COL100: Introduction to Computer Science

5.2: How to design programs

Where to begin?

- Write down a specification of the desired function.

Example: Find the number of primes between *a* and *b* (both inclusive).

$$countPrimes : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

Given natural numbers *a* and *b*, *countPrimes*(*a*, *b*) gives the number of primes in the set $\{a, a + 1, \ldots, b\}$.

- Design a high-level outline of the algorithm

To count the number of primes between *a* and *b*, we need to check if *a* is prime, and count the number of primes between $a + 1$ and b.

- Write down the specification of any helper functions you need

$$isPrime : \mathbb{N} \rightarrow \mathbb{B}$$

Given a natural number *n*, *isPrime*(*n*) is true if and only if *n* is a prime number.

- Formalize the algorithm using the helper functions

Start by identifying the trivial cases, e.g. $n = 0$ for power, $a = b$ for gcd

$$countPrimes(a, b) = 0 \text{ if } a > b.$$

For the nontrivial case, break it down into simpler problems, and/or smaller version(s) of the same problem.

Choose a breakdown carefully, so the desired solution can be found with a small amount of work!

$$countPrimes(a, b) = 1 + countPrimes(a + 1, b) \text{ if } isPrime(a),$$
$$countPrimes(a, b) = countPrimes(a + 1, b) \text{ otherwise.}$$

- Repeat the process for the helper functions you needed

$$isPrime : \mathbb{N} \rightarrow \mathbb{B}$$

Specification: Given a natural number $n > 1$, $isPrime(n)$ is *true* if and only if n is a prime number.

Outline: To check if n is prime, find its smallest divisor greater than 1 and check if it is equal to n.

Helper functions: $smallestDivisor : \mathbb{N} \rightarrow \mathbb{N}$
$smallestDivisor(n)$ is the smallest number $i > 1$ such that $n \bmod i = 0$.

Algorithm: $isPrime(n) = (n = smallestDivisor(n))$

- Repeat… (recursively!)

$$smallestDivisor : \mathbb{N} \to \mathbb{N}$$

$$smallestDivisorAbove : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

…

(or define *smallestDivisor* using *findSmallest* from earlier)

- Stop when no more helper functions are left to be defined

- Prove correctness of all functions involved

To prove correctness of *countPrimes*, we have to show that

1. if *isPrime* is correct (i.e. matches its specification) then *countPrimes* is also correct, and
2. *isPrime* is correct.

To prove correctness of *isPrime*, we…

- Translate your algorithm into a programming language

  If the algorithm is fully specified, implementation is easy!
  If you’re not sure about the algorithm, you’ll struggle write the code.

```
fun smallestDivisorAbove(n, k) = …;
fun smallestDivisor(n) = smallestDivisorAbove(n, 1);
fun isPrime(n) = (n = smallestDivisor(n));
fun countPrimes(a, b) =
  if a > b
  then 0
  else if isPrime(a)
  then 1 + countPrimes(a + 1, b)
  else countPrimes(a + 1, b);
```

Summary

- Write down a specification of the desired function.
- Design a high-level outline of the algorithm
- Write down the specification of any helper functions you need
- Formalize the algorithm using the helper functions
- Repeat the process (recursively!) for the helper functions, until no more helper functions are left to be defined
- Prove correctness of all functions involved
- Translate your algorithm into a programming language

Testing

“Beware of bugs in the above code;
I have only proved it correct, not tried it.”

—Donald E. Knuth

- In the specification of each function, try to include test cases (known inputs and outputs) covering both trivial and non-trivial cases

countPrimes(4, 2) = 0
countPrimes(10, 20) = 4

- After implementing, try function on those test cases

Note: Testing is not a substitute for a correctness proof! (Why?)

Afterwards

- Design recursive algorithms for the following:
    - Reversing the digits of a positive integer in base 10, e.g. *reverseInt*(123) = 321.
    - Computing a div b using only addition and subtraction.
    - Checking if two natural numbers are *amicable*, i.e. their proper divisors add up to each other. E.g. 220 and 284 are amicable because 1 + 2 + 4 + 5 + 10 + 11 + 20 + 22 + 44 + 55 + 110 = 284 and 1 + 2 + 4 + 71 + 142 = 220.
    - Counting the number of ways to make change for Rs. n, given an infinite amount of coins/notes of Rs. 1, 2, 5, 10, 20, 50. Assume that a function $d(k)$ gives the denominations, i.e. $d(0) = 1$, $d(1) = 2$, $d(2) = 5$, …