CS1010 Laboratory 04

Testing, Past Year PE Papers

Zhang Puyu

Group BD04

September 19, 2023

Plan of the Day

Importance of Testing

A simple logical expression:

$$\text{My code is correct} \implies \text{My code can pass all test cases.}$$

Caution: The *converse* of the above is **NOT** true in general!

Example: `divide.c` from Exercise 1.

How to Design Test Cases

1. It is **impossible** to *enumerate all* test cases! (In fact possible if there are certain constraints imposed, but you can never do that with a reasonable amount of time.)
2. But it is **possible** to divide the test cases into a few groups, i.e., all inputs must fall into a few *types/scenarios*.
3. We can choose to test for all possible **classes of inputs/outputs**, or
4. Test for the **boundary cases** according to the constraints given.

Test Case Design: Example

Task: Check if a **positive integer** is **prime**.

Input restriction: Positive integers.

Catogorise possible inputs: Either *prime* or *composite*.

Edge cases:

- 1: The smallest positive integer AND the only integer which is neither prime nor composite.
- 2: The smallest prime AND the only even prime.
- 3: The smallest odd prime.
- 4: The smallest composite number AND the smallest even composite number.

Test Case Design: Ensure the Following

- Conditionals: Your test cases should cover **all** conditional branches (i.e., for all conditional branch, there exists at least one test case which can enter that branch).
- Loops: Your test cases should contain at least one case for 0 iteration and at least one case for k iterations where $k \geq 1$.
- You should always have test cases for the boundary conditions.

Test Case Design: Another Example


```
bool is_prime(long n) {
  // bad code for demo only
  if (n % 2 == 0 || n % 3 == 0) {
    return false;
  }
  long factor = 4;
  while (factor <= n) {
    if (n % factor == 0) {
      return false;
    }
    factor += 1;
  }
  return true;
}
```

Test cases:

1. 4: `true`-block at Line 2.
2. 6: `true`-block at Line 2.
3. 1: 0 iteration.
4. 5: Execute the loop.
5. 8: `n % factor == 0` is true.
6. 7: `n % factor == 0` is always false.

Test Case Design: Try This!


```
long find_least_significant_9(long n) {
  long count = 1;
  while (n != 0) {
    if ((n % 10) == 9) {
      return count;
    }
    n /= 10;
    count += 1;
  }
  return 0;
}
```

Testing: Tips

1. Test each function separately (print out the value to be returned and compare it with your expectation!).
2. Decompose your program into shorter and simpler functions: simpler code is easier to test!

Exercise 2 Review

Main issues are concentrated in `prime.c` and `factor.c`.

- IMPORTANT: Always **analyse** the constraints of inputs and outputs before you code! (Crucial to rule out unnecessary operations.)
- Read the question carefully. `collatz.c` for example asks for the integer **in the range** $\{m, m+1, m+2, \cdots, n\}$ with the largest stopping time.
- **Please** make use of your *mathematical knowlegde* to optimise your algorithms. E.g., at least use $\frac{n}{2}$ to bound your loop in `factor.c`, and iterate backwards from n to find the biggest prime.

Exercise 2 Review: An Illustration Using `factor.c`

- You should have convinced your self that $n - 1$ is far beyond what is necessary as an upper bound for the factors of n.
- First attempt to optimise: $n - 1 \rightarrow \frac{n}{2}$ — anything bigger than $\frac{n}{2}$ for sure cannot divide n!
- Try to prove the following to yourself: if i divides n, then surely $\frac{n}{i}$ divides n as well.
- The above implies that we can count factors in **pairs**!
- Note that: if i divides n and $i < \sqrt{n}$, then $\frac{n}{i} > \sqrt{n}$.
- So the *supremum* is $\sqrt{n}$.
- We should then deal with the special case where n is a perfect square (a perfect square has an odd number of factors while any other integer has an even number of factors, if you haven't noticed).

Exercise 2 Review: `collatz.c`

- Notice that the Collatz operations have been clearly defined to us:

$$c_{n+1} = \begin{cases} \frac{c_n}{2}, & \text{if } c_n \text{ is even} \\ 3c_n + 1, & \text{otherwise} \end{cases}$$

- If we write this as a function `collatz` that takes in an integer and apply one step of the Collatz operation, we can **repeat it in a loop** until the input integer is reduced to 1.
- In each iteration, we can increment the count of steps by 1. This way, after the loop is completed, we will have the total number of steps required for the integer to complete the Collatz processes.
- Then, we will repeat the above **in another loop** for integers between m and n, keeping track of `max_steps` and `max_num`.

Exercise 2 Review: Critique This `collatz.c`


```
void total_stop_t(long m, long n) {
    long max_count = -1;
    long max_num = m - 1;
    for (long i = m; i <= n; i++) {
        long next_step_count = 0;
        while (i != 1) {
            if (i % 2 == 0) {
                i /= 2;
            } else {
                i = i * 3 + 1;
            }
            next_step_count += 1;
        }
        if (next_step_count > max_count ||
            (next_step_count == max_count && i > max_num)) {
            max_count = next_step_count;
            max_num = i;
        }
    }
    cs1010_println_long(max_count);
    cs1010_println_long(max_num);
}
```

Exercise 2 Review: before `pattern.c`

Several things to take note of for your prime checker:

- `sqrt` is an **expensive** operation. Do it once only and store the value as a local variable.
- Bound your loop for checking primes by $\sqrt{n}$ instead of $\frac{n}{2}$. (WHY?)
- To find the biggest prime less than n, iterate from n backwards (way faster).

Exercise 2 Review: `pattern.c`

We will demonstrate how to do **reverse engineering** when designing a program.

- Let C_{ij} be the j-th cell in the i-th row. We wish to find whether it contains **at least** (IMPORTANT) one prime.
- If t_{ijk} is the k-th number in C_{ij}, we can **loop** through all the t's from $k = 1$ to $k = m$ until we encounter the **first** prime. (WHY?)
- How to find t_{ijk}? Well, C_{ij} is just an A.P. (or lemon-run) with m terms and a common difference of i, so it suffices to find t_{ij1} i.e. the **first term** of C_{ij}.
- How to find t_{ij1}? Well, $t_{ij1} = t_{i(j-1)1} + 1 = t_{i11} + (j-1)$, so it suffices to find t_{i11}, i.e., the **first term** of the **first cell** in the i-th row.
- How to find t_{i11}? Well, this may take some time as you observe the pattern (but very fast if you have the mathematical instinct). You will realise that

$$t_{i11} = \frac{i(i-1)}{2}m + 1.$$

- Now, all you need to do is to build your algorithm in reverse order to the above reasoning.

`simple.c` (AY22/23)

- Suppose the given number is

$$a_n^* a_{n-1}^* \cdots a_1^*$$

where a_i^* denotes a sequence of repeated occurrence of a_i. The question can be decomposed into 2 steps:

1. retrieve each of the a_i's,
2. build the digits into the number $a_n a_{n-1} \cdots a_1$.

- **Retrieving a_i:** Loop through the digits of the input integer from right to left. For every different integer seen, record it in a variable `curr_digit`.
- **Collecting a_i into the result:** Append a_i to the beginning of a partially-built result $a_{i-1} a_{i-2} \cdots a_1$. (HOW?)
- What should the result be initialised to?

root.c (AY22/23)

- Finding a and r should be easy (hopefully).
- How to unpack the infinite fraction? (Need some observation or mathematical intuition.)
- Notice that

$$\frac{b_k}{c_k} = \frac{r}{2a + \frac{r}{2a + \frac{r}{2a + \cdots}}} = \frac{r}{2a + \frac{b_{k-1}}{c_{k-1}}}.$$

  And now we have a **recurrence relation** on b_k and c_k.
- What is the base case when $k = 1$? $\frac{b_1}{c_1} = \frac{r}{2a}$.
- Now we only need to write a function to compute $\frac{b_k}{c_k}$ for any integer $k \geq 1$ recursively (or iteratively).

largest.c (AY22/23)

- An iterative approach is trivial (find the maximum digit repeatedly) but sadly you have to solve this in recursion :(
- We are given $a_1 a_2 a_3 \cdots a_n$.
- Suppose we can find an arrangement for $a_1 a_2 a_3 \cdots a_{n-1}$, where should a_n be placed to produce the biggest n-digit number?
- For those of you who have learnt in advance or have programming background: This is essentially **insertion sort**...
- Note that we need n layers of recursion and the k-th layer in the worst case needs to traverse all k digits, so in total we have $\frac{n(n+1)}{2}$ steps.
- VERY SLOW in fact. Challenge yourself to beat this speed :)

Fun Stuff: A Little Extension on `prime.c`

- Note that in `prime.c`, although in the theoretical worst case it takes about n iterations to find the biggest prime less than or equal to n, in practice this largest prime won't be too far away from n.
- But what if we were to find **all primes** less than or equal to n? In this case, we have to iterate **every** integer from n to 2 (that's $\frac{n}{2}$ steps even if we exclude all even integers bigger than 2). Checking if each integer is prime takes another $\sqrt{n}$ steps in the worst case, so in total there are about $\frac{n\sqrt{n}}{2}$ steps — slow!
- However, using an **array**, we can speed up the above process!

Fun Stuff: Sieve of Eratosthenes (I have difficulty pronouncing this name)

Consider all integers from 2 to n:

$$2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, \cdots, n$$

Take 2 (which happens to be the smallest prime), and cross out all multiples of 2 in the list.

Take the next un-crossed integer (which is 3 in this case) and cross out all multiples of 3.

Iterate through the list while repeating the above process. We claim that: **each time we come across an un-crossed integer i, it will be a prime!** (WHY?)

Stop when we encounter **the first integer that is bigger than $\sqrt{n}$**. We claim that from this integer onwards, **all un-crossed integers are prime!** (WHY?)

Number of operations in the worst case: upper-bounded by $n \log(\log n)$ (fewer than $n\sqrt{n}$).