Class Notes for Week 4: Algorithms and Asymptotic Analysis

Read these sections from the textbook:

§1.1 – 1.2;
§2.2;
§3.1;
§10.1 – 10.2;
§12.1;
§22.1;
§A.1-A.2
§B.4 – B.5

Topics for week 4

1) Algorithms
2) Example 1: algorithm to count the number of positive values in an array of real numbers
3) Example 2: The Greatest Common Divisor (GCD) problem.
4) Fundamentals of Algorithmic Problem Solving
5) Important Problem Types
6) Fundamental Data Structures
7) Essential Mathematical Ideas
   A) Rounding – floor, ceiling, nearest
   B) Integer arithmetic - mod and div
   C) Powers/Exponents
   D) Logarithms base 2
   E) Summations
8) Asymptotic Analysis
   the basics of algorithm analysis
   three asymptotic classes – Big-Oh, Big-Omega, Big-Theta
9) Example 3: Sequential Search
10) Example 4: Maximum Element Problem
11) Example 5: Element Uniqueness Problem
12) Example 6: Binary Digit Counting Problem

1) Algorithms

Definition (Algorithm). A well-defined computational procedure that takes some value, or set of values, as input and produces a value, or set of values, as output.

A sequence of computational steps (instructions) that convert an input to an output.

You can also think of an algorithm as tool for solving a computational problem.

We will often write a computational problem as an input and output relation.

----

2) Example 1: algorithm to count the number of positive values in an array of real numbers

Input: An array A of real numbers
Output: The number of positive values in the array.
Notice that both the inputs and the outputs are clearly described.

Version 1:

```
1       # Input: An array A [0..n-1] of real numbers
2       # Output: Return the number of positive values in the array
3       def CountPositiveElements(A):
4              count = 0
5              for j in range(len(A)):
6                     if A[j]>0:
7                            count = count + 1
8              return count
```

Version 2:

```
1       # Input: An array A [0..n-1] of real numbers
2       # Output: Return the number of positive values in the array
3       def CountPositiveElements(A):
4              count = 0
5              for x in A
6                     if x>0:
7                            count += 1
8              return count
```

We will implement/code all of our algorithms for this class using Python (version 3).

The book's pseudocode is very similar to Python version 3. As an added benefit you will learn Python version 3. It's a very quick language to pick up.

A specific set of inputs for a problem is called an **instance**.

To familiarize yourself with the workings of the algorithm, walk/trace through the algorithm with the problem instances:

<2, 5, -7, 8, 0, 11, -4, -3, 5> and <2.6, -13.45, 16.80, 0, 12.75, -44.4, -78.2> and < -5, -3, 0, -8>

Note: Your version of Python might require arrays and lists to be enclosed in square brackets [ ] rather than the angle brackets < > that we use throughout the class notes for this course.

Does the algorithm **terminate** for all valid inputs? In other words, does its execution conclude?

Yes, the for loop will terminate after examining each entry in the array exactly once.

Is the algorithm correct for all valid inputs?? That is, does it always solve the problem at hand? In fact, we will call an algorithm **correct** if, for every instance, it halts with the correct output.

For this simple problem, it is easy to convince yourself that the algorithm generates the correct answer whether none, some, or all the entries in the array are positive.

Can you come up with another algorithm to solve the same problem? For most problems, there are multiple strategies (algorithms) that will solve the problem.

Notice that by changing the condition we check in step 6, we can count the number of "qualifying" elements in any array. So, this is a template for a more general problem type.

----

3) Example 2: The Greatest Common Divisor (GCD) problem.

Input: Two non-negative integers m > 0 and n > 0

Output: The greatest common divisor, d, of m and n. In other words, d is the largest integer such that there exists two positive integers a and b where m = ad and n = bd. We say that m and n are multiples of d.

There are many algorithms to solve this problem. Here is one very primitive way to find the greatest common divisor of m and n (denoted GCD (m, n)).

```
1       # Input: Two positive integers m and n.
2       # Output: The greatest common divisor of m and n.
3       def GCD(m, n):
4               x = min(m, n)
5               divisor = 1
6               j = 2
7               while j <= x:
8                       if (m%j == 0 and n%j == 0)
9                               divisor = j
10                      j = j+1
11              return divisor
```

Walk through the algorithm with the problem instances GCD (42, 12) and GCD (4, 7).

Does the algorithm terminate for all valid inputs?

Each time through the while loop, the integer j gets closer to x. Since x is a finite number, j will eventually reach and exceed x.

Is the algorithm correct for all valid inputs?

The algorithm works for the two instances we checked, but will it always work. We will defer this analysis for now.

Practice: Can you come up with another algorithm to solve the greatest common divisor problem?

Later in the course, we will revisit this problem and look at the euclidean algorithm for the gcd, attributed to Greek mathematician Euclid.

----

4) Fundamentals of Algorithmic Problem Solving

There is a sequence of steps called algorithmic problem solving that are generally used to solve a problem.
1. Understand the Problem: Be sure to completely understand the problem at hand. Understand its:
– inputs,
– outputs,
– whether or not it is similar to another problem you know how to solve.

2. Determine the Computational Model: What is the computing device allowed to do and not do.
In this class, we will work mostly with the Random Access Machine (RAM) model of computation.
Definition (Random Access Machine (RAM) Model). The RAM is a model of computation where
(a) Instructions are executed sequentially.
(b) Instructions are only those instructions "commonly" found in hardware.
*   Basic arithmetic: add, subtract, multiply, divide, modulus, floor, ceiling on fixed size numbers
*   Memory access: load, store, and copy
*   Basic comparisons: <, ≤, >, ≥, ==, !=
*   Basic control structures: loops and conditionals
*   Every number is represented using $c \lg n$ bits for some $c \geq 1$
*   Memory hierarchy is ignored (for those of you who have taken Computer Architecture).

3. Choose between an exact or approximate solution
We may want to approximate a solution when we can't always give the best answer. For example, computing square roots is generally an approximate algorithm. (So are raking leaves and shoveling snow.)

4. Select a design technique: This course (and the next) will teach you several design techniques or generalized strategies to employ to develop the solution to a problem.

5. Design and Specify the algorithm: Once the design technique is chosen, employ ingenuity, along with data structures and pseudo-code to capture the solution.

6. Prove Correctness: We already briefly mentioned correctness. To prove correctness for exact algorithms we employ a mathematical proof of the fact.
– Generally, this involves induction. Specifically through the use of Loop Invariants
– A loop invariant is generally used to understand why a specific loop correctly computes what is intended.
– To prove a loop invariant, we need to prove three things
(a) Initialization: Invariant is true prior to the first iteration.
(b) Maintenance: If the invariant is true before an iteration of the loop, it is true before the next iteration. (Generally, this is the step the requires induction.)
(c) Termination: When the loop terminates, the invariant gives us a useful property to show the algorithm is correct.
For approximate algorithms we generally look at bounding the error.

7. Analyze: We analyze several types of efficiency in this class.
Time Efficiency: How much time does it take to solve any instance of size n, for any n?
– Space Efficiency: How much space does it take to solve any instance of size n, for any n?
– Unmeasurable Characteristics
*   Simplicity: Make the algorithm simple to understand and implement. This is not always possible to do with good Time or Space efficiency.
*   Generality: Strive to be general both in the problem the algorithm solves and the inputs the algorithm takes.

8. Code: Implement and test. This is by-and-large the subject of other courses, though we will implement algorithms in this course.

During this course we will also be concerned about optimality of an algorithm.
Optimality is all about the complexity of a problem. In other words what is the minimum amount of work we need to do to solve any instance of the problem.
– This is not known for many problems. We do however, have a bound for the sorting problem. More on this later in the course!
----

5) Important Problem Types

While there are several problem types investigated in computer science, perhaps the most common are:
1. Sorting – rearrange a set of values into ascending/descending/alphabetical order

        Given 84.7, 0.0, -62.3, -15.8, 91.0, 66.4, 76.1

        produce the sorted list -62.3, -15.8, 0.0, 66.4, 76.1, 84.7, 91.0

2. Searching – find the first occurrence of a searchkey in a sequence of objects

        Find 17 in the sequence 24, 38, 11, 89, 75, 94, <u>17</u>, 80, 62, 77

3. String Processing – find first occurrence of a substring in a larger string

        Find "A B C" in "A B A C B B C A C B A <u>A B C</u> C B A"

4. Graph Problems – find the shortest path using available edges from one node to another

        Find the shortest route from home to work using public roads

5. Combinatorial Problems

        Determine the number of ways to order a dozen donuts at Dunkins.

6. Geometric Problems

        Given a set of points on a two-dimensional surface, determine which line segments
        joining which pairs of points will form an outline that encloses all other points.

7. Numerical Problems

        Determine the root of a polynomial (where its graph crosses the x-axis) to three decimal
        places.

----

6) Fundamental Data Structures

Data structures are extremely important in the study of algorithms.

Definition (Data Structure). A particular scheme for organizing related data items.

Algorithm efficiency is partly dictated by the way in which data is organized and accessed.

We will briefly review material about data structures.

In this course, we will use

- Arrays/vector

- Matrix

– Tree

– Graph

- Linked list

– Stack

- Queue

- Set

– Dictionary

7) Essential Mathematical Ideas

A) Rounding – floor, ceiling, nearest

$\mathbb{R}$ = Real number—positive, negative, zero, fractions, decimals, whole numbers, rational, irrational

$$2, -7, 0, \frac{2}{3}, 2.74, \pi, \sqrt{3}$$

$\mathbb{Z}$ = Integers—positive, negative, zero, whole numbers

$$..., -3, -2, -1, 0, 1, 2, 3, ...$$

Zahl = German word for whole number

A1. Round down

Floor, truncate, round down, $\lfloor x \rfloor$ or math.floor(x) in Python

Gives the "largest integer not greater than x" (x can be real or integer)

For positive numbers we "round down" to the next lower integer if x is not already an integer (i.e., throw away the decimal places)

Examples: $\lfloor 7.42 \rfloor$ = 7   math.floor (14.735) = 14

$\lfloor 91 \rfloor$ = 91   math.floor (9.5) = 9

A2. Round up

Ceiling, round up, $\lceil x \rceil$, or   math.ceil (x) in Python

Gives the "smallest integer not less than x"

For positive numbers, we "round up" to the next higher integer if x is not already an integer

Examples: $\lceil 7.42 \rceil$ = 8,   math.ceil(14.735) = 15

$\lceil 91 \rceil$ = 91,   math.ceil (9.5) = 10

A3. Round off

[x], or round (x) in Python

Rounds to the nearest integer

If the decimal portion is $\geq$ .5, round up, otherwise round down.

Look at only the first digit after decimal.

Examples: [7.42] = 7   round (14.735) = 15

[91] = 91   round (9.5) = 10

B) Integer arithmetic - mod and div

B1. Integer division, floor division

div, // in python

For positive integers a and b, $a // b$ returns the integer quotient when a is divided by b.

B2. Modulo operator, mod operator

mod, % in Python

For positive integers a and b, $a \% b$ returns the remainder when a is divided by b.

Example: 29/8 = 3 with remainder 5 because 29 = (8 * 3) + 5

so, 29//8 = 3 and 29%8 = 5

Since 29/8 = 3.625, which rounds down to 3, // is also called "floor division"

a//b = math.floor (a/b)

The remainder $a \% b$ will always be in the range 0,1,2,…b-1. It cannot be $\geq b$.

C) Powers/Exponents

a^n

power, exponent

base
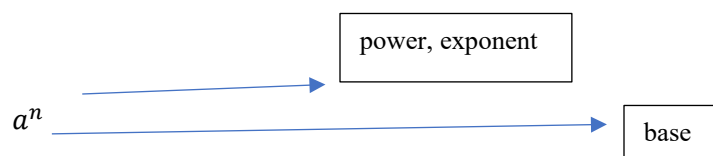
C1. A positive integer exponent means repeated multiplication

$2^3 = 2 * 2 * 2 = 8$

3 factors of 2

$$2^5 = 2 * 2 * 2 * 2 * 2 = 32$$

5 factors of 2

$$2^{10} = 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 = 1,024$$

10 factors of 2

$$2^1 = 2$$

$$2^0 = 1 \text{ (We will see why soon)}$$

$$10^3 = 10 * 10 * 10 = 1,000$$

Note: $2^{10} \approx 10^3$ (approximately equal)

C2. A negative exponent means "in the denominator"

$$2^{-3} = \frac{1}{2^3} = \frac{1}{(2 * 2 * 2)} = \frac{1}{8} = .125$$

$$2^{-5} = \frac{1}{2^5} = \frac{1}{2 * 2 * 2 * 2 * 2} = \frac{1}{32} = .03125$$

$$2^{-10} = \frac{1}{2^{10}} = \frac{1}{1024} \approx .001$$

C3. $a^m * a^n = a^{m+n}$

C4. $\frac{a^m}{a^n} = a^{m-n}$

Why are these rules true? (Some motivation, not a real mathematical proof.)

$$2^3 * 2^4 = (2 * 2 * 2)(2 * 2 * 2 * 2) = 2^7$$

3 factors   4 factors

$$3 + 4 = 7 \; factors \; of \; 2$$

$$10^7 * 10^9 = 10^{7+9} = 10^{16}$$

$$\frac{2^9}{2^4} = \frac{2*2*2*2*2*2*2*2*2}{2*2*2*2} = 2^5 = 2^{9-4} \text{ after cancelling common factors}$$

$$\frac{10^8}{10^6} = \frac{10*10*10*10*10*10*10*10}{10*10*10*10*10*10} = 10^2 = 10^{8-6} = 100$$

$$\frac{3^4}{3^4} = \frac{3*3*3*3}{3*3*3*3} = \frac{1}{1} = 1 = 3^{4-4} = 3^0$$

$$\frac{5^3}{5^7} = \frac{5*5*5}{5*5*5*5*5*5*5} = \frac{1}{5^4} = \frac{1}{625} = 5^{3-7} = 5^{-4}.0016$$

C5. $(a^m)^n = a^{m-n}$

Why? (Again, some motivation not a proof.)

$(10^2)^3 = (10^2)*(10^2)*(10^2) = 10^6$ = 1,000,000

$(2^5)^7 = 2^{5*7} = 2^{35}$

C6. $\left(\frac{a}{b}\right)^n = \frac{a^n}{b^n}$

C7. $(a*b)^n = a^n * b^n$

$$\left(\frac{2}{7}\right)^4 = \frac{2}{7}*\frac{2}{7}*\frac{2}{7}*\frac{2}{7} = \frac{2^4}{7^4} = \frac{16}{2401}$$

$(5n)^3 = (5n)(5n)(5n) = 5*5*5*n*n*n = 5^3 * n^3 = 125n^3$

C8. Negative 1 exponent means reciprocal

$$n^{-1} = \frac{1}{n}$$

$(a*b)^{-1} = \frac{1}{a*b}$ or $a^{-1} * b^{-1} = \frac{1}{a} * \frac{1}{b} = \frac{1}{ab}$

C9. Fractional exponents mean roots

$n^{\frac{1}{2}} = \sqrt{n}$ ---Square Root

$n^{\frac{1}{3}} = \sqrt[3]{n}$ ---Cube Root

Note: $n^{\frac{1}{2}} = \sqrt{n}$

because if you "square both sides", $\left(n^{\frac{1}{2}}\right)^2 = n^{\frac{1}{2}\cdot 2} = n^1 = n$ and $\left(\sqrt{n}\right)^2 = \sqrt{n} * \sqrt{n} = n$

D) Logarithms base 2

Fact: Every positive real number can be written as a power of 2.

For example, 8 can be written as 2^3 .

$32 = 2^5$.

$1,024 = 2^{10}$.

$\frac{1}{2} = 2^{-1}$.

$1 = 2^0$.

In general, if n = 2^k, then $k = log_2(n)$, which is also written as *lg(n)* or *lg n*.

In other words, the base-2 logarithm of a positive integer *n* is the power of 2 that "produces" *n*.

A more helpful way to think about base-2 logarithms is, for any positive number n, lg(n) means:

if you start with n, how many times can you divide by 2 before you get to a fraction between 0 and 1?

8/2 = 4, 4/2 = 2 , 2/2 = 1 since we "divided by 2" 3 times, lg(8) = 3.

32 → 16 → 8 → 4 → 2 → 1, since we "divided by 2", 5 times, lg(32) = 5.

Some properties of logarithms that we will use frequently are:

D1. $2^{\lg(n)} = n$ ← This is the equation version of the previous sentence "In other words,…"

D2. $lg(a^b) = b \cdot lg(a)$ ← This is the logarithm version of the exponent formula $(a^m)^n = a^{m \cdot n}$ since

$$2^{b \cdot \lg(a)} = \left(2^{\lg(a)}\right)^b = a^b.$$

D3. $lg(2) = 1$ ← This follows from the fact that $2^1 = 2$

D4. $lg(2^n) = n$ ← This follows from formulas 2 and 3.

D5. $lg(n^k) = k \cdot \lg(n)$← This follows from formula 2.

E) Summations

The uppercase Greek letter sigma (Σ) is used to denote the sum of a sequence of terms.

$$\sum_{i=1}^{n} a_i = a_1 + a_2 + a_3 + \cdots + a_n$$

Often, the general term a_i is a function of the subscript/index *i.*

$$\sum_{i=1}^{6} \frac{1}{i} = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} = 2.45$$

$$\sum_{i=1}^{4} i(i+1) = 1(2) + 2(3) + 3(4) + 4(5) = 2 + 6 + 12 + 20 = 40$$

A different letter (such as j or k) can be used for the index/subscript.

$$\sum_{j=1}^{5} j^2 = 1^2 + 2^2 + 3^2 + 4^2 + 5^2 = 1 + 4 + 9 + 16 + 25 = 55$$

The initial value of the index/subscript can be another number instead of 1.

$$\sum_{i=0}^{4} 2^i = 2^0 + 2^1 + 2^2 + 2^3 + 2^4 = 1 + 2 + 4 + 8 + 16 = 31$$

Certain types of summations occur often enough that we should know the following formulas:

E1. Sums of sums: If the general term of a summation is a sum of terms, we can create separate summations for each term.

$$\sum (a_i + b_i) = \sum a_i + \sum b_i$$

ex.

$$\sum_{j=1}^{n} (j^3 + 5j) = \sum_{j=1}^{n} j^3 + \sum_{j=1}^{n} 5j$$

E2. Sums with constant coefficients: If every term of a summation includes the same constant, we can factor it out in front of the summation.

$$\sum c \cdot a_i = c \sum a_i$$

ex.

$$\sum_{j=1}^{n} 5j^2 = 5 \sum_{j=1}^{n} j^2$$

E3: Sums of constants: If the general term is a constant (k in the following formula), we are simply adding n copies of the constant.

$$\sum_{i=1}^{n} k = k + k + k + \cdots + k = k \cdot n$$

ex.

$$\sum_{i=1}^{n} 3 = 3 + 3 + 3 + \cdots + 3 = 3n$$

E4. Sums of consecutive positive integers:

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

ex.

$$\sum_{i=1}^{10} i = 1 + 2 + 3 + \cdots + 10 = \frac{10(11)}{2} = 55$$

E5. Sums of square of consecutive positive integers:

$$\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

ex.

$$\sum_{i=1}^{10} i^2 = 1^2 + 2^2 + 3^2 + \cdots + 10^2 = \frac{10(11)(21)}{6} = 385$$

E6. Sums of positive integer powers of 2: (Note that the summation starts with i = 0.)

$$\sum_{i=0}^{n} 2^i = 2^0 + 2^1 + 2^2 \ldots + 2^n = 2^{n+1} - 1$$

ex.

$$\sum_{i=0}^{7} 2^i = 2^0 + 2^1 + 2^2 \ldots + 2^7 = 2^8 - 1 = 255$$

E7. Sums of negative integer powers of 2:

$$\sum_{i=1}^{n} 2^{-i} = 2^{-1} + 2^{-2} + 2^{-3} \ldots + 2^{-n} = 1 - 2^{-n}$$

ex.

$$\sum_{i=1}^{5} 2^{-i} = 2^{-1} + 2^{-2} + 2^{-3} \ldots + 2^{-5} = 1 - 2^{-5} = 1 - \frac{1}{32} = \frac{31}{32}$$

E8. At times we change the limits of the summation, such as replacing I by j+1 as in the following example:

$$\sum_{i=1}^{4} i2^{i-1} = \sum_{j=0}^{3} (j+1)2^{j} = 1 \cdot 2^0 + 2 \cdot 2^1 + 3 \cdot 2^2 + 4 \cdot 2^3 = 1 + 4 + 12 + 32 = 4$$

8) Asymptotic Analysis

Asymptotics form the basis of algorithm analysis. In general, the use of asymptotic analysis allows us to reason about the approximate growth of a function as the input, *n*, to that function gets very large. When discussing algorithms, **the function T(n) represents the number of primitive operations that must be performed for an input of size *n*.**

a) How do we measure an input size n? Could be the number of bits. Could be the number of elements in an array. Could be the degree of a polynomial. In any case, it is generally easy to find the size parameter. It is possible to have multiple size parameters. For example, consider a m x n matrix.

b) What do we care about in terms of efficiency? There are two types of efficiency: time and space.
  Time: How much time it takes to for an algorithm to run on an input of size n?
  Space: How much extra space an algorithm needs for an input of size n?
While space complexity is not as important as it once was, we will still consider it though to a less extent than time complexity. Fact: The time to run an algorithm increases as the size of the input increases.

c) What should the units of our running time be? Should they be seconds or minutes? Should they be picoseconds? Do any time units actually make sense? No! The time is dependent on a specific computer; that is not general enough. Instead, let's count the operations on the idealized RAM model of computation we talked about.

d) What operation(s) do we count? The book talks about counting the number of times a basic/primitive operation is executed. A basic operation is an operation that contributes most to the running time (perhaps a comparison of two array element, or a swap of two elements, or popping an object off a stack, or multiplying two numbers, or making a pass through a loop, etc.). This is what we will usually do. However, this requires a lot of intuition to spot. If we do not know what the basic operation should be, we will count all the operations and work on building our intuition.

e) How carefully do we count? Let's say that, for a problem of size n, our algorithm performed $10n^2 + 3n + 4$ operations. If we want to be less formal (which we will usually choose to do), we can ignore constants and only look at the largest term in a count to get the order of growth of the function, so we would have the order of growth n^2. This makes sense as we are really concerned with the behavior of algorithms on large values of n. In other words, what term dominates as $n \rightarrow \infty$? When we perform formal asymptotic analysis, we take great care with the constants.

To see common order of magnitudes behavior, for homework we will complete a table comparing the values of seven commonly occurring functions for the values of n = 1 to 10.

One can easily imagine wanting to sort 100 elements in an array. Do you really want an algorithm that preforms n^2 = 10,000 operations?

f) For what type of situation are we measuring the workload of an algorithm? When we consider time efficiency in terms of n we also look at different types of behaviors. In particular we look at:

1. *Worst-Case Efficiency*: The efficiency for the worst-case input of size n. This is the input of size n for which the algorithm runs the longest (performs the most operations) among all possible inputs of the size in question. This is almost always the type of analysis we will be performing.

2. *Best-Case Efficiency*: The efficiency for the best-case input of size n. This is the input of size n for which the algorithm runs the shortest (performs the least amount of operations) among all possible inputs of the size in question. Generally, we are not so concerned with best-case efficiency.

3. *Average-Case Efficiency*: The algorithms behavior on a typical random input of size n. This type of analysis depends on assumptions about the structure of the input of size n. We will not be performing a lot of this type of analysis. It is more common in graduate school.

4. *Amortized Efficiency*: Look at the efficiency of a sequence of operations performed on a data structure. It is similar to amortizing costs in a business. In other words, you look some operations will be very expensive while others are very cheap. So, if we have few uses of the expensive operation it becomes cheaper to perform over time as the cost is absorbed in to the long sequence of n operations. It is unlikely that we will do any of this form of analysis this semester.

g) Asymptotic Notations and Basic Efficiency Classes
To bound orders of growth, we will use three different notions making up a study known as aysmptotics. We will consider three types of asymptotic notations that can be used to bound T(n).
1. Big-Oh notation   O()
2. Big-Omega notation   Ω()
3. Big-Theta notation   θ()
We will almost always be concerned about Big-Oh.

Big-Oh notation gives us an asymptotic upper bound on a function. Intuitively, we are looking for a function g (n) that is always greater than our function T(n). Formally, we define big-Oh notation as follows:

Definition (Big-Oh Notation). For a given function g (n) we denote by O (g (n)) the set of functions:
$$O\big(g(n)\big) = \{f(n) | \exists c, n_0 > 0 \; such \; that \; 0 \leq f(n) \leq c \cdot g(n), \forall n \geq n_0\}$$
Note, that O (g (n)) is the set of all functions bounded above by some constant multiple of g (n).
Example: If T(n) = $5n^2$ + 3n + 2 then T(n) = O (n^2).
To prove this is true we can simply find a constant c and a positive integer n0,
so that $5n^2$ + 3n + 2 $\leq cn^2$ whenever n ≥ n0.

When n ≥ 1, n^2 ≥ n so $3n^2$ ≥ 3n.
Also, when n ≥ 1, n^2 ≥ 1 so $2n^2$ ≥ 2.
Thus, $5n^2$ + 3n + 2 $\leq 5n^2 + 3n^2 + 2n^2 = 10n^2$ for all n ≥ 1.
Therefore, with n0 = 1 and c = 10, we have shown that T(n) = $5n^2$ + 3n + 2 = O(n^2).

Example: If T(n) = 7n lg n + 3n and we want to show that T(n) = O (n^2).
We must show that there exists a constant c and a positive integer n0,

7n lg n + 3n ≤ cn² whenever n ≥ n0.

For n ≥ 1, lg(n) < n   (We will see this in the homework worksheet.)

so n lg(n) < n²

and 7n lg(n) < 7n²

Also, for n ≥2, n < n²

so 3n < 3n²

Thus, when n ≥2,

 7n lg(n) + 3n < 7n² + 3n²

         7n lg(n) + 3n < 10n²

Therefore, with c = 10 and n0 = 2, we have

        7n lg(n) + 3n < cn² whenever n > n0.

We have then shown that T(n) = 7n lg n + 3n = O (n²).

Big-Ω (Big-Omega) notation gives us an asymptotic lower bound on a function.

Intuitively, we are looking for a function g (n) that is always less than our function T(n)

Formally, we define big-Ω notation as follows

Definition 3 (Big- Notation). For a given function g (n) we denote by Ω(g (n)) the set

of functions:

$$\Omega\big(g(n)\big) = \{f(n)|\exists c, n_0 > 0 \; such \; that \; 0 < c \cdot g(n) \le f(n), \forall n \ge n_0\}$$

Note, that Ω(g (n)) is the set of all functions bounded below by some constant multiple

of g (n).

Example: If T(n) = 5n2 + 3n + 2 then T(n) = Ω(n2).

To prove this is true we can simply find a constant c and a positive integer n0,

so that 5n² + 3n + 2 ≥ cn² whenever n ≥ n0.

We proceed as follows

    2 > 0

When n ≥ 1,   3n > 0

Adding these inequalities,   3n + 2 > 0 + 0

3n + 2 > 0

Add 5n² to both sides of the inequality 5n² + 3n + 2 > 5n² + 0

                                         5n² + 3n + 2 > 5n²

So, with c = 5 and n0 = 1 we have

5n² + 3n + 2 > cn² whenever n > n0.

Thus T(n) = 5n² + 3n + 2 = Ω(n²).

Big-θ (Big-Theta) notation gives us an asymptotic tight bound on a function.

Intuitively, we are looking for a function g (n) that bounds T(n) both above and below.

Formally, we define big-θ notation as follows

Definition (Big-θ Notation). For a given function g (n) we denote by θ(g (n)) the

set of functions:

$$\theta\big(g(n)\big) = \{f(n) | \exists c_1, c_2, n_0 > 0 \text{ such that } 0 < c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n), \forall n \geq n_0\}$$

Note, that $\theta(g(n))$ is the set of all functions bounded above and below by constant multiples of g (n).

Example: If T(n) = 5n2 +3n+2 then T(n) = $\theta(n^2)$. To prove this is true we need to find c1, c2, and n0. One way to solve this problem is to break the inequality in half.
In the discussion of Big-Oh, we showed that $5n^2 + 3n + 2 \leq 10n^2$ for all n ≥ 1.
In the discussion of Big-Omega, we showed that $5n^2 + 3n + 2 > 5n^2$ whenever n ≥ 1.
We can rewrite this as $5n^2 < 5n^2 + 3n + 2$ for all n ≥ 1.
Therefore, we have $5n^2 \leq 5n^2 + 3n + 2 \leq 10n^2$ for all n ≥ 1
and T(n) = 5n2 +3n+2 = $\theta(n^2)$.
If the two values of n0 had been different, we would use the larger one.

This hints at the following theorem
Theorem. T (n) = $\theta(g(n))$ if and only if T (n) = O (g (n)) and T (n) = $\Omega(g(n))$.

The "if and only if" expression means there are really two statements here:
If T (n) = $\theta(g(n))$ then T (n) = O (g (n)) and T (n) = $\Omega(g(n))$.
If T (n) = O (g (n)) and T (n) = $\Omega(g(n))$ then T (n) = $\theta(g(n))$.
We just used the second of these statements to show that 5n2 +3n+2 = $\theta(n^2)$.

As you get more sophisticated with your analysis, another useful theorem when dealing with asymptotics is the following theorem about consecutive execution of algorithms.
Theorem. If T1 (n) = O (g1 (n)) and T2 (n) = O (g2 (n)), then T1 (n) + T2 (n) =
    O(max {g1 (n) ; g2 (n)}).
For example, if T1 (n) = O (n^2) and T2 (n) = O (n^3), then T1 (n) + T2 (n) = O (max {n^2; n^3}), which is O (n^3). The theorem also holds when dealing with Ω and θ.

h) Classes Here is a table of the common efficiency classes we will "bump into" during the course.

Class	Name
1	constant (algorithms that perform a fixed number of operations for all n)
$\lg(n)$	logarithmic
n	linear
$n\lg(n)$	linear-logarithmic
n^2	quadratic
n^3	cubic
2^n	exponential
$n!$	factorial

i) Mathematical Analysis of Non-recursive Algorithms

We are going to do a bunch of examples to apply the methods of analysis we learned.
Our text has the following generic process, I won't be strictly following all the of the steps as it relies on intuition in some parts. As you get more comfortable with analysis you will find it easier to rely on your intuition.
1. Decide on a parameter (or parameters) indicating the inputs size.
2. Identify the algorithms basic operation. Most likely in the innermost loop. This is an intuition step.
3. Check whether the number of times the basic operation is executed is only dependent on the size of input n. If not, your best-case, worst-case, and average-case analysis will all be different.
4. Set up a sum expressing the number of times the basic operation is executed.
5. Using standard formulas and rules for sum manipulation, either find a closed form of the sum, or establish an order of growth.

9) Example 3: Sequential Search

We have looked at a lot of definitions and discussed a lot of pros and cons. Let's look at a simple
problem and go through all of our steps of determining the work done in the worst case.
The problem we are solving is the search problem
Definition 1 (Search Problem).
Input: A sequence of n keys A = <a0, a1, …, an-1> and a search key k.
Output: The index i, of k in the input sequence if there exists an ai = k or nil otherwise.
In our case our sequence will be an array A.
What's the most basic algorithm for searching for a key k in array A?
Our good friend the sequential search.
The linear search algorithm is given as follows:

```
1 # Input: An array A [0..n-1], a search key k
2 # Output: Return the index of the first element in A
3          that matches search key k or None if there is no
4          matching element.
5 def SequentialSearch(A, k):
6      for i in range(len(A)):
7          if A[i] == k:
8              return i
9      return None
```

Using the formal strategy from the book, what is our input size?
Answer: n.

What's the worst-case running time?
Informally, we can identify the basic operation to be the comparison in line 7. Since this is performed
once on each of the n passes through the for loop in line 6, the work is O(n). Done.

To compute this formally, we would determine the cost for each
line and number of times each line executes. The results are in the table below.

Line	Cost	Count
6	c_1	n
7	c_2	n
8	c_3	1
9	c_4	1 (in the worst case)

We then total the cost multiplied by the count for each line. Since only one of line 8 or 9 executes (never
both), we really only need to add one of the constants, c3 or c4.

$T(n) = c_1 n + c_2 (n) + max(c_3 + c_4)$

= (c1 + c2)n + max(c3 + c4)

= c5n + c6   Collapse constants into new constants.

≤ c5n + c6n   for n ≥ 1

= c7n   where c7 = c5 + c6

= O(n)

What about a lower bound for T(n)?

We have T(n) = c1n + c2 (n) + max(c3 + c4)

Since all terms are ≥ 0 when n ≥1, we can drop all terms except c1 n to obtain

T(n) = c1n + c2 (n) + max(c3 + c4) ≥ c1n

So, T(n) = Ω(n)

Since, T(n) = O(n) and T(n) = Ω(n), we actually have T(n) = θ(n).

10) Example 4: Maximum Element Problem

The maximum element problem is:
Definition (Maximum Element Problem).
Input: An array of n elements A = <a1, a2, …, an>.
Output: An element aj in A such that aj ≥ai for all ai in A

How might you solve this problem?
The solution I want to consider is the natural algorithm below
The intuition is that we walk left to right through the list and update the current maximum element we have seen. At the end of the loop we return the maximum element.

```
1       # Input: An array A of 1 or more integers.
2       # Output: Return the largest element in A, A[i].
3       def MaxElement(A):
4               largest = A[0]
5               for i in range(1, len(A)):
6                       if A[i] > largest:
7                               largest = A[i]
8               return largest
```

Using the formal strategy from the book, what is our input size?
Answer: n.

Informally, we can identify the basic operation to be the comparison in line 6. Since this is performed once on each of the n-1 passes through the for loop in line 5, the work is n-1, which is <n, so O(n). Done.

Formally, let's build a table for the cost and count of each line in our algorithm (i.e. our time T (n)).

Line	Cost	Count
4	c_1	1
5	c_2	n
6	c_3	n - 1
7	c_4	n - 1 (in the worst case)
8	c_5	1

What is our sum?
$T(n) = c_1 + c_2 n + c_3 (n - 1) + c_4 (n - 1) + c_5$.

Let's determine the asymptotic growth of our sum.
$T(n) = c_1 + c_2 n + c_3 (n - 1) + c_4 (n - 1) + c_5$
$= c_1 + (c_2 + c_3 + c_4) n - c_3 - c_4 + c_5$
$= c_6 + c_7 n$ where $c_6 = c_1 - c_3 - c_4 + c_5$ and $c_7 = c_2 + c_3 + c_4$
$\leq c_6 n + c_7 n$
$= c_8 n$ where $c_8 = c_6 + c_7$

= O(n)

What about a lower bound for T(n)?

We have $T(n) = c_1 + c_2n + c_3(n - 1) + c_4(n - 1) + c_5$

Since all terms are ≥ 0 when $n \geq 1$, we can drop all terms except $c_2 n$ to obtain

$T(n) = c_1 + c_2n + c_3(n - 1) + c_4(n - 1) + c_5 \geq c_2n$

So, $T(n) = \Omega(n)$

Since, $T(n) = O(n)$ and $T(n) = \Omega(n)$, we actually have $T(n) = \theta(n)$.

11) Example 5: Element Uniqueness Problem

Formally, the problem is:

Definition 6 (Element Uniqueness Problem).

Input: A set of n elements A = <a1, a2, …. an>.

Output: True if every element in A is unique. An element ai is unique if ai ≠ aj for all j ≠ i.

Can you think of a way to solve it?

Here is one solution.

The intuition is that we work left to right comparing an element to all elements that follow it. We do not need to worry about the elements that precede the current element as these have already been checked (think induction).

```
1       # Input: An array A if integers.
2       # Output: True is elements are unique; otherwise, False.
3       def UniqueElements(A):
4               for i in range(0, len(A)-1):
5                       for j in range(i+1, len(A)):
6                               if A[i] == A[j]:
7                                       return False
8               return True
```

Using the formal strategy from the book, what is our input size?

Answer: n.

Informally, we can identify the basic operation to be the comparison in line 6. Because we have nested loops, that cause the inner loop to run one less time on each initiation. Therefore, this operation will be performed (n-1) + (n-2) + (n-3) + …. + 3 +2 +1 = n(n-1)/2 times = $\frac{1}{2}n^2 - \frac{1}{2}n$. Focusing on this highest-powered term and ignoring its coefficient, we get O(n²). Done.

Formally, we would build a table for the cost and count of each line in our algorithm (i.e. our time T (n)).

Line	Cost	Count
4	c1	n - 1
5	c2	$\sum_{i=0}^{n-2}(n-1-i)$ = (n-1) + (n-2) + (n-3) + … + 1
6	c3	$\sum_{i=0}^{n-2}(n-1-i)$ = (n-1) + (n-2) + (n-3) + … + 1
7	c4	1
8	c5	1

T(n) = c1(n-1) + c2$\sum_{i=0}^{n-2}(n-1-i)$ + c3$\sum_{i=0}^{n-2}(n-1-i)$ + c4 + c5

What is the summation that occurs in steps 5 and 6?

After completing the homework, we will see that this is $\frac{(n-1)n}{2}$.

Therefore, T(n) = c1(n-1) + c2$\frac{(n-1)n}{2}$ + c3$\frac{(n-1)n}{2}$ + c4 + c5

= c1n − c1 + .5c2n² - .5c2n + .5c3n² - .5c3n + c4 + c5

27

$= (.5c_2 + .5c_3)n^2 + (c_1)n + (-c_1 + c_4 + c_5)$

$= c_6n^2 + c_1n + c_7$

$= \theta(n^2)$

12) Example 6: Binary Digit Counting Problem

The binary digit counting problem is formally defined as:
Definition (Binary Digit Counting Problem).
Input: An integer n ≥ 0.
Output: The b the number of bits in the binary expansion of n.
How might you solve this?
Intuitively, I would count the number of powers of two in the number n.
Here is one formal algorithm that solves the problem.

```
1       # Input: n a non-negative integer.
2       # Output: Return the number of bits in the
3       #       binary expansion of n.
4       def BinaryDigitCount(n):
5               count = 1
6               while n>1:
7                       count = count+1
8                       n = math.floor(n/2)
9               return count
```

Note: In some versions of Python, whenever you use a function from the math library (such as math.floor), you need to import the math library by including the statement
        import math
in your code.

Using the formal strategy from the book, what is our input size?
Answer: n.

Informally, we can identify the basic operation to be the floor operation in line 8 because this is the operation that determines the number of passes through the loop. On each pass, we divide the current vale of n by 2 (and throw away any fractional part, since this is integer arithmetic). How many times can we divide a positive number by 2? This is precisely the definition of the base-2 logarithm of n, or $\lg(n)$. Therefore, our algorithm is $O(\lg n)$. Done.

If we want to go through the formal process, we would build a table for the cost and count of each line in our algorithm (i.e. our time $T(n)$).

Line	Cost	Count
5	c_1	1
6	c_2	t
7	c_3	$t - 1$
8	c_4	$t - 1$
9	c_5	1

What is t?
It depends on the value that n is currently (i.e. the loop control variable).

What part of the loop updates the loop control and how is it updated?

Notice that each time through the loop, n is divided by two. Repeating the process t times results in dividing n by the t-th power of two.

The loop terminates when $\frac{n}{2^t} = 1$

So what is t?

Let's use those logarithms!

$\frac{n}{2^t} = 1$

$n = 2^t$

$\lg(n) = t$

Technically, we do not require that n be an exact power of two so t = floor(lg(n)) +1.

Notice floor(lg(n)) +1 = θ(lg n). We will therefore use t = lg n in practice.