CSCI3104: Algorithms

Introduction, Algorithms, and Complexity

- course logistics
- what is an algorithm
- course goals
- algorithm analysis

Course logistics

basic information

- course materials: moodle.cs.colorado.edu (enrollment key: csci3104_200)
- time and location: MWF 1:00pm-1:50pm in ECCR 200
- office hours:
  - instructor (ECOT335): MW 2:00-3:00pm, or by appointment (email is most convenient)
  - Varad Deshmukh (ECCS112A): TTH 11:00 am-12:00pm
  - Sebastian Lautenschlger (ECCS122): T 2:00-4:00pm
  - Jacob Hallberg (ECCS112): MWTH 3:30-6:30pm
- textbook: Introduction to Algorithms (3rd edition), by Cormen et al.
- prerequisite: CSCI 2270 Data Structures, CSCI 2824 Discrete Structures

course composition

- lectures 3 times a week (MWF)

- weekly recitation sections with course TAs
  - recitation activities must be submitted by the end of recitation
  - your lowest recitation grade will be dropped

- 10 problem sets (written and programming questions) assigned weekly, due 10 days from date assigned
  - 3 days extension with 20% grade penalty; after 3 days, cannot be turned in
  - in case of personal, family, medical emergency, consult TAs for penalty-free extension
  - collaboration is strongly encouraged, but write your own solutions
  - first assignment will be out before class on Friday September 1, due before class on Monday September 11

- online lecture quizzes a few times a week over lecture material for that week
- one midterm exam in 8th week
- one final project (more like a take-home exam)

grading

- recitation 10%
- homework 30%
- lecture quizzes 10%
- midterm exam 25%
- final project 25%

final course grade

- the grade follows the standard percentage breakdown for the College of Engineering:
  - 93%-100% A
  - 90%-93% A-
  - 87%-90% B+
  - 83%-87% B
  - 80%-83% B-
  - 77%-80% C+
  - 73%-77% C
  - 70%-73% C-
  - 67%-70% D+
  - 63%-67% D
  - 60%-63% D-
  - 0%-60% F

policies

- accommodation for disability
- religious observances
- discrimination and harassment
- classroom behaviors
- student honor code
- specific collaboration policy for CSCI3104

see the syllabus and pointers there for more details

Algorithms

- a sequence of primitive steps to perform a task
- what constitutes a primitive steps?
  - instruction set of a modern process?
  - things carried out by a person mentally?
  - depends on context, more later...

An example

a question commonly asked in job interviews:

- you are given an array containing integers between 1 and 1,000,000.
- every integer from 1 and 1,000,000 is in the array once, but one is in the array twice
- questions:
  - can you determine which integer is in the array twice?
  - can you do it while iterating through the array only once?

A simple solution

- a simple algorithm
  - create a new array L of ints between 1 and 1,000,000; use this array to count the occurrences of each number
  - initialize all entries of L to 0
  - iterate over the input array; each time a number i is seen, increment the count $L[i]$ in the new array
  - iterate over L and see which number occurs twice, $L[i] > 1$
  - Return that number, i

- how long does the algorithm take?
  - iterate through 1,000,000 numbers 3 times
  - it the size of array is n, it takes $\Theta(n)$ time

A simple solution

- a simple algorithm
  - create a new array L of ints between 1 and 1,000,000; use this array to count the occurrences of each number
  - initialize all entries of L to 0
  - iterate over the input array; each time a number i is seen, increment the count $L[i]$ in the new array
  - iterate over L and see which number occurs twice, $L[i] > 1$
  - Return that number, i

- how much space does the algorithm take?
  - twice as much space as the input sequence
  - it the size of array is n, it takes $\Theta(n)$ space

A simple solution

- a simple algorithm
  - create a new array L of ints between 1 and 1,000,000; use this array to count the occurrences of each number
  - initialize all entries of L to 0
  - iterate over the input array; each time a number i is seen, increment the count $L[i]$ in the new array
  - iterate over L and see which number occurs twice, $L[i] > 1$
  - Return that number, i

- can we do better?

A better solution

recall that $\sum_{i=1}^{n} i = n(n+1)/2$, so we have

- let S be the sum of the values in the input array
- Let n be the largest integer in the array; in this case, $n = 1;000;000$
- let x be the value of the repeated number
- then $S = n(n+1) = 2 + x$
- and $x = S - n(n+1)/2$

A better solution

thus, an efficient algorithm:

- iterate through the input array, summing the numbers; let S be this sum; let n be the largest value observed in this iteration
- let $x = S - n(n+1)/2$
- return x

A better solution

thus, an efficient algorithm:

- iterate through the input array, summing the numbers; let S be this sum; let n be the largest value observed in this iteration
- let $x = S - n(n+1)/2$
- return x

how much time and space?

- roughly 3 times faster; still $\Theta(n)$ time
- much less space; only 3 constants stored, constant space - $\Theta(1)$

Key message

- designing good algorithm matters:
  - correct algorithm (does not fail on any input)
  - efficiency algorithm (use as few resources as possible)
    - time
    - space (storage)
    - communication
    - ...
  - performance guarantee where possible (think carefully about worst possible behavior)

- achieving the above goals not always easy;
  - many are clever
  - very few are most efficient

Course goals

learn how to think rigorously about algorithms:

- a sequence of primitive steps to perform a task
- a computational process transforming some input X into an output Y with specified properties, under a resource budget of Z
- will be done in part by learning specific algorithms and how to analyze them mathematically
  - mathematically prove an algorithm A is correct
  - mathematically characterize the resource (time and/or space) $f(n)$ used for an input of size n

course goals

specifically:

- become familiar with "standard" algorithms for abstract problem solving
- learn how to mathematically prove properties of algorithms, including their correctness
- analyze the time and space complexity of algorithms
- understand the relative merits or demerits of different algorithms in practice
- adapt and combine algorithms to solve problems that may arise in practice
- learn common strategies in the design of new algorithms for emerging applications

Analyzing algorithms

random access machine (RAM) model of computation

- a single processor with infinite amount of memory
- instructions executed one after another, with no concurrent operations
- pretty close to how most off-the-shelf computers work today

Atomic v.s. non-atomic

- atomic operations that take 1 unit of time ($O(1)$ time)
  - simple mathematical: $+$, *, -, $=$
  - simple functional: if, call, store, etc
- higher-order operations composed of many atomic operations
  - loops: for, while, etc
  - subroutines: e.g., sort a list
- atomic data types that take 1 unit of space ($O(1)$ space) to store and 1 unit of time to access
  - simple data types: a single boolean, integer, character, real number
- higher-order data types composed of many atomic data types
  - arrays, lists, trees

- all models make a distinction between atomic and non-atomic, but may differ on what they consider atomic
  - division, modular arithmetic, generating random variable

Two goals of Algorithm design/analysis

algorithms that are correct

- a function f that takes input X and maps it to an output Y with certain properties: $f : X \to Y$
- a function f is called correct if and only if for every possible input X, it outputs a Y with the correct properties
- example: f is a function of sorting a list of n number
  - X is one of $n!$ permutations of n numbers
  - Y will be the permutation of X such that $y_1 \leq y_2 \leq y_3 \leq 1 \cdots \leq y_n$ (if f sorts in ascending order)
- will learn how to design correct algorithm, show an algorithm is correct, and identify when an algorithm is not correct

Two goals of Algorithm design/analysis

algorithms that are efficient

- use few resources
  - two kinds of resources: time and space
- determined by adding up all the atomic operations and expressing this quantity as a function of the size of the input $n = |X|$
- will learn how to design efficient algorithm, quantify how efficient an algorithm is, and identify when an algorithm is not efficient

Worst-case v.s. average case algorithm analysis

- worst case analysis: analyze the algorithm's behavior under the worst possible input for correctness or for resource usage (time or space)
  - pessimistic or adversarial approach to algorithm analysis
  - be able to identify worst cases and use them to show mathematically the correctness and efficiency of the algorithm
- average case analysis: average over all possible inputs
  - implies certain distribution on inputs
  - sometimes significantly better than the worst case

Asymptotic analysis

- interested in the functional form of the resource usage as the size of the input n goes to infinity

- simplifies the analysis and concisely communicates the core differences between different algorithms

- $\Theta(\cdot),\ O(\cdot),\ \Omega(\cdot)$
  - ignore all (additive or multiplicative) constants
    - 10^{10} and 4 are all $O(1)$
    - n, $5n + 2$ and $100000n$ are all $O(n)$
  - keep leading terms
    - $n^2 + n + \log n$ and $100n^2 + 2n - \sqrt{n}$ are all $O(n^2)$

Announcements

- office hours:
  - instructor (ECOT335): MW 2:00-3:00pm, or by appointment (email is most convenient)
  - Varad Deshmukh (ECCS112A): TTH 11:00 am-12:00pm
  - Sebastian Lautenschlger (ECCS122): T 2:00-4:00pm
  - Jacob Hallberg (ECCS112): MWTH 3:30-6:30pm
- first assignment out today and due by 1pm Monday September 11
  - submit through modal
  - see problem set solutions template
  - see any additional instruction
- reading
  - this week: chapters 1-3
  - next week: chapters 3-4

Proofs by induction

- induction is a simple and powerful technique for proving mathematical claims
- one of the most common tools in algorithm analysis
- example: show the sum of n positive number

$$S(n) = 1 + 2 + \cdots + n = \frac{n(n+1)}{2};$$

proof by induction (on n) has two parts:

- base case: verify that $S(n)$ is correct for the smallest possible value, i.e., $n = 1$.

$$S(1) = \frac{1(1+1)}{2} = 1$$

- induction step:

* (1) assume that S(n) holds for an arbitrary input of size n and then
* (2) prove that it also holds for $n+1$:

$$\begin{aligned} S(n+1) &= S(n) + (n+1) && \text{(by definition)} \\ &= \frac{n(n+1)}{2} + (n+1) && \text{(by inductive hypothesis)} \\ &= \frac{(n+1)((n+1)+1)}{2} && \text{(algebra)} \end{aligned}$$

1 The asymptotic analysis of algorithms[1]

Recall that in algorithm analysis and design, we are mainly interested in (i) algorithms that are *correct*, and (ii) algorithms that are *efficient*. The first of these is a binary statement. An algorithm is either correct, or it is not. Efficiency, however, is a relative term: algorithm $\mathcal{A}$ can be more or less efficient than algorithm $\mathcal{B}$.

In order to formally compare the resource usage (time or space) of different algorithms, we use *asymptotic analysis* and *asymptotic notation*. The "asymptotic" here means that we are only interested in the functional form of the resource usage as the size of the input n goes to infinity. This allows us to state formally and precisely how much of a resource (time or space) an algorithm uses, and to make statements like $\mathcal{A}$ is more, less, or equally efficient than $\mathcal{B}$.

For instance, if $T(n)$ is the running time of an algorithm when the input has size n, then we want to know the $g(n)$ such that

$$\lim_{n\to\infty} T(n) = \Gamma(g(n)) \ ,$$

where $g(n)$ is typically one of a small number of simple functional forms, and Γ indicates the kind of asymptotic relationship, typically O, Θ, or Ω (more about these below). Common examples of $g(n)$ include:

$g(n)$		meaning
1	:	constant resource usage, independent of n
$\log n$	:	sublinear, specifically logarithmic resource usage
n^c for $c < 1$	:	sublinear resource usage
n	:	linear resource usage
$n \log n$	:	super-linear, but much less than quadratic
n^c for $c > 1$	:	polynomial resource usage, super-linear
c^n for $c > 1$	:	exponential resource usage
n^n	:	extremely fast-growing resource usage

1.1 An example, and the tight bound of Θ

For instance, how much space does an array consume? In a low-level language like `C`, an array is literally a contiguous block of memory and thus takes an exact amount of space. If an element in the array takes up 32 bits of space, then an array of n elements takes $32n$ bits or $4n$ bytes. If an element takes 64 bits or 128 bits, then an n element array takes $64n$ or $128n$ bits of space.

[1] Acknowledgement: Adopted from Prof. Aaron Clauset's lecture notes.

Notice, however, that each of these has a common property: each takes space that grows exactly proportional to a *linear* function of the size of the array n. This property is true no matter how many bits each element takes, so long as an element's "size" does not itself vary with n. This is precisely what we mean by saying that, in this case, the amount of space required by an array is $\Theta(n)$.

Formally, $\Theta(n)$ means something very specific. When we say that the amount of resources consumed (time or space) is some function $f(n)$ and that $f(n) = \Theta(g(n))$, we mean that

$$c_1 g(n) \leq f(n) \leq c_2 g(n) \qquad \exists c_1 > 0, \exists c_2 > 0, \forall n \geq n_0 > 0 \ . \tag{1}$$

In words, saying $f(n) = \Theta(g(n))$ is a mathematical claim that there exists some choice of input size n_0 such that for all larger inputs $n > n_0$, there are choices of the constants c_1, c_2 so that the function $g(n)$ is both an upper *and* a lower bound on $f(n)$. (See Figure 1a.)

A tight bound. When the upper- and lower-bound functions have the same functional form, we call the bound *tight*, which is what Θ means in algorithms. If you are asked to justify a claim that some function is $\Theta(g(n))$, one way to do this is to identify the constants c_1, c_2, n_0 that makes the statement true.

This way of comparing functions is useful in algorithms because it lets us focus on the dominant or leading-term behavior. It allows us to quickly compare different algorithms and make judgements about which will, in the long run, be more efficient, i.e., use fewer resources. For instance, $\Theta(n)$ is more efficient than $\Theta(n^2)$, which is more efficient than $\Theta(c^n)$ for $c > 1$, etc.

1.2 Upper or lower bounds: O and Ω

Tight bounds are the goal of all algorithm analysis: they tell us exactly how fast a function grows. But, tight bounds are often hard to obtain and instead we must settle for a weaker statement.

The most common of these is an **upper bound** or $O(g(n))$ or "Big-O", which relaxes the definition of Θ by omitting the claim that $g(n)$ is a lower bound on the resource usage.

Formally, $f(n) = O(g(n))$ means

$$f(n) \leq c_2 g(n) \qquad \exists c_2 > 0, \forall n \geq n_0 > 0 \ . \tag{2}$$

In words, O (or sometimes $\mathcal{O}$) says that our function $f(n)$ grows no faster than $g(n)$, and thus $g(n)$ provides an upper bound on the resource use. This is by far the most common type of statement we will make about an algorithm.

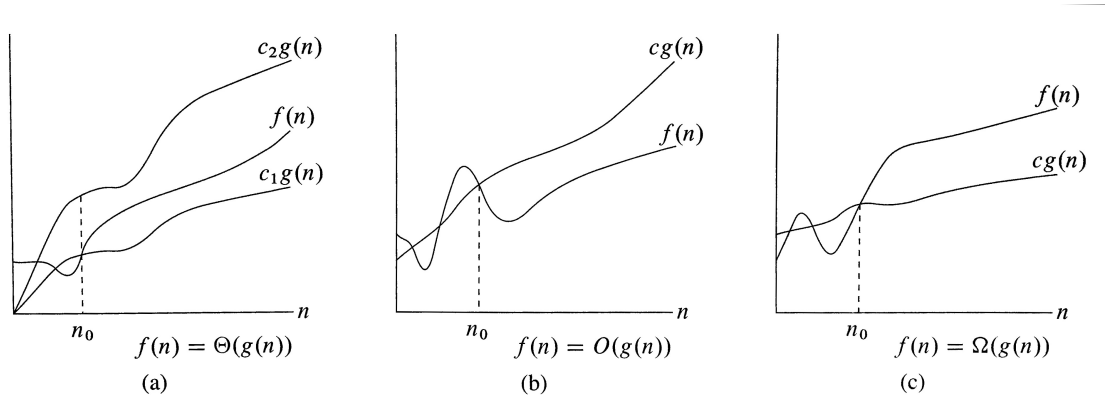

Figure 1: Schematics of the asymptotic growth of functions, showing (a) a tight bound $f = \Theta(g(n))$, (b) an upper bound $f = O(g(n))$, and (c) a lower bound $f = \Omega(g(n))$, for input sizes $n > n_0$. (Reproduced from our textbook *Introduction of Algorithms*, by Cormen et al.)

On the other hand, if we relax the definition of Θ to omit the upper bound, then we obtain a **lower bound** or $\Omega(g(n))$ or "Big-Ω", defined as

$$c_1 g(n) \leq f(n) \qquad \exists c_1 > 0, \forall n \geq n_0 > 0 \ . \tag{3}$$

Lower bounds on resource usage are substantially less useful than upper bounds, because lower bounds say nothing about the worst case scenario.

1.3 Strict bounds: o and ω

You may have noticed that in the definitions of Θ, O, and Ω, we use the $\leq$ symbol, meaning that $f(n)$ may grow exactly like $g(n)$.

This is an important detail: if we remove the equality from the definitions of O and Ω in Eqs. (2) and (3), we obtain *strict* bounds, which we denote by o and ω respectively. For instance, $n = \omega(1)$ and $n^2 = o(n)$. These bounds will be less common in this class, but often appear in more advanced algorithms topics.

1.4 A warning, and two examples of sloppy thinking

Asymptotic analysis is a powerful way to make precise statements about the way different functions grow with n, but with great power comes great responsibility.[2] A claim that some function is $\Theta(n)$ may be obvious, but you should be prepared to back up that claim by providing (or knowing that

[2]Yes, you read that correctly.

you can provide with a moment of work) the corresponding constants.

It is easy to think of asymptotic notation, i.e., O, Θ, and Ω as heuristics. This can be dangerous. For instance, what is wrong with the following statement?

$$f(n) \text{ is at least } O(g(n))$$

The phrase "at least" is a lower-bound-type statement, while O is an upper-bound-type statement, so the statement overall is a contradiction: no function can grow at least as fast as a function that it grows no faster than.

Here is another problematic statement:

$$2n = O(2^n)$$

Technically, this statement is correct because all exponential functions grow faster than all linear functions. (Can you prove this?[3]) Here, we can derive the specific values of c_2 and n_0 that makes this statement true:

$$c_2 = 1 \qquad n_0 > 1 \ .$$

To verify this, we substitute these values into the original form: $2 \cdot 1 \leq 1 \cdot 2^1$, which is true. But, this statement is not very helpful because the exponential function grows much, much more quickly than the linear function. In fact

$$n = O(n^2) = O(n^3) = O(1.1^n) = O(1.11^n) = O(2^n) \ , \tag{4}$$

which illustrates that there is a lot of room (in fact, infinitely much) between $O(n)$ and $O(2^n)$. So, while technically correct, this bound is so loose as to be nearly trivial. In algorithms, it is a virtue to strive for the tightest bound possible. Ideally, we provide a tight bound Θ. Lacking that, we provide a tight upper O or tight lower bound Ω, depending on what we are trying to show about the algorithm.

1.5 Asymptotic analysis in practice

We will use asymptotic analysis and asymptotic notation throughout the semester. The final statements we are aiming for are compact Big-O statements. There are two basic strategies for formally arriving at a correct asymptotic statement.

1. Identify the constants n_0 and c_1 or c_2 for Ω or O (or both for Θ), and demonstrate that the formal definitions given in Eqs. (1), (2), and (3) are satisfied by these constants. Or,

[3]Hint: compare $f(n) = n^a$ and $g(n) = b^n$, for $a \geq 0$ and $b > 1$.

2. Use asymptotic analysis, typically by applying L'Hopital's rule in the limit of $n \to \infty$, in order to obtain the correct functional relation.

Recall L'Hopital's rule, which states that the asymptotic relationship of $f(n)$ to $g(n)$ is the same as the asymptotic relationship of the derivatives of $f(n)$ and $g(n)$:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\frac{\partial}{\partial n} f(n)}{\frac{\partial}{\partial n} g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)} \ .$$

When one applies L'Hopital's rule, the limit of the resulting ratio will itself be a function, and the form of this function tells us unambiguously whether $f(n)$ grows faster, slower, or as quickly as $g(n)$. Specifically,

$$\begin{aligned}
&\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \qquad \text{where } c \text{ is some constant} && \implies f(n) = \Theta(g(n)) \\
&\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty && \implies f(n) = \Omega(g(n)) \\
&\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 && \implies f(n) = O(g(n)) \ .
\end{aligned}$$

For instance, consider $f(n) = n^2 + n - 1$ and $g(n) = n^2$:

$$\begin{aligned}
\lim_{n\to\infty} \frac{n^2+n-1}{n^2} &= \lim_{n\to\infty} \frac{\frac{\partial}{\partial n}(n^2+n-1)}{\frac{\partial}{\partial n} n^2} \\
&= \lim_{n\to\infty} \frac{2n+1}{2n} \\
&= 1 \\
&\implies f(n) = \Theta(g(n)) \ .
\end{aligned}$$

Or, $f(n) = n\log n + n$ and $g(n) = n^2$:

$$\begin{aligned}
\lim_{n\to\infty} \frac{n\log n + n}{n^2} &= \lim_{n\to\infty} \frac{\frac{\partial}{\partial n}(n\log n + n)}{\frac{\partial}{\partial n} n^2} \\
&= \lim_{n\to\infty} \frac{2+\log n}{2n} \\
&= 0 \\
&\implies f(n) = O(g(n)) \ .
\end{aligned}$$

And, finally $f(n) = 2^n$ and $g(n) = 1.1^n$:

$$\begin{aligned}\lim_{n\to\infty} \frac{2^n}{1.1^n} &= \lim_{n\to\infty} \frac{\mathrm{e}^{(\ln 2)n}}{\mathrm{e}^{(\ln 1.1)n}} \\ &= \lim_{n\to\infty} \mathrm{e}^{n(\ln 2 - \ln 1.1)} \\ &= \lim_{n\to\infty} \mathrm{e}^{n \ln(2/1.1)} \\ &= \infty \\ &\implies f(n) = \Omega(g(n)) \ ,\end{aligned}$$

which did not require applying L'Hopital's rule, but did require some simple mathematical identities from the following useful list:

1. $(x^y)^z = x^{yz}$
2. $x^y x^z = x^{y+z}$
3. $\log_x y = z \implies x^z = y$
4. $x^{\log_x y} = y$ by definition
5. $\log(xy) = \log x + \log y$
6. $\log(x^c) = c \log x$
7. $\log_c(x) = \log x / \log c$

2 On your own

1. Read Chapters 3 and 4 in CLRS.