What is CS 340 All About?

Data Structures + Algorithms = Programs

Good Data Structures + Good Algorithms = Good Programs

THINK SERIOUSLY ABOUT ALGORIHMS

THINK MORE SERIOUSLY ABOUT DATA STRUCTURES

We will learn abstract data types (ADT) such as QUEUE, STACK, PRIORITY QUEUE, and data structures to implement these data types.

"Algorithm" derived from Mohammed Al-Khowarizmi, 9th century Persian, mathematician credited with formalizing pencil-and-paper methods for addition, subtraction, multiplication, and division.

Examples of "algorithms" in the nature:

- Your DNA
- Your cook book

Big-O notation

On input of say n numbers, an algorithm could use n steps, or $n+1$, or $n+35$, or $10n$ steps. This will be all the same to us: we will use $O(n)$ to mean: "hmm, this is approximately n within a multiplicative factor." In General,

Definition.

- We say $f(n)$ is $O(g(n))$ if there are constants c and n_0 such that for all $n \geq n_0$, $f(n) \leq cg(n)$.

- We say $f(n)$ is $o(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

Example. $3n^2, n^2 + 100n, 10n$ are all $O(n^2)$. $100\sqrt{n}$ is $o(n)$. $n \log n$ is $o(n^2)$. n^{100} is $o(2^n)$.

A good algorithm can be much better than a bad one:

Example: Evaluate

$$\sum_{i=0}^{n} a_i x^i.$$

Solution 1. Do it term by term:

- n additions;
- i multiplications for $a_i x^i$.

$\sum_{i=0}^{n} i = \frac{(n+1)n}{2}$ multiplications and n additions. Total: $O(n^2)$ operations.

Solution 2. (Horner's Rule)

```
P:= a(n)
for i := n-1 step -1 downto 0 do
    P:= P*x + a(i)
```

- n additions;
- n multiplications.

Total: $O(n)$ operations.

Efficiency of Computation

Run time of a program is affected by many factors, for example

- input data, or problem size
- algorithm
- Compiler and operating system
- hardware
- too many users on the computer

To analyze the efficiency of an algorithm, we will only count the number of operations (as a function of the input size). This is **machine-independent**.

But

- If a program is to be run only a few times, then the cost of writing/debugging dominates.

- With small input, "constant factor" dominates.

- We also need to watch space requirement: space complexity.

- One trick: Divide & Conquer. Try to find a "good" way to divide the problem (input) into two halves, and solve them (recursively) separately.

An Example for Divide and Conquer. Majority Problem: Given n numbers, find the one that appears more than $\frac{n}{2}$ times, using only "="

Example: In 1,2,1,2,1,1,3,1,1,5. '1' is the majority.

You already know how to solve this as follows: You take each number, compare to rest, and count. This costs you $O(n^2)$ comparisons in the worst ase.

But we want to do better.

My algorithm:

Majority($A[1..n]$: array of n integers);
m_1 := Majority($A[1, \frac{n}{2}]$);
m_2 := Majority($A[\frac{n}{2}+1, n]$):
Compare m_1, m_2 with A, return the true majority m_1 or m_2.

My time:

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + O(n) \\ &= O(n \log n). \end{aligned}$$

Other algorithm design techniques (other than divide & conquer).

- reduce to known problem (e.g. sorting)
- recursion
- invent (or augment) a data structure
- dynamic programming
- greedy algorithm
- probabilistic solutions
- approximate solutions

We will discuss some of these topics as we meet concrete problems during this course.

- Read Chapter 1.