CMSC 451: Lecture 1
Introduction to Algorithm Design

What is an algorithm? This course will focus on the study of the design and analysis of algorithms for discrete (as opposed to numerical) problems. We can define *algorithm* to be:

> Any well-defined computational procedure that takes some values as *input* and produces some values as *output*.

The concept of a "well-defined computational procedure" dates back to ancient times. In fact, the word "algorithm" is derived from the Latin form of the Persian scholar Muhammad ibn Musa al-Khwarizmi, who lived in ninth century A.D. Al-Khwarizmi codified procedures for numerous arithmetic operations (such as addition, multiplication, and division with Arabic numerals) and algebraic and trigonometric operations (such as computing square roots and computing the digits of π).

Why study algorithm design? While the study of algorithms predates digital computers, the field really took off with the advent of computers. The use of asymptotic (big-Oh) notation became popular in the 1960's and 1970's as a means to provide a rigorous mathematical measure of an algorithm's running time. This evolved into the field of *computational complexity*, which seeks to categorize computational problems according to their complexity. This gave rise to the study of NP-Hard problems.

The field has also led to the development of general techniques for the design of efficient algorithms, such as divide-and-conquer, greedy algorithms, dynamic programming, and so on.

From a more practical perspective, algorithm design and analysis is often the first step the development of software for tricky combinatorial problems. Asymptotic analysis is used to identify promising solutions, which can then be prototyped in order to determine which methods perform best.

Course Overview: This course will consist of a number of major sections. The first will be a short review of some preliminary material, including asymptotics, summations and recurrences, sorting, and basic graph algorithms. These have been covered in earlier courses, and so we will breeze through them pretty quickly. Next, we will consider a number of common algorithm design techniques, including greedy algorithms, dynamic programming, and augmentation-based methods (particularly for network flow problems).

Most of the emphasis of the first portion of the course will be on problems that can be solved efficiently, in the latter portion we will discuss intractability and NP-hard problems. These are problems for which no efficient solution is known. Finally, we will discuss methods to approximate NP-hard problems, and how to prove how close these approximations are to the optimal solutions.

Issues in Algorithm Design: Algorithms are mathematical objects (in contrast to the must more concrete notion of a computer program implemented in some programming language and executing on some machine). As such, we can reason about the properties of algorithms mathematically. When designing an algorithm we need to be concerned both with its *correctness* and *efficiency.*

Intuitively, an algorithm's efficiency is a function of the amount of computational resources it requires, measured typically as execution time and the amount of space, or memory, that the algorithm uses. The amount of computational resources can be a complex function of the size and structure of the input set. In order to reduce matters to their simplest form, it is common to consider efficiency as a function of *input size*, which is usually represented by the symbol n. For example, in a sorting algorithm, this might be the number of items to be sorted. In a graph algorithm, this might be the number of vertices and/or edges in the graph. In a numerical algorithm like factoring, this might be the number of digits in a number. Since there are many inputs of the same input size, there are two common ways to aggregate these into a single quantity.

Worst-case complexity: Among all inputs of the same size, what is the *maximum* running time?

Average-case complexity: Among all inputs of the same size, what is the *expected* running time? This expectation is computed assuming that the inputs are drawn from some given *probability distribution.* The choice of distribution can have a significant impact on the final conclusions.

Asymptotic Notation: Asymptotic O-notation ("big-O") provides us with a way to simplify the messy functions that often arise in analyzing the running times of algorithms. Suppose that we analyze two algorithms, and find that they have running times of:

$$T_1(n) \;=\; 3.9n + 4.17\log n + 3.5n^2 \quad\text{and}\quad T_2(n) \;=\; \max(4.6n(\log n)^4, 6.4n^3 - 3n\log_3 n).$$

Which of these algorithms is better? Asymptotic analysis is based on (1) focusing on the growth rate by considering the performance as the value of n increases to infinity and (2) ignoring constant factors, which tend to rely on secondary issues such as programming style and machine architecture. We'll give the formal definition later, but intuitively we can say that T_1 grows on the order of n^2 and T_2 grows on the order of n^3, that is

$$T_1(n) \;=\; O(n^2) \quad\text{and}\quad T_2(n) \;=\; O(n^3).$$

Here's a formal definition:

Asymptotic noation: A function $f(n)$ is $O(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that, $f(n) \leq c \cdot g(n)$, for all $n \geq n_0$.

Intuitively, big-O notation can be thought of as a way of expressing a sort of *fuzzy* "$\leq$" relation between functions, where by fuzzy, we mean that constant factors are ignored and we are only interested in what happens as n tends to infinity.

Another (and often easier) way to think about asymptotics is in terms of limits. An alternative definition is that $f(n)$ is $O(g(n))$ if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \leq c, \quad \text{for some constant } c \geq 0.$$

For example, we can say that $T_1(n)$ is $O(n^2)$ since

$$\lim_{n\to\infty} \frac{3.9n + 4.17\log n + 3.5n^2}{n^2} \;=\; \lim_{n\to\infty}\left(3.9\frac{1}{n} + 4.17\frac{\log n}{n^2} + 3.5\right) \;=\; 3.5,$$

since in the limit $1/n$ and $\log n/n^2$ both tend to zero in the limit.

Big-O notation has a number of relatives, which are useful for expressing other sorts of relations. These include Ω ("big-omega"), Θ ("theta"), o ("little-oh"), ω ("little-omega"). Let c denote an arbitrary positive *constant* (not 0, not ∞, and not depending on n). Intuitively, each represents a form of "asymptotic relational operator":

Notation	**Relational Form**	**Limit Definition**
$f(n)$ is $o(g(n))$	$f(n) \prec g(n)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
$f(n)$ is $O(g(n))$	$f(n) \preceq g(n)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ or 0
$f(n)$ is $\Theta(g(n))$	$f(n) \approx g(n)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$
$f(n)$ is $\Omega(g(n))$	$f(n) \succeq g(n)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ or ∞
$f(n)$ is $\omega(g(n))$	$f(n) \succ g(n)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty.$

By far, the most commonly arising functions in algorithm analysis are of one of three forms. They are:

Polylogarithmic: Of the form $(\log n)^a = \log^a n$, for some constant a,

Polynomial: Of the form n^a, for some constant a, and

Exponential: Of the form a^n, for some constant a.

(There are, of course, many functions that do not fit into any of these categories, such as $O(n^{\log n})$.)

For any a, b, c, such that $a, b > 0$ and $c > 1$ we have the following relative order:

$$(\log n)^a \quad \prec \quad n^b \quad \prec \quad c^n.$$

To keep matters simple, we will focus almost exclusively on **worst-case analysis** measured using **asymptotic analysis** in this course. You should be mindful, however, that worst-case analysis is not always the best way to analyze an algorithm's performance. For example, some algorithms have the property that they run very fast on typical inputs but might run extremely slowly (perhaps hundreds to thousands of times slower) on a very small fraction of *pathological* inputs. For such algorithms, an average case analysis may be a much more accurate reflection of the algorithm's true performance. Also, sometimes one algorithm will have a better asymptotic complexity, but the constant factors are so large that there is no practical value of n where its running time is better, such as $9999n$ versus $2n\log n$.

Describing Algorithms: Throughout out this course, when you will be asked to present an algorithm. This means that you need to do three things:

Present the Algorithm: Give a clear, simple, and unambiguous description of the algorithm (in plain English prose or pseudo-code, for example). A guiding principal here is to remember that your description will be read by a human, and **not** a compiler. Obvious technical details should be kept to a minimum so that the key computational issues stand out.

Prove its Correctness: Present a justification (that is, an informal proof) of the algorithm's correctness. This justification may assume that the reader is familiar with the basic background material presented in class. Try to avoid rambling about obvious or trivial elements and focus on the key elements. A good proof provides a high-level overview of what the algorithm does, and then focuses on any tricky elements that may not be obvious.

Analyze its Efficiency: Present a worst-case analysis of the algorithms efficiency, typically it running time (but also its space, if space is an issue). Sometimes this is straightforward and other times it might involve setting up and solving a complex recurrence or a summation. When possible, try to reason based on algorithms that you have seen. For example, the recurrence $T(n) = 2T(n/2) + n$ is common in divide-and-conquer algorithms (like Mergesort) and it is well known that it solves to $O(n \log n)$.

Note that your presentation does not need to be in this order. Often it is good to begin with an explanation of how you derived the algorithm, emphasizing particular elements of the design that establish its correctness and efficiency. Then, once this groundwork has been laid down, present the algorithm itself. If this seems to be a bit abstract now, don't worry. We will see many examples of this process throughout the semester.

Background Information: We will assume that you have familiarity with the information from an introductory algorithms course. It is expected that you have knowledge of:

- Basic programming skills (programming with loops, pointers, structures, recursion)
- Discrete mathematics (proof by induction, sets, permutations, combinations, and probability)
- Understanding of basic data structures (lists, stacks, queues, trees, graphs, and heaps)
- Knowledge of sorting algorithms (MergeSort, QuickSort, HeapSort, CountingSort, and RadixSort) and basic graph algorithms (minimum spanning trees and depth-first search)
- Basic calculus (manipulation of exponentials, logarithms, differentiation, and integration)

Topics to be Covered: Here is a tentative list of topics to be covered in this course.

Introduction: Review of algorithm design and analysis, review of basic graph theory and graph representations¡

Graph Exploration: DFS and BFS, topological sorting, strong components, shortest paths

Greedy Algorithms: Interval scheduling and partitioning, scheduling to minimize lateness, greedy graph algorithms

Dynamic Programming: Weighted interval scheduling, longest common subsequences, chain matrix multiplication, all-pairs shortest paths in graphs

Network Flow: Network flow algorithms, bipartite matching, circulations and applications

NP-Hardness and Intractability: Polynomial-time reductions, the definition of NP, NP-complete problems

Approximation Algorithms: Greedy algorithms and polynomial-time approximation schemes (and examples)