# Chapter Six

# Algorithms

1

## Algorithms

- An *algorithm* is an *abstract strategy* for solving a problem and is often expressed in English
- A *function* is the *concrete realization* of an algorithm in the context of a programming language
- There exist *more than one algorithm* for most problems

2

## Testing for Primality

- A positive integer *n* is a *prime* if it has exactly two positive divisors, which are itself and 1
- Prime numbers are important today because they play a central role in many forms of *cryptography*

3

## A Simple Algorithm

- Check each number between 1 and *n* to see whether it divides evenly into *n*
- Add 1 to a counter each time you encounter a new divisor
- Check to see whether the counter is 2 after all numbers have been tested

4

## A Simple Function

```
bool isPrime(int n)
{
   int  divisors, i;

   divisors = 0;
   for (i = 1; i <= n; i++) {
      if (n % i == 0) divisors++;
   }
   return (divisors == 2);
}
```

5

## Verification of Algorithms

- An *algorithm* must satisfy the following three requirements
- *Clear* and *unambiguous* in its definition
- *Effective*, in the sense that its steps are executable
- *Finite*, in the sense that it terminates after a bounded number of steps

6

## Verification of Algorithms

- Is the definition of isPrime *clear* and *unambiguous*
- Are the steps in isPrim *effective* in the sense that it is possible to carry them out
- Does isPrime terminates after a *finite* amount of time

7

## Correctness of Algorithms

- Proving that an algorithm is *correct* is very hard to do in any *formal* way
- A property that is true at the entry to a loop and that continues to be true at the end of each loop cycle is called a *loop invariant*
- Proving *loop invariants* are maintained is an important tool for proving *correctness*
- *Testing* is an approach to reveal the presence of errors in the algorithms, but never their absence

8

## Correctness of Algorithms

- Before loop begins, no divisors have been found, and *divisors* is initialized to 0
- In each cycle, 1 is added to *divisors* if a new divisor is found. Thus, at the end of *i*th cycle, *divisors* holds the number of divisors between 1 and *i*
- At the end of *n*th cycle, *divisors* contains the number of divisors between 1 and *n*
- If *divisors* contains 2, then *n* must be a prime

9

## Efficiency of Algorithms

- There exist *more than one algorithm* for most problems
- Look for algorithms that *run fast* and *use memory less*

10

## Efficiency of Algorithms

- As soon as isPrime finds any divisor greater than 1 and less than n, it can stop right there
- Once isPrime has checked whether *n* is divisible by 2, it doesn't need to check other even numbers
- isPrime only needs to check numbers up to $\sqrt{n}$

11

## Efficiency of Algorithms

```
bool isPrime(int n)
{
   int  i;

   if (n % 2 == 0) return FALSE;
   for (i = 3; i <= sqrt(n); i += 2) {
      if (n % i == 0) return FALSE;
   }
   return TRUE;
}
```

12

## Common Pitfalls

- When you design a general algorithm, it is important to consider whether there are any *special cases* in which the general algorithm might fail. If there are, you must be sure to handle these cases *explicitly* in the program

```
if (n <= 1) return FALSE;
if (n == 2) return TRUE;
```

13

## Common Pitfalls

- Check to see if there are any calculations *within* the loop that could just as well be performed *before* the loop begins. If there are, you can increase the efficiency of the program by calculating the result *once*, storing the result in a variable, and the using that variable inside the loop

```
limit = sqrt(n);
for (i = 3; i <= limit; i += 2)
```

14

## Common Pitfalls

- Be very careful when *testing floating-point numbers for equality*. Because floating-point numbers are only *approximations*

```
limit = sqrt(n) + 1;
```

15

## Efficiency of Algorithms

```
bool isPrime(int n) {
   int  i, limit;
   if (n <= 1) return FALSE;
   if (n == 2) return TRUE;
   if (n % 2 == 0) return FALSE;
   limit = sqrt(n) + 1;
   for (i = 3; i <= limit; i += 2) {
      if (n % i == 0) return FALSE;
   }
   return TRUE;
}
```

16

## Measuring Execution Time

- The function
  clcok_t clock(void)
  in the library <time.h> returns the processors time used by the program since the beginning of execution. The value of
  (double) clock() / CLOCKS_PER_SEC
  is a time in seconds

17

## Measuring Execution Time

```
bool isPrime(int n) {
   int  divisors, i;
   clock_t begin, end;
   begin = clock();
   divisors = 0;
   for (i = 1; i <= n; i++) {
      if (n % i == 0) divisors++;
   }
   end = clock();
   printf("%lf\n", ((double) end-begin)/CLOCKS_PER_SEC);
   return (divisors == 2);
}
```

18

## Evaluating Algorithms

- There are many *measures* for evaluating algorithms: *efficiency*, *maintainability*, *portability*, and so on
- It is often the case that no algorithm is *best* in *all* measures
- You then need to choose your algorithm based on your *requirements*

19

## Greatest Common Divisor

- Given two numbers, *x* and *y*, the *greatest common divisor* is the largest number that divides evenly into both

  int gcd(int x, int y);

20

## A Simple Algorithm

- The greatest common divisor of *x* and *y* must be between 1 and *x* (or *y*)
- Starting from *g* = *x*, check whether *g* divides evenly into both *x* and *y*
- If it does, *g* is the greatest common divisor
- Otherwise, decrement *g* by 1 and check again

21

## A Simple Algorithm

```
int  gcd(int x, int y)
{
   int  g;

   g = x;
   while (x % g != 0 || y % g != 0) {
      g--;
   }
   return g;
}
```

22

## Correctness

- Does this algorithm always give the correct answer
- Does this algorithm always terminate

23

## Euclid's Algorithm

- Divide *x* by *y* and compute the remainder *r*
- If *r* is zero, the procedure is complete, and the answer is *y*
- If *r* is not zero, set *x* equal to the old value of *y*, set *y* equal to *r*, and repeat the entire process

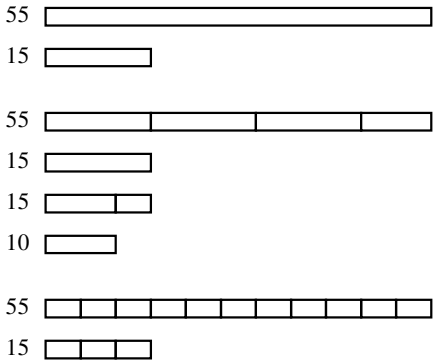
24

## Euclid's Algorithm

```
int  gcd(int x, int y)
{
   int  r;
   while (TRUE) {
      r = x % y;
      if (r == 0) break;
      x = y;  y = r;
   }
   return y;
}
```

25

## Correctness




26

## Efficiency

- Consider the numbers 1000005 and 1000000
- The simple algorithm requires 1000000 steps
- Euclid's algorithm requires only 2 steps

x = 1000005, y = 1000000, r = 5
x = 1000000, y = 5, , r = 0

27

## Numerical Algorithms

- The techniques used by computers to implement mathematical functions like **sqrt** are called *numerical algorithms*
- *Successive approximation* and *series expansion* are two of the techniques
- Use the computation of *square root* to illustrate these two techniques

28

## Successive Approximation

- Successive approximation is a general strategy for finding an *approximate* answer
- It starts by making a guess at the answer
- It then uses that guess to generate a better one
- If you can somehow guarantee that your guess is *getting closer and closer* (or *converging*) to the real answer on each cycle, repeating the process will eventually result in a guess that is *close enough* to satisfy the needs of any application

29

## Newton's Algorithm

Square root of 16.0:

guess = 8.0,
16.0 / 8.0 = 2.0, $2.0 < \sqrt{16.0} < 8.0$
guess = (2.0 + 8.0) / 2 = 5.0,
16.0 / 5.0 = 3.2, $3.2 < \sqrt{16.0} < 5.0$
guess = (3.2 + 5.0) / 2 = 4.1,
guess = 4.001219512,
guess = 4.00000018584,

30

## Newton's Algorithm

- To compute the square root of $x$, it starts by making an arbitrary guess $g$
- If $g$ is close enough to the actual square root, $g$ is returned
- If $g$ is not close enough, new $g = (g + x/g)/2$, and repeat the process

31

## Newton's Algorithm

```
double  mySqrt(double x)
{
   double  g;
   if (x == 0) return 0;
   if (x < 0) {
      printf("Error: negative argument %lf\n", x); return 0;
   }
   g = x;
   while (!approximatelyEqual(x, g * g))
      g = (g + x / g) / 2;
   return g;
}
```

special cases

32

## Newton's Algorithm

```
#include  <math.h>
#define  MIN(x, y)  (((x) < (y)) ? (x) : (y))
#define  EPSILON  0.000001

bool approximatelyEqual(double x, double y)
{
   double z;

   z = fabs(x – y) / MIN(fabs(x), fabs(y));
   return z < EPSILON;
}
```

33

## Series Expansion

- In *series expansion*, the value of a function is approximated by adding together terms in a mathematical series that converges

  1/2 + 1/4 + 1/8 + 1/16 + 1/32 + 1/64 + …

34

## Taylor Series

- The square root of $x$ can be approximated by the following *Taylor series*

  $1 + (1/2)(x\text{-}1) – (1/4)(x\text{-}1)^2/2! + (3/8)(x\text{-}1)^3/3!$
  $- (15/16)(x\text{-}1)^4/4! + \ldots$

35

## Taylor Series

$t_0 = 1$
$t_1 = (1/2)(x\text{-}1)$
$t_2 = -(1/4)(x\text{-}1)^2/2!$
$t_3 = (3/8)(x\text{-}1)^3/3!$
$t_4 = -(15/16)(x\text{-}1)^4/4!$

coeff * xpower / factorial

coeff *= (0.5 – $i$)
xpower *= ($x$ – 1)
factorial *= ($i$ + 1)

36

## Taylor Series

```
double  tsqrt(double x) {
   double  sum, term, coeff, xpower, fact;
   int i;
   fact = coeff = xpower = 1;
   sum = 0;  term = 1;
   for (i = 0; sum != sum + term; i++) {
      sum += term;
      coeff *= 0.5 – i; xpower *= x –1; fact *= i + 1;
      term = coeff * xpower / fact;
   }
   return sum;
}
```

37

## Radius of Convergence

- The Taylor expansion for the square root function is effective only when the argument falls within a *limited range* ($0 < x < 2$) that allows the calculation to converge. This range is called the *radius of convergence*

$$\sqrt{4x} = 4\sqrt{x} = 2\sqrt{x}$$

38

## Radius of Convergence

```
double  mySqrt(double x)
{
   double  correction;
   if (x == 0) return 0;
   if (x < 0) {
      printf("Error: negative argument %lf\n", x); return 0;
   }
   correction = 1;
   while (x >= 2) {
      x /= 4;  correction *= 2;
   }
   return tsqrt(x) * correction;
}
```

39