthermore:

- Use names that start in lowercase for variables.
- Use all uppercase for names of variables that do not change after initialisation (constants).
- Use assertions where appropriate.

3.1 Square root

You are going to write a program that computes the *integer square root* of an integer number.

1. Write a program that reads, after a prompt, a non-negative integer from input, stores it in N and then computes the integer x that is the square root of N, rounded down to the nearest integer. In other words:

   $x^2 \leq N < (x+1)^2$

   Do this by starting from 0 and increasing x until you find the right value. Don't use the built-in square root function of Java. Use $x * x$ to compute x^2. At the end, print the values of N, x, and x^2.

2. The program above is not as efficient as possible. It tries all values from 0 to the square root. We call this a *linear search* approach. Try to find a better solution, one that tries fewer values.

3.2 Sieve of Eratosthenes

A prime number is an integral number that has no divisors greater than 1 and smaller than itself. We want to enumerate the first prime numbers up to some number N. Testing each number separately for primeness is not optimal in efficiency.

You are going to write a program that enumerates prime numbers more efficiently, using an ancient algorithm, the *sieve of Eratosthenes*.

The idea is that you make a list of all numbers from 2 to N and then strike from the list all numbers that are multiples of 2, then all that are multiples of 3, etc. In the end, only the prime numbers remain.

1. Declare a constant integer N. Give N not too high a value, e.g., 25. Declare and create a boolean array `sieve` with highest index N. This array corresponds to the list of numbers in the description above. The members with index 0 and 1 are not used. The intended meaning is that `sieve[m]` is `true` if and only if m has divisors between 1 and m. Initially, all members are `false`.

2. Striking a number from the list is implemented by setting the value of this index in the array `sieve` to `true`. Perform this in a nested loop that sets all multiples of 2 to `true` and repeats this for the following numbers. Print the resulting prime numbers, i.e., the indices of the array that have value `false` (ignore indices 0 and 1).

3. Think about the following questions.

   - When do you stop, i.e., what is the highest number of which multiples should be considered?
   - Do you need to process all numbers between 2 and this maximum?

4. Instead of using a fixed value, read the value of N from input.

3.3 Collatz

A *Collatz sequence* is a sequence of integer numbers $a_0, a_1, a_2, \ldots$ that is obtained as follows.

1. start with an arbitrary positive number a_0;
2. if a_n is even, then $a_{n+1} = a_n/2$;
3. if a_n is odd and $a_n > 1$, then $a_{n+1} = 3a_n + 1$;
4. if $a_n = 1$, the sequence ends with a_n.

Example: 12, 6, 3, 10, 5, 16, 8, 4, 2, 1.

In 1937, the German mathematician Lothar Collatz proposed the conjecture (*vermoeden*) that these sequences are all finite. Since then, it has been shown that for many starting numbers the sequence ends, but no proof of the conjecture has been found yet. Other names are *Ulam conjecture* and *Syracuse problem*. The sequence is sometimes called the *hailstone sequence* (from the movement of hailstones while they are formed, I guess).

1. Write a program that reads from input a positive number and then prints the Collatz sequence starting with that number, followed by the length of the sequence and the maximum number reached.

2. Make this into a game by asking two starting numbers and then printing side by side the sequences. Use a tab character to separate them (`"\t"`). Example:

```
16      8
8       4
4       2
```

```
2     1
1
Player 1 wins with 5 to 4.
```

3. Make the game more interesting by specifying an interval from which the starting values have to be chosen. The interval is specified with lower and upper bound (maybe it's fair to have one player enter the lower bound and the other one the upper bound). When a player enters a number outside the interval, he has to repeat his entry.

4. Forbid entering values that were previously used. So remember all the starting numbers in an `ArrayList<Integer>`. Search through the ArrayList to see whether the number has been entered already. Do not use the built-in functionality of ArrayList (such as *contains* or *indexOf*.

5. Try to find a better winning criterion than the length of the sequence. Is the length divided by the starting number a good criterion, or does it favour low numbers too much? Maybe dividing by the log of the starting number is a good one?

6. * If you keep the ArrayList sorted, the search method can be more efficient. Implement this.

Kees Huizing – contact2ip65@gmail.com – 10 Sep, 2010