CS5132
DATA STRUCTURES AND ALGORITHMS

NUS HIGH SCHOOL of Math & Science

Name:_____ Date: _____

CHAPTER 1: ANALYSIS OF ALGORITHMS

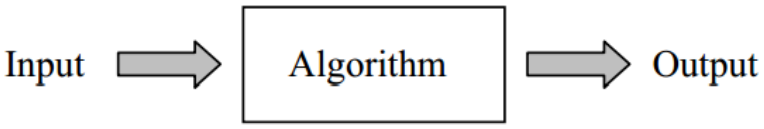
We are interested in the design of "good" data structures and algorithms. A **data structure** is a systematic way of organizing and accessing data, and an **algorithm** is a step-by-step procedure for performing some given task in a finite amount of time.

In algorithmic problem solving, we deal with objects. Objects are data manipulated by the algorithm. To a cook, the objects are the various types of vegetables, meat and sauce. In algorithms, the data are numbers, words, lists, files, and so on. In solving a geometry problem, the data can be the length of a rectangle, the area of a circle, etc. Algorithm provides the logic; data provide the values. They go hand in hand. **Hence, we have this great truth:**

Program = Algorithm + Data Structures

1.1 What is an Algorithm?

There are many definitions of algorithms. An algorithm is a well-defined computational procedure consisting of a set of instructions, that takes some value or set of values, as input, and produces some value or set of values, as output. In other words, an algorithm is a procedure that accepts data, and manipulates them following the prescribed steps, to eventually fill the required unknown or output with the desired value(s) produced from the procedure.



There are many different ways to express an algorithm, including natural language, pseudocode, flowcharts, and programming languages. Natural language expressions of algorithms tend to be verbose and ambiguous, hence rarely used for complex or technical algorithms. Pseudocode and flowcharts are structured ways to express algorithms that avoid many of the ambiguities common in natural language statements, while remaining independent of a particular implementation language. Programming languages are primarily intended for expressing algorithms in a form that can be executed by a computer, but are often used as a way to define or document algorithms.

For example: Pseudocode for linear search algorithm

```
For each item in the list:
     If item = value
         Stop search and return the item's location.
     End If
End For
```

Types of Algorithms

Algorithms that use a similar problem-solving approach can be grouped together.

Algorithm Types	General Outline of Algorithm	Specific Examples
Simple recursive algorithms	○ Solves the base cases directly. ○ Recurs with a simpler sub-problem. ○ Does some extra work to convert the solution to the simpler subproblem into a solution to the given problem. ○ Note that several of the other algorithm types are inherently recursive	Factorial function
Backtracking algorithms	○ Based on a depth-first recursive search ○ Tests to see if a solution has been found, and if so, returns it. Otherwise: ○ For each choice that can be made at this point, ▪ Make that choice, ▪ Recur, ▪ If the recursion returns a solution, return the solution. ○ If no choices remain, return failure	N-Queens problem
Divide and Conquer algorithms	○ Divide the problem into smaller subproblems of the same type, and solve these subproblems recursively ○ Combine the solutions to the subproblems into a solution to the original problem	Quicksort Mergesort
Brute Force algorithms	○ Tries *all* possibilities until a satisfactory solution is found.	Finding Fibonacci numbers
Dynamic Programming algorithms	○ Remembers past results (“memoization”) and uses them to find new results	
Greedy algorithms	○ An optimization problem is one in which you want to find, not just *a* solution, but the *best* solution ○ A “greedy algorithm” sometimes works well for optimization problems ○ A greedy algorithm works in phases: At each phase: ▪ You take the best you can get right now, without regard for future consequences ▪ You hope that by choosing a *local* optimum at each step, you will end up at a *global* optimum	Gradient / Steepest Descent algorithm in AI
Randomized algorithms	○ A randomized algorithm uses a random number at least once during the computation to make a decision ○ E.g: In Quicksort, using a random number to choose a pivot	Using a random number to choose a pivot in Quicksort Factor a large number by choosing random numbers as divisors

What makes a "good" algorithm?

There are three desirable properties of a good algorithm. We seek algorithms that are **correct** and **efficient**, while being **easy to implement**.

Program efficiency: time vs. space. It is interesting to know how much of a particular resource (such as time or storage) is required for a given algorithm. Methods have been developed for the **analysis of algorithms** to obtains such quantitative answers, such as the big O notation. For example, the time needed for traversing an array of *n* slots is proportional to *n*, and we say the time is in the order of $O(n)$. However, accessing the *i* th element in an array takes only constant time, which is independent of the size of the array, thus is in the order of $O(1)$.

The computational complexity and efficient implementation of the algorithm are important in computing, and this depends on suitable data structures.

To classify some data structures and algorithms as "good", we must have precise ways of analyzing them. Analyzing the efficiency of a program involves characterizing the **running time** and **space usage** of algorithms and data structure operations. Particularly, the running time is a natural measure of goodness, since time is precious.

1.2 Running Time

Most algorithms transform input objects into output objects. The running time of an algorithm or a data structure method typically grows with the input size, although it may also vary for different inputs of the same size. Also, the running time is affected by a lot of factors, such as the hardware environment and the software environment.

Using exact run time is not meaningful when we want to compare two algorithms,

- coded in different languages,
- using different data sets, or
- running on different computers.

Algorithm analysis should be independent of ,

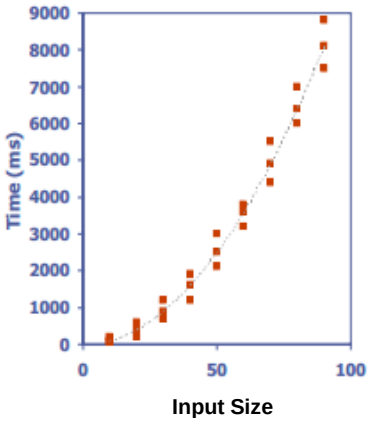
- Specific implementations
- Compilers and their optimizers
- Computers

But what is the proper way of measuring it?

- Experimental analysis
- Theoretical analysis

Experimental Studies

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Measure the runtime using time
- Plot the results



Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult
- Good range of inputs?
- To compare two algorithms, the same hardware and software environments must be used

Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, *n*.
    - Look at large input sizes
- Accounts for all possible inputs
- Evaluates algorithm independent of hardware, implementation, input set, etc.
- Count operations not actual clock time

We would like to focus on **the relationship between the running time of an algorithm and the size of its input**. In the end, we would like to characterize an algorithm's running time as a function $f(n)$ of the input size n.

Exact Values?

It is sometimes possible, *in assembly language,* to compute *exact* time and space requirements. We know exactly how many bytes and how many cycles each machine instruction takes. For a problem with a known sequence of steps (factorial, Fibonacci), we can determine how many instructions of each type are required.

However, often the exact sequence of steps cannot be known in advance. The steps required to sort an array depend on the actual numbers in the array (which we do not know in advance)

In a higher-level language (such as Java), we *do not know* how long each operation takes

- Which is faster, x < 10 or x <= 9?
- We don't know exactly what the compiler does with this
- The compiler probably optimizes the test anyway (replacing the slower version with the faster one)

In a higher-level language we *cannot* do an exact analysis.

- Our timing analyses will use *major* oversimplifications
- Nevertheless, we can get some very useful results

We may analyse time efficiency by determining the number of repetitions of the basic operation as a function of input size.

Basic or characteristic operation: the operation that contributes most towards the running time of the algorithm.

- In computing time complexity, one good approach is to count characteristic operations
- What a "characteristic operation" is depends on the particular problem"
    - If searching, it might be comparing two values
    - If sorting an array, it might be:
        - comparing two values
        - swapping the contents of two array locations
        - both of the above
- Sometimes we just look at how many times the *innermost loop* is executed

Primitive operations are basic computations performed by an algorithm. Examples are evaluating an expression, assigning a value to a variable, indexing into an array, calling a method, returning from a method, etc. They are easily identifiable in pseudocode and largely independent from the programming language.

By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm as a function of the input size.

For example, determine the number of instructions as a function of input size N for the following code in the worst case **(Write down the respective frequencies in the table below)**:

```
int count = 0;
for (int i = 0; i < N; i++)
   if (a[i] == 0)
      count++;
```

Code	No. of primitive operations
variable declaration	
assignment statement	
less than compare	
equal to compare	
array access	

Define **T(*n*) to be the running time of the algorithm** as a function of the size of the problem (*n*) and the time it takes to solve the said problem.

As T(*n*) is directly proportional to the number of operations required, C(*n*), we can write the relation as follows:

$T(n) \approx c_{op}\ C(n)$

- $T(n)$: running time
- c_{op}: execution time for basic operation
- $C(n)$: Number of times basic operation is executed

Let *a* = Time taken by the fastest primitive operation.
Let *b* = Time taken by the slowest primitive operation.

Hence we have, $a(5n + 5) \leq T(n) \leq b(5n + 5)$, and we say T(*n*) is bounded above and below by two linear functions. Hence, we can say T(*n*) **grows linearly** with respect to *n*.

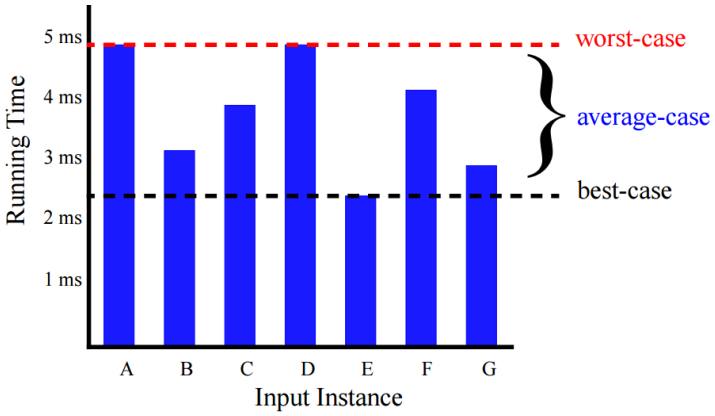
Changing the hardware/software environment will not affect the ***growth rate*** of T(*n*).

1.3 Focusing on the Worst Case

Think about the example of a linear search on an array. What is the best-, average-, and worst-case performance?

An algorithm may run faster on some inputs than it does on others of the same size. Thus, we wish to express the running time of an algorithm as a function of the input size, by taking the average over all possible inputs of the same size. However, an **average case** analysis is typically challenging.

An average case analysis requires that we calculate expected running times based on a given input distribution, which involves probability theory. In real-time computing, the **worst-case** analysis is of concern as it is important to know how much time is needed in the worst case to guarantee that the algorithm always finishes on time. The **best-case** analysis is used to describe the way an algorithm behaves under optimal conditions.



A worst-case analysis is much easier than an average case analysis, as it requires only the identification of the worst-case input. This approach typically leads to better algorithms. Making the standard of success for an algorithm to perform well in the worst case necessarily requires that it will do well on every input.

1.4 Common Mathematical Functions Used Worst Case Analysis

The Constant Function f(n) = C

For any argument *n*, the constant function *f*(*n*) assigns the value *C*. It does not matter what the input size *n* is, *f*(*n*) will always be equal to the constant value *C*. The most fundamental constant function is *f*(*n*) = 1, and this is the typical constant function that is used in the book.

The constant function characterizes the number of steps needed to do a basic operation, like adding two numbers, assigning a value to some variable, or comparing two numbers. Executing one instruction a fixed number of times also needs constant time.

An algorithm which runs on ***constant*** time does not depend on the input size.
Examples: arithmetic calculation, comparison, variable declaration, assignment statement, invoking a method or function.

The Logarithm Function f(n) = log n

It is one of the interesting and surprising aspects of the analysis of data structures and algorithms. The general form of a logarithm function is $f(n) = \log_b n$, for some constant $b > 1$. This function is defined as follows: $x = \log_b n$, if and only if $b^x = n$.

The value *b* is known as the **base** of the logarithm. Computing the logarithm of any integer *n* is not always easy, but we can easily compute the smallest integer greater than or equal to $\log_b n$. This number is equal to the number of times we can divide *n* by *b* until we get a number less than or equal to 1. For example, $\log_3 27$ is 3, since 27/3/3/3 = 1. Likewise, $\log_2 12$ is approximately 4, since 12/2/2/2/2 = 0.75 <= 1.

The base-two approximation arises in algorithm analysis, since common to many algorithms is to repeatedly divide an input in half. In fact, the most common base for the logarithm in computer science is 2. We typically leave it off when it is 2.

An algorithm which runs on ***logarithmic*** time becomes slightly slower as *n* grows.
Whenever *n* doubles, the running time increases by a constant.
Example: Binary search.

The Linear Function f(n) = n

Another simple yet important function. This function arises in algorithm analysis any when we do a single basic operation for each of *n* elements.

For example, comparing a number *x* to each element of an array of size *n* will require *n* comparisons. The linear function also represents the best running time we hope to achieve for any algorithm that processes a collection of *n* inputs.

An algorithm which runs on ***linear*** time is proportional to the growth of *n*
Whenever *n* doubles, so does the running time proportionately.
Example: print out the elements of an array of size *n*.

The N-Log-N Function f(n) = nlogn

This function grows a little faster than the linear function and a lot slower than the quadratic function (n^2). If the run-time of an algorithm can be improved from quadratic to N-Log-N, it runs much faster in general.

An algorithm which runs on ***nlogn*** time runs slower than linear time but faster than quadratic time (n^2).
Whenever *n* doubles, the running time will more than double.
Examples: Merge sort (which will be discussed later in the module)

The Quadratic Function f(n) = n^2

It frequently appears in algorithm analysis, since there are many algorithms that have nested loops, where the inner loop performs a linear number of operations and the outer loop is performed a linear number of times. In such cases, the algorithm performs $n*n = n^2$ operations.

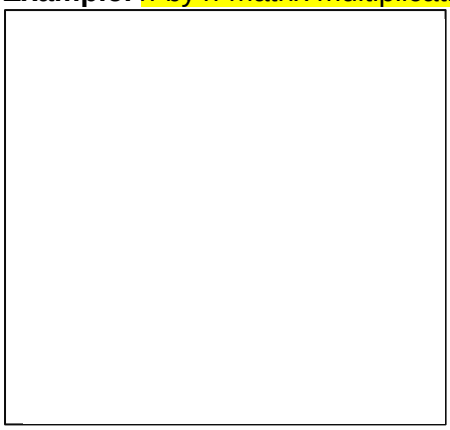
The quadratic function can also be used in the context of nested loops where the first iteration of a loop uses one operation, the second uses two operations, the third uses three operations, and so on. That is, the number of operations is $1 + 2 + 3 + ... + (n-1) + n = n*(n+1) / 2$.

An algorithm that runs on quadratic time are practical for relatively small problems.
Whenever *n* doubles, the running time increases fourfold.
Example: Performing manipulations on the *n* by *n* array.

The Cubic Function and Other Polynomials

The cubic function $f(n) = n^3$, appears less frequently in the context of the algorithm analysis than the constant, linear, and quadratic functions. It's practical for use only on small problems.

Whenever *n* doubles, the running time increases eightfold.
Example: *n* by *n* matrix multiplication.



The functions we have learned so far can be viewed as all being part of a larger class of functions, the **polynomials**. A polynomial function is a function of the form:

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0.$$

where a_0, a_1, ... , a_n are constants, called the coefficients of the polynomial, where a_n is not 0. The integer *n*, which indicates the highest power in the polynomial, is the **degree** of the polynomial.

The Exponential Function f(n) = b^n

In this function, *b* is a positive constant, called the **base**, and the argument *n* is the **exponent**. In algorithm analysis, the most common base for the exponential function is $b = 2$. For instance, if we have a loop that starts by performing one operation and then doubles the number of operations performed with each iteration, then the number of operations performed in the *n*th iteration is 2^n.

An algorithm that runs on ***exponential*** time is usually not appropriate for practical use.
Example: Towers of the Hanoi.

The Factorial Function f(n) = n!

An algorithm that runs on ***factorial*** time runs even worse than exponential time.
Whenever *n* increases by 1, the running time increases by a **factor of *n***.

Example: Permutations of *n* elements.

Comparing Growth Rates

Ideally, we would like data structure operations to run in times proportional to the constant or logarithm function, and we would like our algorithms to run in linear or *n*-log-*n* time. Algorithms with quadratic or cubic running times are less practical, but algorithms with exponential running times are infeasible for all but the smallest sized inputs.

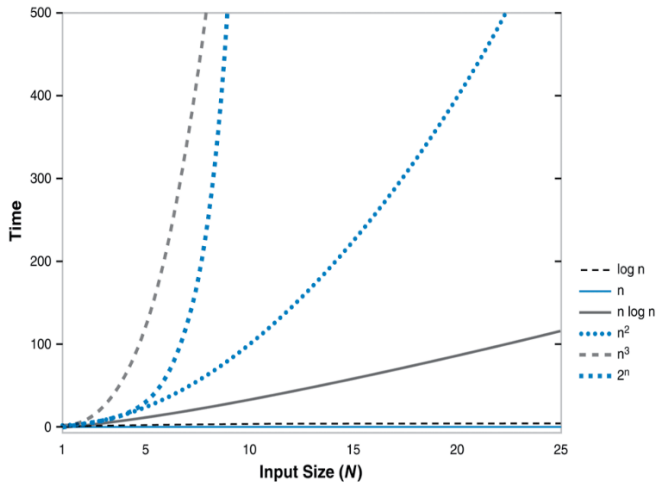


FIGURE 2.4 Comparison of typical growth functions for small values of n

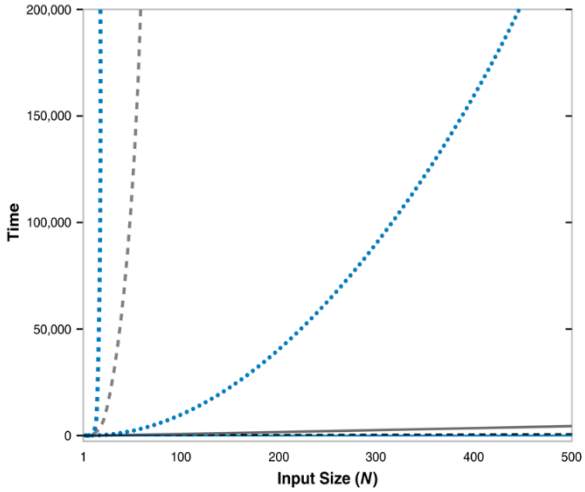
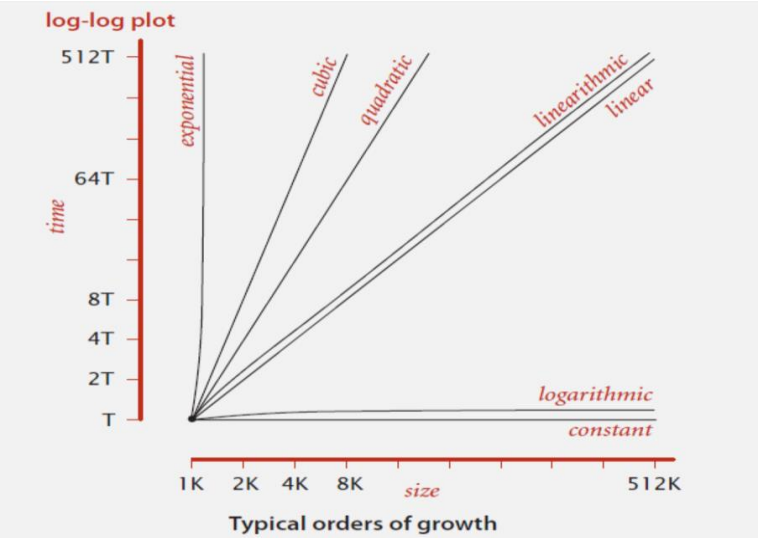


FIGURE 2.5 Comparison of typical growth functions for large values of n

n $f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

Growth rates of common functions measured in nanoseconds



Order of growth of some common functions:

O(1) < O(log *n*) < O(*n*) < O(*n* log *n*) < O(n^2) < O(n^3) < O(2^n)

1.5 Asymptotic Notation

In the algorithm analysis, we focus on the growth rate of the running time as a function of the input size *n*, taking a "**big-picture**" approach. It is often enough to know that the running time of an algorithm such as a linear search on an array **grows proportionally to *n***, with its true running time being *n* times a constant factor that depends on the specific computer.

We analyze algorithms using a mathematical notation for functions that **disregards constant factors**. That is, we characterize the running times of algorithms by using functions that map the size of the input, *n*, to values that correspond to the **main factor** that determines the growth rate in terms of *n*. This approach allows us to focus on the **big-picture** aspects of an algorithm's running time. Often, we are interested only in using a simple term to indicate how efficient an algorithm is. The exact formula of an algorithm's performance is not really needed.

Thus, in asymptotic analysis, we "disregard" constant factors (as they are system dependent) and lower order terms (as they become irrelevant for large inputs, see example below).

Number of dishes (n)	$15n^2$	45n	$15n^2 + 45n$
1	15	45	60
2	60	90	150
5	375	225	600
10	1,500	450	1,950
100	150,000	4,500	154,500
1,000	15,000,000	45,000	15,045,000
10,000	1,500,000,000	450,000	1,500,450,000
100,000	150,000,000,000	4,500,000	150,004,500,000
1,000,000	15,000,000,000,000	45,000,000	15,000,045,000,000
10,000,000	1,500,000,000,000,000	450,000,000	1,500,000,450,000,000

The asymptotic analysis of an algorithm determines the running time in big-O notation.

To perform the asymptotic analysis:

- Find the worst-case number of primitive operations executed as a function of the input size.
- Then express this function with the big-O notation.
- Since constant factors and lower order terms are eventually dropped, we can disregard them when counting the number of primitive operations.

Asymptotic analysis is an analysis of algorithms that focuses on,

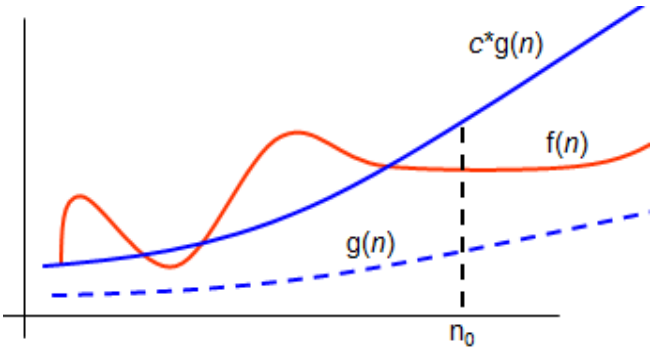
- analyzing the problems of large input size,
- considering only the leading term of the formula, and
- ignoring the coefficient of the leading term

The following will cover some notations are needed in asymptotic analysis.

The Big-O Notation

Given a function f(*n*), g(*n*) is an (asymptotic) upper bound of f(*n*), denoted as f(*n*) = O(g(*n*)), if there exist a constant $c > 0$, and a positive integer n_0 such that $f(n) \leq c*g(n)$ for all $n \geq n_0$.

- f(*n*) is said to be bounded above by g(*n*).
- O( ) is called the "big O" notation.



Example 1: The function $f(n) = 8n - 2$ is $O(n)$.
By the big-Oh definition, we need to find a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $8n - 2 \leq cn$ for every integer $n \geq n_0$.
It is easy to see that a possible choice is c = 8 and $n_0 = 1$.

Example 2: Given $f(n) = 2n^2 + 100n$, prove that $f(n) = O(n^2)$.
$2n^2 + 100n <= 2n^2 + n^2 = 3n^2$ whenever $n >= 100$.
Set the constants to be $c = 3$ and $n_0 = 100$. Hence, we have $f(n) = O(n^2)$.

Notes:
1. $n^2 < 2n^2 + 100n$ for all n, i.e., $g(n) < f(n)$, and yet $g(n)$ is an asymptotic upper bound of $f(n)$
2. c and n_0 are not unique. For example, we can choose $c = 2 + 100 = 102$, and $n_0 = 1$

Example 3: Find relevant c and n_0 values such that $f(n) = O(n^2)$.
$f(n) = 5n^2 + 2n + 1$ $\quad\quad$ $g(n) = n^2$
$5n^2 + 2n + 1 \leq 8n^2$ for $n \geq 1$
Hence, c = 8, $n_0 = 1$

The big-O notation allows us to say that **a function $f(n)$ is less than or equal to another function $g(n)$** up to a constant factor and in the asymptotic sense as n grows towards infinity. If $f(n)$ is of the form of $An + B$, where A and B are constants, it is $O(n)$.

The big-O notation gives an **upper bound** on the growth rate of a function. The statement "$f(n)$ is $O(g(n))$" means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$.

Characterizing Running Times using the Big-O Notation
The big-O notation is used widely to characterize **running times** and **space bounds** in terms of some parameter n, which varies from problem to problem, but is always defined as a chosen measure of the size of the problem. For example, if we are interested in finding a specific element in an array of integers, we should let n denote the number of elements of the array. Using the big-O notation, we can write the following mathematically precise statement on the running time of a sequential search algorithm for any computer.

Proposition: The sequential search algorithm, for searching a specific element in an array of n integers, runs in $O(n)$ time.
Justification: The number of primitive operations executed by the sequential search algorithm in each iteration is a constant. As each primitive operation runs in constant time, we can say that the running time of the algorithm on an input of size n is at most kn, where k is a constant. Hence, we may conclude that the running time of the sequential search algorithm is $O(n)$.

Some Properties of the Big-O Notation
The big-O notation allows us to ignore constant factors and lower order terms and focus on the main components of a function that affect its growth the most.

Proposition: $5n^4 + 3n^3 + 2n^2 + 4n + 1$ is $O(n^4)$
Justification: $5n^4 + 3n^3 + 2n^2 + 4n + 1 \leq (5 + 3 + 2 + 4 + 1)n^4 = cn^4$, for $c = 15$ and $n_0 = 1$.

Proposition: If $f(n)$ is a polynomial of degree d, that is, $f(n) = a_0 + a_1n + a_2n^2 + ... + a_dn^d$, and $a_d > 0$, then $f(n)$ is $O(n^d)$.
Justification: Note that, for $n \geq 1$, we have $1 \leq n \leq n^2 \leq ... \leq n^d$. Hence,
$f(n) = a_0 + a_1n + a_2n^2 + ... + a_dn^d \leq (a_0 + a_1 +... + a_d)\, n^d$.
Therefore, $f(n)$ is $O(n^d)$ by defining $c = a_0 + a_1 +... +a_d$ and $n_0 = 1$.
The highest degree term in a polynomial is the term that determines the asymptotic growth rate of that polynomial.

General rules: Characterizing Functions in Simplest Terms

In general we should use the big-O notation to characterize a function as closely as possible. For example, while it is true that $f(n) = 4n^3 + 3n^2$ is $O(n^5)$ or even $O(n^4)$, it is more accurate to say that $f(n)$ is $O(n^3)$.

It is also considered a poor taste to include constant factors and lower order terms in the big-O notation. For example, it is unfashionable to say that the function $2n^3$ is $O(4n^3 + 8n\log n)$, although it is completely correct. We should strive to describe the function in the big-O in **simplest terms**.

Rules of using big-O:

- **If $f(n)$ is a polynomial of degree d, then $f(n)$ is $O(n^d)$.**
  We can drop the lower order terms and constant factors.
- **Use the smallest/closest possible class of functions**.
  For example, "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- **Use the simplest expression of the class.**
  For example, "$3n + 5$ is $O(n)$" instead of "$3n+5$ is $O(3n)$"

Some words of caution

It is somewhat misleading when the constant factors are very large. It is true that $10^{100}n$ is $O(n)$, when it is compared with another running time $10n\log n$, we should prefer $O(n\log n)$ time, even though the linear-time algorithm is asymptotically faster.

Generally, any algorithm running in $O(n\log n)$ time (with a reasonable constant factor) should be considered efficient. Even $O(n^2)$ may be fast when n is small. But $O(2^n)$ should almost never be considered efficient.

If we must draw a line between efficient and inefficient algorithms, it is natural to make the distinction between those algorithms running in polynomial time and those running in exponential time. Again, be reasonable here. $O(n^{100})$ is not efficient at all.

1.6 Mathematical Analysis of Non-Recursive Algorithms

Generally, we will set up a sum expressing the number of times the algorithm's basic operation is executed.

Example 1: Consider the algorithm for finding the maximum value in an array.

ALGORITHM *MaxElement*$(A[0..n-1])$
//Determines the value of the largest element in a given array
//Input: An array $A[0..n-1]$ of real numbers
//Output: The value of the largest element in A
$maxval \leftarrow A[0]$
for $i \leftarrow 1$ **to** $n-1$ **do**
  if $A[i] > maxval$
    $maxval \leftarrow A[i]$
return $maxval$

The operations that are going to be executed most often are in the algorithm's for loop. Since the comparison is executed on each repetition of the loop but not the assignment, we will consider the comparison operation. Hence, we define C(n) as the number of times the comparison operation is executed.

C(n) = _____ Thus complexity is _____

Example 2: Compute C(n), where C(n) is the number of times the statement "if A[i] = A[j]" is executed.

ALGORITHM *UniqueElements*$(A[0..n-1])$
```
//Determines whether all the elements in a given array are distinct
//Input: An array A[0..n − 1]
//Output: Returns "true" if all the elements in A are distinct
//         and "false" otherwise
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] = A[j] return false
return true
```

Example 3:
Find the worst case time complexity of the following code.
```
for (int i=1; i<n; i++) {
      for (int j=1; j<n; i++) {
            sum++;
}
```

Example 4:
Find the worst case time complexity of the following code.
```
int sum = 0;
for (int i=1; i<n; i=i*2) {
      sum++;
}
```

Mathematics Review

Properties of logarithms:

$\log_b xy = \log_b x + \log_b y$

$\log_b \frac{x}{y} = \log_b x - \log_b y$

$\log_b x^r = r \log_b x$

$\log_b 1 = 0$

$\log_b \frac{1}{x} = -\log_b x$

$\log_b a = \frac{1}{\log_a b}$

$\log_b a = \frac{\log_c a}{\log_c b}$

Properties of exponentials:

$a^{b+c} = a^b a^c$

$a^{b-c} = \frac{a^b}{a^c}$

$(a^b)^c = a^{bc}$

$a^0 = 1$

$a^{-b} = \frac{1}{a^b}$

$b = a^{\log_a b}$

$b^c = a^{c \log_a b}$

Quick Notes on Summation:

$$\sum_{i=1}^{n} 1 = n$$

$$\sum_{i=1}^{n} ca_i = c \sum_{i=1}^{n} a_i$$

$$\sum_{i=1}^{n} (a_i \pm b_i) = \sum_{i=1}^{n} a_i \pm \sum_{i=1}^{n} b_i$$

Sum of Arithmetic Progression (AP)

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

More generally, for the sequence such that $a_{n+1} = a_n + c$, for some constant c, then

$$\sum_{i=1}^{n} a_i = a_1 + a_2 + a_3 + \cdots + a_n = \frac{n(a_n + a_1)}{2}$$

Sum of Geometric Progression (GP)

$$\sum_{i=0}^{n} 2^i = 1 + 2 + 4 + \cdots + 2^n = 2^{n+1} - 1$$

More generally, for the sequence such that $a_{n+1} = ra_n$, for some constant r, then

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

If $0 < r < 1$, then the sum of the infinite Geometric Series is,

$$\sum_{i=1}^{\infty} a_i = a_1 + a_2 + a_3 + \cdots = \frac{a_1}{1 - r}$$

Sum of Squares

$$\sum_{i=1}^{n} i^2 = 1 + 4 + 9 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

C1: Class Exercise

Exercise 1

State the class of growth rate for each of the following code snippets:
n is the input size

Code A

```
for (int i = n; i > 0; i /= 2) {
       // some O(1) expressions
}
```

Code B

```
for (int i = 1; i <=n; i += 2) {
       for (int j = 1; j <=n; j++) {
          // some O(1) expressions
       }
}
```

Code C

```
for (int i = n; i > 0; i -= 3) {
        // some O(1) expressions
}
```

Code D

```
for (int i = 1; i <= 1000000; i++) {
        // some O(1) expressions
}
```

Exercise 2

Prove that running time $T(n) = n^3 + 20n + 1$ is $O(n^3)$.

Exercise 3

Find the worst case time complexity of the following code. Assume n is some power of 3.

```
int sum = 0;
for (int i=1; i<=n; i=i*3) {
      for (j=1; j<=i; j++) {
            sum++;
      }
}
```

Exercise 4

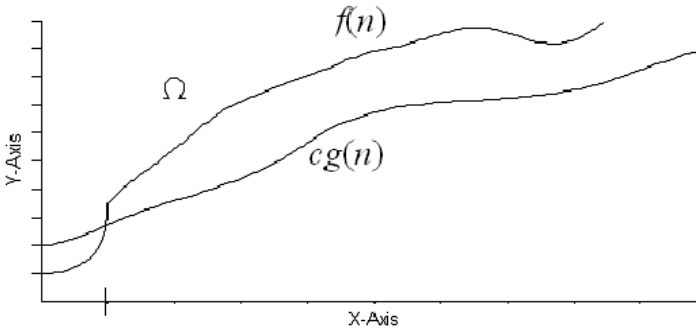
Order the following functions according to their order of growth (from lowest to highest). Show your working clearly.

$$\sqrt[3]{n},\ \ 3^n,\ \ (n-2)!\ ,\ \ 5\lg(n+100)^{10},\ \ 2^{2n},\ \ 0.001n^4+3n^3+1$$

[Optional] 1-A Other Asymptotic Notations of Measure

Big-Omega

The big-O notation provides an asymptotic representation that a function is less than or equal to another function, whereas, the big-Omega notation provides an asymptotic representation that a function grows at a rate that is **greater than or equal to** that of another.



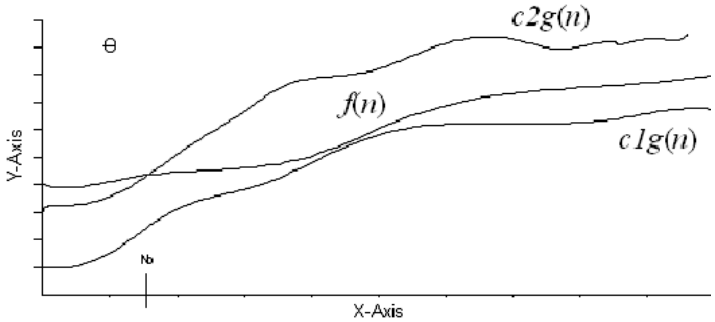
Let $f(n)$ and $g(n)$ be functions mapping nonnegative integers to real numbers. We say that $f(n)$ is **$\Omega(g(n))$** (pronounced "$f(n)$ is big-Omega of $g(n)$") if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that ***f*(*n*) ≥ *cg*(*n*)**, for $n \geq n_0$.

Example Show that n^2 is $\Omega(n\log n)$.
Justification: $f(n) \geq cg(n)$
$n^2 \geq c * n \log n$
let $c = 1$
$n^2 \geq n \log n$
$n \geq \log n$ where $n \geq 1$. Thus $c=1$, $n_0 = 1$

Big-Theta

In addition, there is a notation that represents two functions growing at the same rate, up to constant factors.

We say that ***f*(*n*) is Θ(*g*(*n*))** (pronounced "$f(n)$ is big-Theta of $g(n)$") if $f(n)$ is $O(g(n))$ and f(n) is $\Omega(g(n))$, that is, there are real constants $c_1 > 0$ and $c_2 > 0$, and an integer constant $n_0 \geq 1$ such that, $c_1 g(n) \leq f(n) \leq c_2 g(n)$, for $n \geq n_0$.



Example Show that $3n\log n + 4n + \log n$ is $\Theta(n\log n)$.
Justification: $3n\log n <= 3n\log n + 4n + 5\log n <= (3 + 4 + 5)\, n\log n$ for $n >= 2$.
Thus $c_1 = 3$, $c_2 = 12$, $n_0 = 2$

To summarize, the asymptotic notations of big-O, big-Omega, and big-Theta provide a convenient language for us to analyze data structures and algorithms. They let us concentrate on the "big-picture" rather than low-level details.