**Chapter 3:**

**Algorithms**: a finite set sequence of precise instructions for preforming a computation or solving a problem.

**Pseudocode**: acts as an intermediate step between English and programming language.

Finding the maximum element in a finite sequence of integers.

Searching algorithms

Linear search: Where we have to locate element x in a list of distinct elements s

Binary Search: splitting what set is given into two smaller subsets

If the terms are numbers listed from smallest to largest

* We start by taking the middle terms
* Split them into two smaller sub lists of the same size

**Sorting:**

Elements in a list (increasing or decreasing order)

**Bubble sort** (sinking set)

* Put a list into increasing order by comparing it to the adjacent elements
  + Interchanging them if they are in the wrong order
* Perform basic operation
  + Interchanging a larger element with a smaller one following it
* Start at the beginning of the list for a full pass
* Continue procedure until the sort is complete

**Algorithm for bubble sort**:

Procedure bubble sort(a1, … , an : real numbers with n > 2)

For i := 1 to n – 1

For j := 1 to n – I

If aj > aj + 1 then interchange aj and aj + 1

{a1, … , an is in increasing order}

**Intersection sort**:

1. If a2 <= a1 then swap a1 and a2 [then first two would be in order]
2. Order a1, a2, a3 correctly [now first three are in order]
3. Order a1, … , aj + 1 correctly [now first j elements are in order]

Algorithm for intersection sort:

Procedure for intersection sort (a1, a2, … , an : real numbers with n >= 2)

For j := 2 to n

i := 1

while aj > ai

i := i + 1

m := aj

for k := 0 to j – i – 1

aj – k := aj – k – 1

ai := m

{a1, … , an is in increasing order}

Greedy algorithms:

For scheduling tasks

In general, a greedy algorithm makes the best choice at each step (according to a specified criteria)

**3.2:**

**The Growth Of functions:**

Big – O notation estimates the growth of a function without worrying about constant multipliers or small order terms

Function f(x) as x increases without bound (i.e., as x 🡪 infinity)

* That means that using big O notation we do not have to worry about the hardware and software used to implement an algorithm.
* Furthermore, using big O notation we can assume that the different operations used in an algorithm take the same time which simplifies the analysis considerably.

Def. Let f and g be functions the set of integers or the set of real numbers to the set of real numbers we say that f(x) is O g(x) if these are constants C and k such that

|f(x)| <= C|g(x)| whenever x > k

* The constants C and k in the definition of big O notation are called witnesses to the relationship f(x) is Og(x)
* Note that f(x) is Og(x) if f grows slower than some fixed multiple of g(x) as x gets larger (as x grows without bound or to infinity)
* Note that f(x) is Og(x) if exists C, k E R (x > k => |f(x)| <= C|g(x)|)

Example: Show f(x) = 4 is O(x)

Theorem: If f(x) is a polynomial of log n then f(x) is O(x^)

**Big Omega Notation:**

Let f and g be functions from integer to integer or real number to real number we say that f(x) is omega g(x) if there are positive constants C and K such that |f(x)| >= C |g(x)| whenever x > k

Note: f(x) is big omega g(x) if and only if g(x) is Of(x)

Big Theta Notation:

Let f and g be functions from integers to integers or real numbers to real numbers that f(x) is ~g(x) if f(x) is O((g(x))) and f(x) big omega (g(x)) when f(x) is ~g(x) we say that f is big theta of g(x) are of the same order. When f(x) is ~g(x) it is also the case that g(x) is ~f(x)

Note: f(x) is ~g(x) if and only if f(x) is O(g(x)) and g(x) is O(f(x))

Big Oh:

Any bounded function is O(x)

Any function f(x) that is O(x) falls between the lines for all x > k

We are trying to find out which function ends up steeper a x 🡪 infinity

3.3:

Complexity of algorithms:

When does an algorithm provide a satisfactory solution to a problem?

* It must produce a correct answer
* It should be efficient

How can efficiency of an algorithm be analyzed?

* One measure of efficiency is the time used by a computer to solve a problem using the algorithm when input values are specified
* Second measure is the amount of computer memory required to implement the algorithm when input values are specified.

Time complexity: How much time or number of instructions does an algorithm require?

Space complexity: How much memory does an algorithm require?

We are concerned only with the time complexity as measured in the number of a defined operation such as addition, subtraction, comparison… etc.

Eg. How many addition in the code segment below? (ignore the loop counters)

**Chapter 4:**

Divisibility and modules Arithmetic’s:

Division of integer by positive integer

Theorem: Let a, b, and c be integers where a =/ 0 then

1. If a|b and a|c then a|(b + c)
2. If a|b then a|bc for all integers c
3. If a|b and b|c then a|c

When integer is divided by a positive integer there is a quotient and a remainder as the division algorithm shows

Theorem: The division algorithm (not really an algorithm though)

Let a be an integer and d be a positive integer then there are unique integers q and r with O <= r <= d such that a = dq + r

In above we say that q = a div d and r = mod d where div and mod are functions over integers

Note: a div d = a/d

Note: a mod d = a – [a/d]d

Modular arithmetic’s: if a and b are integers and m is a positive integer, a is congruent to b modulo m

[a = b(mod m)] if and only if m|(a – b)

Theorem: let a and b be integers and let m be a positive integer then a is equivalent to b(mod m) if and only if a mod m is equivalent to b mod m

Arithmetic Modula m: integer of m = {0, 1, …, m - 1} addition of these integers +m

A +m b = (a +b)mod m

Multiplication of these integers would look like this:

a \*m b = (a \* b)mod m

4.2: Integer Representation and Algorithms:

Integers can be expressed using any integer > 1 as base

Theorem 1: the base expansion of n: Let b be an integer greater than 1 then if n is a positive integer it can be

n = a base k b^k + a base k – 1 b^k – 1 + … + a, b^1 + a base 0

where k = is a non-negative integer a0, a1, …, ak are non-negetive integers less than b and ak =/ 0

Use binary {0, 1}

Octa {0, 1, 2, 3, 4, 5, 6, 7}

Hexadecimal {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}

Note: Binary, octa, hexa conversion (they are easy)

Binary addition

C := 0

For j:= 0 to n – 1

D := [(aj + bj + c)base 2]

Sj := aj + bj + c – 2d

C := d

Sn := C

Extra Credit Hw: Implement the above in mathmatica

4.2: Primes and greatest common divisor:

Prime definition: an integer p greater than one is prime if the only positive factor of p are 1 and p

* A positive number that is greater than 1 and is not prime is called a composite number
* The integer n is composite if and only if there exists an integer a such that a|n and |< a < n

Theorem 1:

The fundamental theorem of arithmetic’s.

4.3:

Find an inverse of 101 modulo 4620

Euclidean Algorithm

#1 4620 = 25 \* 101 + 75

101 = 1 \* 75 + 26

75 = 2 \* 26 + 23

26 = 1 \* 23 + 3

23 = 7 \* 3 + 2

3 = 1 \* 2 +1

2 = 2 \* 1 + 0

GCD = 1 (Last non-zero remainder)

1 = 3 + (-1)2

1 \* 3 + (-1)(23 – 7 \* 3)

Check pictures from here

Note: every multiple of 29 or 29 away from

Cryptography

Julius Ceasar invented it

Chapter 5:

Mathematical induction:

Most important proof in CS

Very Easy

Induction is just an axiom

Let p(n) be a predicate. If p(1) is true and n E |N then P(n) 🡪 P(n + 1) is true then P(n) is true as well

**This is a proof by mathematical induction:** To prove that P(n) is true for all positive integers n, where P(n) is a propositional function. To find this fist we verify that P(1) is true. Then through inductive reasoning we show that the conditional statement P(k) 🡪 P(k + 1) is true for all positive integers k.

Note that Induction does not always begin at n = 0 or n = 1

False induction:

Not a theorem: All horses are the same color

Since color (H1) = Color… ya know what. Look at picture

Moral – you must establish the inductive step for all n greater than or equal to the base case. If the base case above were n = 2 it would have failed.

**Strong induction**: To prove that P(n) is true for all positive integers n, where P(n) is a propositional function we complete two steps

* Basis step: we verify that P(1) is true
* Inductive step: We show that the conditional statement [P(1)^ P(2)^…^ P(k)^ 🡪 P(k + 1) is true for all positive integers

Strong induction is equivalent to induction but is easier in some cases.

Both follow from the Well Ordering property (W.O.P) which means that every non-empty set of non-negative integer has a least element

**Strong induction using examples**

**Fundamental theorem of arithmetic’s:**

If n is an integer greater than 1 then n can be written as a product of primes

**Proof by strong induction:**

Let P(n) be the proposition that n can be written as a product of primes

**Basis case**: 2 is the product of the prime itself so P(2) is true

**Inductive step**: The inductive hypothesis is the assumption that P(j) is true for all integers j with 2 <= j <= k. To complete the inductive step we must show P(k + 1) is true under assumption that is that k +1 is the product of primes. If k + 1 is prime we immediately see that P(k + 1) is true. Otherwise k + 1 is composite and can be written as the product of two positive integers a and b with 2 <= a <= k and 2 <= b <= k such that k + 1 = a \* b

Because both a and b are integers at least 2 and not exceeding k we can use the inductive hypothesis to write both a and b as the products of primes namely those primes in the factorization of a and the factorization of b

Theorem: All strategies for the n-block game produce the same score S(n)

**From here look at picture**

Recall that when we are stuck on induction proof try making the predicates stronger

Determine which amount of postage can be formed using just 4 cent and 11 cent stamps

Base case : (30, 31, 32, 33)

4 base case [since 4 is smaller]

Got 30 because we used 11 + 11 + 4 + 4

31 = 11 + 4+ 4 + 4 + 4 + 4

32 = 4 + 4 + 4+ 4 + 4 + 4 + 4 + 4

33 = 11 + 11 + 11

IH assume true for all j where 30 <= j <= k where k >= 33

We prove that (k + 1) – cent postage can be formed

Just add one four cent stamp and we are done

K – 3cent + 4 cent = k + 1

5.3

Recursive Definitions:

Specify a sequence by defining a term by previous terms

a base n = 2^n

can also be defined as a base 0 = 1 and a base n+1 = 2a base n

recursive definition of a sequence

Eg: Suppose that f is defined recursively by

F(0) = 3

F(n + 1) = 2f(n) + 3

Find f(1), f(2), f(3) and f(4)

Solution from the recursive definition it follows that

F(1) = 2f(0) + 3 = 2 \* 3 + 3 = 9

F(2) = 2f(1) + 3 = 21

F(3) = 2f(2) + 3 = 45

F(4) = 2f(3) + 3 = 93

Chapter 6

6.1

Counting

The basics of counting

We have two principles: Product rule and sum rule

Product rule: suppose that a procedure can be broken down into a a sequence of two tasks. If there are n ways to do the first task and for each of these ways of doing the first task there are n base 2 ways to do the second task then there are n \* n base 2 ways to do the procedure

Ex: a meal consists of a salad, main course, drinks, and dessert.

Number of choices 3 for salad, 4 for meals, 5 for drinks, 2 for desserts

So: 3 \* 4 \* 5 \* 2 = 120

Ex: 10 empty offices with 3 new employees

In how many ways can you assign an office to 3 new employees

Multiply the number of choices for each

Def. a procedure consists of k tasks

Let n base I = number of ways to do task I for all 1 <= I <= k

Then there are n base 1 \* n base 2… \* n base k ways to do the procedure

Ex; How many different bit strings of length 7 are there?

Each of the seven bits can be chosen in 2 ways because each bit is either 0 or 1

Therefore the product rule shows there are a total of 2^7 or 128 different bit strings of length

Counting functions:

How many functions are there from a set with n elements to a set with n elements?

A function corresponds to a choice of one of the n elements in the codomain for each of the m elements in the domain. Hence by the PRODUCT RULE there are n \*n \* n… \*n = n^m functions

Ex: f: [3] 🡪 [5]

Counting one-to-one Functions:

How many one-to-one functions are there from a set with m elements to one with n elements

Ex: f[3] 🡪 [5] = 5 \* 4 \* 3 = 60 one to one functions