Introduction

Madiba Hudson-Quansah

Contents

Chapter 1

Introduction

1.1 What is an Algorithm

Definition 1.1.1: Algorithm

A sequence of unambiguous instructions for solving a problem i.e. for obtaining a required output for any legitimate input in a finite amount of time.

Using the task of finding the greatest common divisor (GCD) of two numbers we can explicitly illustrate the general notions of an algorithm. Namely:

- The requirement of non-ambiguity in describing each step of an algorithm.
- The specified range of inputs for which an algorithm works.
- The different ways the same algorithm can be expressed / implemented.
- The non uniqueness of algorithms for a particular problem
- Different algorithms for the same problem may be based on different ideas and have different time and space complexities.

The GCD of two non-negative, non-zero integers m and n, denoted $\gcd(m, n)$ is defined as the largest integer that divides both m and n evenly, i.e., with a remainder of zero. Euclid of Alexandria, a Greek athematician, outlined an algorithm for solving this problem in his book *Elements*. Defined as follows:

$$\gcd(m, n) = \begin{cases} m & \text{if } n = 0 \\ \gcd(n, m \mod n) & \text{otherwise} \end{cases}$$

Where $m \mod n$ is the remainder of the division of m by n, and when n is zero the GCD is m as it is the only and largest number that divides m evenly between m and 0. In pseudocode:

Algorithm 1 gcd (m, n)

▷ Computes the greatest common divisor of two numbers
▷ Input: Two non-negative integers m and n
▷ Output: The greatest common divisor of m and n

while $n \neq 0$ **do**
    $r \leftarrow m \mod n$
    $m \leftarrow n$
    $n \leftarrow r$
end while
return r

This algorithm eventually terminates as the value of n decreases with each iteration as its value is set to $m \mod n$ every iteration, and the value of m is always greater than n.

Another algorithm for solving this problem is based on the definition of the GCD as the greatest common divisor of m and n as the largest integer that divides both number evenly. This leads us to the conclusion that such a common divisor cannot be greater than the smallest of the two numbers, denoted $t = \min\{m, n\}$. We can then check if t divides both numbers m and n evenly, if it does then t is the GCD of m and n. If it does not, we decrease t by 1 and then check if $t - 1$ divides both m and n evenly. This process is repeated until we find a number that divide both m and n evenly or until $t = 1$ in which case the GCD is 1 as 1 divides all numbers evenly. In pseudocode:

Algorithm 2 Consecutive Integer Checking (m, n)

$t \leftarrow \min\{m, n\}$
while $t > 1$ **do**
    if $m \mod t == 0$ and $n \mod t == 0$ **then**
        return t
    end if
    $t \leftarrow t - 1$
end while
return t

The final algorithm considers the prime factorization of the two numbers m and n. The GCD of two numbers is the product of the common prime factors of the two numbers. As there is no unambiguous way to find the prime factorization of a number we simply layout the steps for the algorithm for now.

1. Find the prime factorization of m
2. Find the prime factorization of n
3. Identify the common prime factors of m and n
4. Multiply the common prime factors and return it as the GCD

A simple way of generating consecutive prime numbers not exceeding a given integer $n > 1$ is the Sieve of Eratosthenes. This algorithm starts by initializing a list of prime candidates with consecutive integers from 2 to n. Then each iteration of the algorithm selects the next number in the list as a prime number and removes all multiples of that prime number in the list. This continues until no more numbers can be selected as prime numbers i.e. no number can be removed from the list. The remaining numbers in the list are the prime numbers not exceeding n. In pseudocode:

Algorithm 3 SieveNaïve (n)

▷ Generates a list of prime numbers not exceeding $n > 1$
▷ Input: A positive number to not exceed n
▷ Output: A list of prime numbers not exceeding n

```
1:  function ZeroMultiples(L, p, n)
2:      for i ← 1 to n do
3:          if L_i ≠ 0 and p ≠ L_i and L_i mod p == 0 then
4:              L_i ← 0
5:          end if
6:      end for
7:      return L
8:  end function
9:
10: P[n]                                              ▷ P is an array of size n
11: for i ← 2 to n do
12:     P_i ← i
13: end for
14: for i ← 1 to n do
15:     if P_i ≠ 0 then
16:         prime ← P_i
17:         P ← ZeroMultiples (P, prime, n)
18:     end if
19: end for
20:
21: T [n]                                             ▷ Temporary array to store prime numbers
22: i ← 0
23: for j ← 1 to n do
24:     if P_j ≠ 0 then
25:         T_i ← P_j
26:         i ← i + 1
27:     end if
28: end for
29:                                                   ▷ Remove trailing zeros
30: return T
```

But through observation, we can deduce that the largest number p whose multiples can still remain to warrant further iterations of the algorithm. We must first note that if p is a number whose multiples are being eliminated on the current pass, then the first multiple of p we must consider is $p \times p$ because all of its smaller multiples $2p, \ldots, (p-1)$ have been eliminated on earlier passes. This prevents us from eliminating the same number more than once. As $p \times p$ should not be greater than n, p cannot exceed $\sqrt{n}$ floored i.e. $\lfloor \sqrt{n} \rfloor$. We assume in the following pseudocode that there is function available for computing $\lfloor \sqrt{n} \rfloor$.

Algorithm 4 Sieve of Eratosthenes (n)

▷ Generates a list of prime numbers not exceeding $n > 1$
▷ Input: A positive number to not exceed n
▷ Output: A list of prime numbers not exceeding n

```
1:  for p ← 2 to n do
2:      A[p] ← p
3:  end for
4:
5:  for p ← 2 to ⌊√n⌋ do
6:      if A_p ≠ 0 then                          ▷ p hasn't been eliminated on previous passes
7:          j ← p × p
8:          while j ⩽ n do
9:              A_j ← 0
10:             j ← j + p
11:         end while
12:     end if
13: end for
14:                                              ▷ Copy remaining elements of A to array L of the primes
15: i ← 0
16: for p ← 2 to n do
17:     if A_p ≠ 0 then
18:         L_i ← A_p
19:         i ← i + 1
20:     end if
21: end for
22: return L
```

1.1.1 Exercises

Question 1

Design an algorithm for computing $\lfloor \sqrt{n} \rfloor$ for any positive integer n. Besides assignment and comparison, your algorithm may only use the four basic arithmetical operations.

Solution:

Algorithm 5 FloorSqrt (n)

▷ Computes $\lfloor \sqrt{n} \rfloor$ for any positive integer n
▷ Input: Integer n
▷ Output: The floored square root of n

```
1:  function ABS( x)
2:      if x < 0 then
3:          return −1 × x
4:      else
5:          return x
6:      end if
7:  end function
8:                                          ▷ Newtons Method for calculating square root
9:  x ← 10
10: while ABS(x × x − n) > 0.001 do
11:     x ← (x + n/x) /2
12: end while
13:
14: r ← x                                   ▷ Approximate definition of the floor function
15: while r ⩾ 1 do
16:     r ← r − 1
17: end while
18: return x − r
```

Algorithm 6 FloorSqrtBinarySearch (n)

▷ Finds the floored square root of a positive integer n using binary search
▷ Input: A positive integer n
▷ Output: The floored square root of the integer n

```
1:
2:  low ← 0
3:  high ← n
4:  while high > low do
5:      mid ← (high + low) / 2
6:      if mid × mid ⩽ n and (mid + 1) × (mid + 1) > n then
7:          return mid
8:      end if
9:      if mid × mid < n then
10:         low ← mid + 1
11:     else
12:         high ← mid − 1
13:     end if
14: end while
```

Question 2

Design an algorithm to find all the common elements in two sorted lists of numbers. For example, for the lists 2, 5, 5, 5 and 2, 2, 3, 5, 5, 7, the output should be 2, 5, 5. What is the maximum number of comparisons your algorithm makes if the lengths of the two given lists are m and n, respectively?

Solution:

Algorithm 7 Common (A, m, B, n)

▷ Finds the common elements in two lists A and B keeping their multiplicities
▷ Input:
▷ Output:

```
1: i ← 1
2: j ← 1
3: k ← 1
4: while i ⩽ m and j ⩽ n do
5:     if A_i == B_j then
6:         L_k ← A_i                          ▷ L is the list of common elements
7:         i ← i + 1
8:         j ← j + 1
9:         k ← k + 1
10:    else if A_j > B_i then
11:        j ← j + 1
12:    else
13:        i ← i + 1
14:    end if
15: end while
```

Chapter 2

Algorithm Design

For a time function $T(n)$ if the assumptions of the RAM model cannot be confirmed, then the actual time function of an algorithm is:

$$a \times T(n) \leqslant T(n) \leqslant b \times T(n)$$

Where a is the least time taken by any primitive operation and b is the most time taken by any primitive operation.

2.1 Primitive Operations

Definition 2.1.1: Primitive Operation

Any operation that takes a constant amount of time to execute. Usually primitive operations are implemented as machine instructions.

Examples of primitive operations include:

- Evaluating an expression
- Variable Assignment
- Array indexing
- Comparisons
- Arithmetic operations
- Function calls
- Returning from a function

2.2 Growth Rate of Running Time

The growth rate of a function can also described informally as:

$$\lim_{n \to \infty} T(n) \approx O(n)$$

For the formal definitions see `2_Algorithm_Analysis.pdf` in the `CompSci/Data_Structures` folder.

Example 2.2.1

Question 3

Show that $T(n) = 5n + 3$ is $O(n)$ using the big-O notation

Solution: Let $f(n) = 5n + 3$ and $g(n) = n$. O is defined as for a given function $g(n)$:

$$O\big(g(n)\big) = \big\{f(n) : \exists c > 0, n_0 \in \mathbb{N} \text{ such that } 0 \leqslant f(n) \leqslant c \cdot g(n)\ \forall n \geqslant n_0\big\}$$

$\therefore$

$$5n + 3 \leqslant cn\ \forall n \geqslant n_0$$

$$\begin{aligned} 5n &\leqslant cn \\ c &= 5, 6, 7, \ldots \\ \text{Since } 5n + 3 \leqslant 5n = F \text{ we chose } c &= 6 \\ 5n + 3 &\leqslant 6n \\ 3 &\leqslant 6n - 5n \\ 3 &\leqslant n \\ \therefore c = 6\, n_0 &= 3 \end{aligned}$$

$\therefore T(n) = 5n + 3$ is $O(n)$.

Example 2.2.2

Question 4

Show that $T(n) = 5n^2 + 3n - 7$ is not $O(n)$

Solution: Let $f(n) = T(n)$ and $g(n) = n$. From the definition of $O(n)$:

$$O\big(g(n)\big) = f(n) \leqslant cg(n)\ \forall n \geqslant n_0$$

$$\begin{aligned} 5n^2 + 3n - 7 &\leqslant cn \\ 5n^2 + 3n &\leqslant cn \\ 5n + 3 &\leqslant c \end{aligned}$$

As c is a constant but is dependent on n there is no c that satisfies the inequality
$\therefore T(n)$ is not $O(n)$