UVa 100: The 3n + 1 problem

Written by Bjarki Ágúst on 18 April 2012

Topics: UVa Online Judge

The $3n + 1$ problem (or UVa 100) is the first problem at UVa's online judge. The problems there are not in any particular order, as can be seen from this first problem which is far from being the easiest (but also very far from being the hardest). You can read the problem statement here or here.

Interpretation

This problem is very similar to Project Euler 14. We are dealing with Collatz sequences (also known as Hailstone numbers), and are asked to find the length of the longest Collatz sequence starting at any number between i and j (inclusive), where i and j are numbers that are given to us as input.

We must be very careful when reading the problem statement. We need to gather as much knowledge as possible about the input and be careful not to assume anything that is not specifically stated. Here we are given that i and j are both integers between 0 and $1.000.000$ (exclusive). Now many people could wrongly assume that i is less than j (because that would be the most natural way for them to be given to us), but it's never said that i would be less than j, and therefore we cannot assume that. A solution that would assume that would get a Wrong Answer verdict from the online judge.

Brute Force

Just like we did in Project Euler 14, we are going to create a brute-force solution to solve the problem. But first of all, since we're dealing with Collatz sequences, we should create a helper function to calculate the number that comes after a given number in the sequence. It's pretty trivial and should look something like:

```
// a function that returns the
// next number in the sequence
public static long next(long n) {
        if (n % 2 == 0)
                return n / 2;       // if n is even
        else
                return 3 * n + 1;   // if n is odd
}
```

Next we need to calculate a given numbers cycle length. We can either do it iteratively (which is probably faster) or we can do it recursively (which might cause stack overflow, but ends up being more beautiful code). I'm going to go with the recursive implementation because of what we are going to do in the next section. 1 will be our base case with a length of 1, and our recursive case will be that the cycle length of the current number is one greater than the cycle length of the next number.

```
public static int cycleLength(long n) {
        // our base case
        // 1 has a cycle length of 1
        if (n == 1)
                return 1;

        // the cycle length of the current number is 1 greater
        // than the cycle length of the next number
        int length = 1 + cycleLength(next(n));

        return length;
}
```

The last thing we have to do to get a working solution is to implement the main method. We need a while-loop that reads the test cases and then we need to loop through all the numbers from i to $j$and find the minimum cycle length. Then we must output the result in exactly the same format as the problem statement tells us to (even the smallest extra whitespace could cause an otherwise working solution to fail), which in this case tells us to output i, j and the minimum cycle length on one line, with one space between them, in this order. Remember that i doesn't have to be smaller than j, so we have to take special care when looping through the numbers (that is, we have to loop from $min(i, j)$ to $max(i, j)$ if we're using a conventional for-loop).

```java
public static void main(String[] args) throws Exception {
	Scanner in = new Scanner(System.in);
	PrintWriter out = new PrintWriter(System.out, true);

	// while there is some input to read
	while (in.hasNextInt()) {
		int i = in.nextInt(),
			j = in.nextInt(),
			from = Math.min(i, j),
			to = Math.max(i, j),
			max = 0;

		// loop through all the numbers
		// and find the biggest cycle
		for (int n = from; n <= to; n++) {
			max = Math.max(max, cycleLength(n));
		}

		out.printf("%d %d %d\n", i, j, max);
	}
}
```

Now our solution is complete. Well, almost. Our solution might be too slow, since it has to solve all allowed inputs in under 3 seconds. We're not sure, but we don't want to take any unnecessary risk, so we might as well optimize it a bit. Which brings us to the next section.

Caching

We calculate the cycle length for some of the numbers again and again and again. Therefore it's a good idea to save (aka. cache) what we've computed, so we don't have to compute it again. Here we are going to have an array for our cache, where the n-th element is the cycle length of the number n. Note that a cycle length of 0 for the n-th element means that we haven't calculated it yet. I've decided to only cache the results for n smaller than $1.000.000$, since that is the range we are mostly dealing with in this problem (take a look at the post and comments of Project Euler 14 for some discussion about this).

We only need to make small changes to our cycleLength function to cache the results, since we went with the recursive approach.

```java
// cache for already computed cycle lengths
public static int[] cache = new int[1000000];

public static int cycleLength(long n) {
	// our base case
	// 1 has a cycle length of 1
	if (n == 1)
		return 1;

	// check if we've already cached the
	// cycle length of the current number
```

```
    if (n < 1000000 && cache[(int)n] != 0)
        return cache[(int)n];

    // the cycle length of the current number is 1 greater
    // than the cycle length of the next number
    int length = 1 + cycleLength(next(n));

    // we cache the result if the
    // current number is not too big
    if (n < 1000000)
        cache[(int)n] = length;

    return length;
}
```

Conclusion

We started off by writing a straightforward recursive solution to our problem. We then optimized it by caching cycle lengths that we had already computed and now our solution is fast enough to get Accepted at the online judge.

You can take a look at the complete source code here.

So what do you think? Did/would you solve it differently? Let me know in the comments.