ECE 406 Course Notes

Algorithm Design and Analysis

Paolo Torres

**University of Waterloo**

**Winter 2021**

Table of Contents

[1 Introduction and Basic Arithmetic 1](#_Toc61715006)

[1.1 Algorithms, Correctness, Termination, Efficiency 1](#_Toc61715007)

[1.1.1 Algorithms 1](#_Toc61715008)

[1.1.2 Correctness 1](#_Toc61715009)

[1.1.3 Termination 1](#_Toc61715010)

[1.1.4 Time Efficiency 2](#_Toc61715011)

[1.1.5 Claim 1 2](#_Toc61715012)

[1.1.6 More Efficient Algorithm 3](#_Toc61715013)

[1.1.7 Note on Measuring Time Efficiency 3](#_Toc61715014)

[1.2 Big-O Notation 3](#_Toc61715015)

[1.2.1 Definition 1 (O) 3](#_Toc61715016)

[1.2.2 Definition 2 () 3](#_Toc61715017)

[1.2.3 Definition 3 () 3](#_Toc61715018)

[1.2.4 Example 4](#_Toc61715019)

[1.2.5 Big-O Explanation 4](#_Toc61715020)

[1.2.6 Big-O Simplifications 5](#_Toc61715021)

[1.3 Arithmetic 5](#_Toc61715022)

[1.3.1 Addition 5](#_Toc61715023)

[1.3.2 Multiplication 5](#_Toc61715024)

[1.3.3 Division 6](#_Toc61715025)

[2 Algorithms with Numbers 7](#_Toc61715026)

[2.1 Modular Arithmetic 7](#_Toc61715027)

[2.1.1 Example Application: Two’s Complement Arithmetic 7](#_Toc61715028)

[2.1.2 Modular Addition, Subtraction 8](#_Toc61715029)

[2.1.3 Modular Multiplication 8](#_Toc61715030)

[2.1.4 Modular Exponentiation 9](#_Toc61715031)

[2.1.5 Towards Modular Division: GCD Using Euclid 9](#_Toc61715032)

[2.1.6 Towards Modular Division: Extended Euclid 10](#_Toc61715033)

[2.1.7 Modular Division 11](#_Toc61715034)

[2.2 Primality Testing 11](#_Toc61715035)

[2.2.1 Fermat’s Little Theorem 12](#_Toc61715036)

[2.3 Generating an n-Bit Prime 13](#_Toc61715037)

# Introduction and Basic Arithmetic

## Algorithms, Correctness, Termination, Efficiency

### Algorithms

Given the specification for a function, an algorithm is the procedure to compute it.

Example:

, where

Commonly used sets:

Fibonacci Sequence:

Important aspects:

* Function has been specified as a recurrence, so a recursive algorithm seems natural
* Imperative (procedural) specification of an algorithm has consequences:
  + Intuiting correctness can be a challenge
  + Intuiting time and space efficiency may be easier
* No mundane error checking, can focus on core logic
* Input value is unbounded but finite

### Correctness

Correctness refers to an algorithm’s ability to guarantee expected termination. In the case of , it is a direct encoding of the recurrence.

### Termination

The end of an algorithm. It can be proven that terminates on every input by induction on .

### Time Efficiency

Can be calculated by counting the number of: comparisons – these happen on Lines and , and number of additions – this happens on Line .

Suppose represents the time efficiency of :

How bad is ? Is it exponential in ?

For all , .

### Claim 1

For all

If this claim is true, then , and because , is exponential in .

Proof for the claim: by induction on .

Does a better algorithm exist from the standpoint of time efficiency?

Diagram

Description automatically generated

Recall how subroutine (recursive, in this case) invocation works:

* Every node in the tree corresponds to an invocation of the algorithm
* Sequence of invocations corresponds to a pre-order traversal
* Maximum depth of the call stack at any moment:

Main point in this case: Redundancy, , appears more than once.

### More Efficient Algorithm

Let be the of comparisons plus additions on input :

Linear in for , more efficient than .

### Note on Measuring Time Efficiency

Need to pick the right level of abstraction, meaning picking some kind of “hot spot” or “hot operation,” then count. For example, number of additions, comparisons, recursive calls, etc.

## Big-O Notation

### Definition 1 (O)

Let , and be functions. Define if there exists a constant such that .

* Typically consider non-decreasing functions only

### Definition 2 ()

Define if

### Definition 3 ()

Define if and

* analogous to
* analogous to
* analogous to

### Example

Chart, line chart

Description automatically generated

Precise answer to this question: depends on .

But in big-O notation:

* . Proof: Adopt as the constant for any

### Big-O Explanation

Suppose algorithm A runs in time, B in , and C in . Now suppose the speed of the computer doubles, which algorithm gives the best payoff?

For a given time period , what is the largest input each algorithm can handle? Set and , solve for :

|  |  |  |
| --- | --- | --- |
| **Algorithm** | **Old Computer** | **New Computer** |
| A |  |  |
| B |  |  |
| C |  |  |

So, payoff with algorithm A is approximately , B is , and C is .

### Big-O Simplifications

* Multiplicative and additive constants can be omitted
* dominates for
* Any exponential dominates any polynomial, any polynomial dominates any logarithm
* Big-O simplifications should be used prudently, not applicable in all settings

## Arithmetic

### Addition

Hypothesize access to a function :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Carry** | **One Digit** | **Other Digit** | **Result Carry** | **Result Sum** |
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 1 |
| … | | | | |
| 1 | 8 | 9 | 1 | 8 |
| 0 | 9 | 9 | 1 | 8 |
| 1 | 9 | 9 | 1 | 9 |

To add and :

digits needed to encode

bits needed to encode

So, time efficiency of an algorithm to add as measured by number of lookups to T:

* in the best case
* in the worst case
* So, either way, , or linear time, where is the size of the input

### Multiplication

For , encoded in binary:

Straightforward encoding as recursive algorithm .

Graphical user interface, text, application

Description automatically generated

Worst case running time:

* + One comparison to , one division by (right bit shift), one assignment to , one check for evenness (check LSB), one multiplication by (left bit shift), one addition of -bit numbers
* So, in the worst case

### Division

**Definition 1:** Given , the pair where , of divided by are those that satisfy:

**Claim 1:** For every , , as defined above exists, and is unique.

To specify a recurrence for , denote as , the result of divided by . Now:

**Claim 2:** The above recurrence is correct.

*Proof:* Cases are exhaustive. Proof by case-analysis and induction on bits to encode .

By induction assumption: .

Text

Description automatically generated

Running time: .

# Algorithms with Numbers

## Modular Arithmetic

* **Definition 1:**
* **Definition 2:** 
  + Example:

### Example Application: Two’s Complement Arithmetic

Suppose we want to represent, using bits, positive and negative integers, and . Could reserve bit for sign. This would allow us to represent integers in the interval , with a “positive zero” and a “negative zero.”

In two’s complement arithmetic, we have exactly one bit-string for , and represent integers in the interval . How? Represent any as the non-negative integer modulo . So:

* Ex:
* Ex:

All arithmetic performed modulo :

* Ex:

**Claim 1:**

**Claim 2:**

(mod N) (mod N) (mod N)

Example:

### Modular Addition, Subtraction

* Any intermediate result is between and
* So, time efficiency is , where

### Modular Multiplication

Let and be our algorithms for non-modular multiplication and division:

* Any intermediate result (specifically, result of ) is between and
* So, time efficiency is , where

### Modular Exponentiation

* Recall: We used “repeated doubling” for non-modular multiplication and “repeated halving” for non-modular division
* Similarly, here, use “repeated squaring”:

Graphical user interface, text

Description automatically generated

### Towards Modular Division: GCD Using Euclid

* In a non-modular world, . Only case for which doesn’t exist:
* In a modular world, may not exist even if

Crucial building block: GCD. Example: What is ?

But factoring into prime factors conjectured to be computationally hard in the worse case

**Claim 3:**

*Proof:* Suffices to prove: . Now prove and .

Table

Description automatically generated with medium confidence

How fast does converge?

**Claim 4:**

So: Guaranteed to lose at least bit for every recursive call time efficiency is

### Towards Modular Division: Extended Euclid

**Claim 5:**

**Claim 6:**

Text

Description automatically generated with medium confidence

Example run on input :

|  |  |
| --- | --- |
| **Arguments** | **Return Value** |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

A good example to segue to our final step in modular division.

We figured out: , which implies:

* . So: is multiplicative inverse of
* So, for example:

### Modular Division

**Definition 3:**

**Claim 7:**

**Claim 8:**

**Definition 4:**

**Claim 9:**

So, to compute :

1. Determine whether
2. If yes to (i) determine , and
3. If yes to (i) compute

(i) and (ii) are done simultaneously by

Running Time:

## Primality Testing

Given , is prime?

For a decision problem, i.e., co-domain of function to be computed as , a randomized algorithm:

* Has access to an unbiased coin
* Is deemed to be correct if:

Suppose:

* We run such an algorithm times, pairwise independently
* We return if and only if every run returns
* Then,

### Fermat’s Little Theorem

**Claim 1:**

To prove Fermat’s little theorem, leverage the following:

**Claim 2:**

*Proof for Claim 2:* We know that are relatively prime, so exists.

*Proof for Claim 1:* From Claim 2:

Also, exists.

So,

Text

Description automatically generated

Issues with the algorithm:

1. Fermat’s little theorem is not an “if and only if”: Carmichael numbers.
2. Suppose is not prime/Carmichael. For chosen, say, uniformly from , what is ?

* We know such an exists, but how likely is it that we will pick it?

We do not deal with (1) – cop out: Carmichael numbers are rare.

**Claim 3:** If for relatively prime, then it must hold for at least half the choices .

*Proof:* If there exists no with , then we are done.

If such a exists, then .

Also, if exist with , , then:

So, at least as many as there are .

So:

For runs of on uniform, independent choices of :

## Generating an n-Bit Prime

**Claim 4:** .

So, algorithm for generating a prime:

1. Randomly generate -bit number, .
2. Check whether is prime.
3. If not, go to Step (1).

Guaranteed return in Step (2) if is indeed prime.

Each trial in the above algorithm is a Bernoulli trial:

* Only one of two outcomes: success or failure

In a Bernoulli trial if , then

* Example:
* Example:

So, we expect the above algorithm to halt in iterations.