CS 344: Design and Analysis of Computer Algorithms

(Spring 2022 — Sections 5,6,7,8)

Week 1: Lectures 1 and 2
Introduction to Algorithms and Asymptotic Analysis

This week's topics:

- Intro to algorithms and algorithmic thinking
- Algorithm design process
- Runtime analysis and asymptotic notation
- Starting the fun part: algorithm design in action

This week's topics:

- Intro to algorithms and algorithmic thinking
- Algorithm design process
- Runtime analysis and asymptotic notation
- Starting the fun part: algorithm design in action

- **Disclaimer:** we are using some of the video lectures from previous iterations of this course

Announcements

- My office hours/Q&A sessions:
  - Thursdays 4pm to 5pm
  - Mondays 5pm to 6pm.
- Quiz 1 from the materials of this week
  - Due Monday, Jan 24, 11:59pm.
- “Homework 0” (basic introduction to LaTeX — only bonus grade)
  - Due next Tuesday, Jan 25, 11:59pm.

Intro to Algorithms and Algorithmic Thinking

Algorithms and Algorithmic Thinking

- An algorithm is simply a sequence of simple instructions:
  - to build a puzzle
  - to put pieces of a bookshelf together
  - a food recipe
  - a computer program

Algorithms and Algorithmic Thinking

- An algorithm is simply a sequence of simple instructions:
  - to build a puzzle
  - to put pieces of a bookshelf together
  - a food recipe
  - a computer program

Simple is relative

Algorithms and Algorithmic Thinking

- An algorithm is simply a sequence of simple instructions:
  - to build a puzzle
  - to put pieces of a bookshelf together
  - a food recipe
  - a computer program

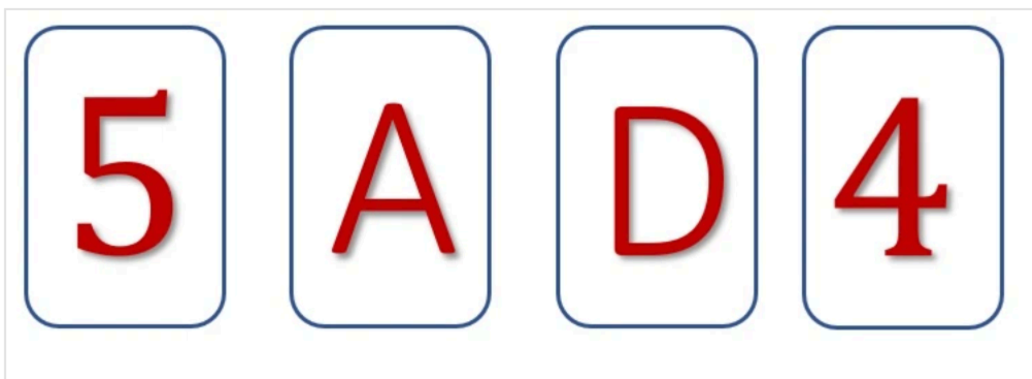
Simple is relative

Algorithms and Algorithmic Thinking

- Algorithmic thinking:
  - how to get to and analyze a solution using clearly specified steps
- Main prerequisite:
  - Logical reasoning
- Other prerequisites:
  - mathematical maturity: proof, basic algebra, and calculus

A Puzzle on Logical Reasoning

- We have a deck of cards each containing a digit on one side and an English letter on the other side
- Suppose we deal the following cards:



- I claim that behind every vowel is an odd number
- If you were to prove me wrong, which card(s) would you flip?

https://adamsheffer.wordpress.com/2019/08/18/abstraction-is-hard

Algorithm Design

A Multi-Step Process

- Algorithm design is a process for answering the following questions:
  - What is the problem to solve?
  - How to solve the problem?
  - Why the solution is correct?
  - How efficient is the solution?

A Multi-Step Process

- What is the problem to solve?
  - Problem: A mapping from any input to valid output(s)
  - Example:
  - Finding maximum: given an array of n numbers $A[1:n]$, find the largest number in the array
  - Reversing a string: given a string of length n, output the string in reverse order
  - Given two points on a map, find the shortest route between them

A Multi-Step Process

- What is the problem to solve?
  - Problem: A mapping from any input to valid output(s)
  - Example:
  - **Finding maximum: given an array of n numbers $A[1:n]$, find the largest number in the array**
  - Reversing a string: given a string of length n, output the string in reverse order
  - Given two points on a map, find the shortest route between them

A Multi-Step Process

- How to solve the problem?
  - Algorithm: a sequence of simple and well-defined operations for solving a given problem
  - Example of an algorithm:

A Multi-Step Process

- An algorithm for finding maximum in array A[1:n]:

  1. Let candidate MAX be the element A[1]

  2. Iterate over elements A[i] for i = 1 to n: if A[i] > MAX, then let MAX=A[i].

  3. Output MAX as the maximum element of the array

A Multi-Step Process

- Example of NOT an algorithm:

    1. Find the largest element of A[1:n]

    2. Output this number

A Multi-Step Process

- Example of NOT an algorithm:
  1. Find the largest element of A[1:n]
  2. Output this number

Not a simple enough step <u>for the purpose of this problem</u>

A Multi-Step Process

- Why your solution/algorithm is correct for the problem?
  - Proof: a series of logical arguments that guarantee the correctness of a mathematical statement
  - Example of a proof:
  - A proof that the first algorithm for finding maximum is correct:

Proof

- **Claim**: At the end of each iteration i, MAX is equal to the maximum entry of A[1:i].
- **Proof**:
  - This is true for i =1 because MAX=A[1] in step 1 of the algorithm
  - Suppose this is true for step i=j, and we prove it for step i=j+1
  - Since we assume the claim is true for i=j, at the end of iteration j, MAX is equal to the maximum value of A[1:j]
  - At iteration j+1, value of MAX would be either the same, or A[j+1], depending which one is larger
  - The larger number is equal to the maximum of A[1:j+1]
  - The claim is true for i=1, and if it is true for i=j, it is also true for i=j+1. Thus, the claim is true for all integers i=1 to n.

Proof

- **Claim**: At the end of each iteration i, MAX is equal to the maximum entry of A[1:i].
- **Proof**:
  - This is true for i =1 because MAX=A[1] in step 1 of the algorithm
  - Suppose this is true f... step i=j+1
  - Since we assume th... of iteration j, MAX is equal to the maximum value ...
  - At iteration j+1, value of M... would be either the same, or A[j+1], depending which one is larger
  - The larger number is equal to the maximum of A[1:j+1]
  - The claim is true for i=1, and if it is true for i=j, it is also true for i=j+1. Thus, the claim is true for all integers i=1 to n.

This type of argument is called induction

Proof

- **Claim**: At the end of each iteration i, MAX is equal to the maximum entry of the array $A[1:i]$.

- Having proved this claim, we can look at the output of the algorithm

- It is equal to MAX at the end of iteration n

- By above claim, MAX is the largest number of $A[1:n]$, proving the correctness

A Multi-Step Process

- Example of NOT a proof:
  - MAX is equal to the largest entry of the array, so the algorithm is correct

A Multi-Step Process

- How efficient is your solution/algorithm?
  - Measure of efficiency in this course: the runtime of the algorithm as a function of the input size
  - Called runtime analysis
  - Example:
  - Runtime analysis for the algorithm that finds maximum

Runtime Analysis

- Algorithm:

  1. Let candidate MAX be the element A[1]

  2. Iterate over elements A[i] for i = 1 to n: if A[i] > MAX, then let MAX=A[i].

  3. Output MAX as the maximum element of the array

Runtime Analysis

- Computing the exact runtime depends on many things:
  - Programming language used to implement the algorithm
  - Hardware for running the algorithm
  - …

Runtime Analysis

- Better measure:
  - Counting the number of “elementary operations”
  - Find out how this number behave as a function of input size

Runtime Analysis

some integer C1

- Algorithm:

    1. Let candidate MAX

    some integer C2 * n

    2. Iterate over eleme … , then let MAX=A[i].

    3. Output MAX as the maximum element of the array

some integer C3

Runtime Analysis

some integer C1

- Algorithm:

  1. Let candidate MAX

     some integer C2 * n

  2. Iterate over eleme , then let MAX=A[i].

  3. Output MAX as the maximum element of the array

     some integer C3

Runtime Analysis

- Under this measure, the runtime of this algorithm is
  - C1 + C2*n + C3
- Depending on the situation, C1,C2, and C3 will change but the formula will be the same

Runtime Analysis

- This is still not convenient enough to work with
- Consider the following two algorithms for finding maximum

Runtime Analysis

1. Let candidate MAX be the element A[1]
2. For i = 1 to n: if A[i] > MAX, then let MAX=A[i].
3. Output MAX as the maximum element of the array

1. For i = 1 to n:
   1. For j=1 to n except for i:
      1. If A[i] < A[j], break the inner-loop
   2. If the inner-loop was never broken, output A[i] as the answer

Runtime Analysis

1. Let candidate MAX be the element A[1]
2. For i = 1 to n: if A[i] > MAX, then let MAX=A[i].
3. Output MAX as the maximum element of the array

C1 + C2*n + C3

1. For i = 1 to n:
    1. For j=1 to n except for i:
        1. If A[i] < A[j], break the inner-loop
    2. If the inner-loop was never broken, output A[i] as the answer

D1 * n * D2 * n

Runtime Analysis

1. Let candidate MAX be the element A[1]
2. For i = 1 to n: if A[i] > MAX, then let MAX=A[i]
3. Output MAX as t... maximum element of the array

1. For i = 1 to n:
   1. For j=1 to n except for i:
      1. If A[i] < A[j], break the ... loop
   2. If the inner-loop was never broken, output A[i] as the answer

Asymptotic notation

$O(n)$

$O(n^2)$

Asymptotic Notation

- A way of stating the runtime of algorithms (among others) to focus on the main parts
- Comparing different algorithms on the same input size
- Example: An algorithm with runtime $O(n)$ will be faster than $O(n^2)$ on large enough inputs
- Comparing the same algorithm on different input sizes
- Example: Increasing the input size by a factor of 5:
  - An algorithm with runtime $O(n)$ becomes 5 time slower
  - An algorithm with runtime $O(n^2)$ becomes 25 time slower

Runtime Analysis

- The goal of runtime analysis is to understand the runtime:
  - As a function input on large enough inputs
  - In the worst case, i.e., the worst time the algorithm can take on any input

Review of Asymptotic Notation

O-notation

- Say we have two algorithms A and B with runtime f(n) and g(n)
- O-notation: “the $\leq$ operator in the limit”
- Writing $f(n) = O(g(n))$ means A runs “faster (or equal to)” B on sufficiently large inputs
- The mathematical formulation is that:
- $\lim_{n \to +\infty} \frac{f(n)}{g(n)} \leq C$ for some absolute constant $C \geq 0$

O-notation Example:

1. If $f(n) = 100n + 500$ and $g(n) = \frac{1}{200} \cdot n$ then $f(n) = O(g(n))$

why?

$$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = \lim_{n \to +\infty} \frac{100n + 500}{\frac{1}{200}n} = \lim_{n \to +\infty} \frac{20000n}{n} + \lim_{n \to +\infty} \frac{100000}{n} = 20000 + 0 = 20000$$

O-notation Example:

2. If $f(n) = 10n + 500$ and $g(n) = \dfrac{1}{2}n^2$ then $f(n) = O(g(n))$

why?

$$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = \lim_{n \to +\infty} \frac{10n + 500}{\frac{1}{2}n^2} = 0$$

O-notation Example:

3. If $f(n) = 100 \log n$ and $g(n) = \sqrt{n}$ then $f(n) = O(g(n))$

why?

$$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = \lim_{n \to +\infty} \frac{100 \log n}{\sqrt{n}} = 0$$

O-notation — general rules

1. $n^c = O(n^{c+1})$ for all constant $c > 0$

2. $n^c = O(2^n)$ for all constant $c > 0$

3. $c^n = O((c+1)^n)$ for all constant $c > 1$

O-notation — general rules

4. Transitivity: (“if $a \leq b$ and $b \leq c$ then $a \leq c$”)

if $f(n) = O(g(n))$ and $g(n) = O(h(n))$ then $f(n) = O(h(n))$

Example. Applying rule 1 repeatedly implies, e.g., $n^4 = O(n^{100})$

5. Change of variable

Example. Prove $(\log n)^c = O(n)$ for all $c > 0$

Define the variable $m = \log n$ and so $n = 2^m$

So we need to prove $m^c = O(2^m)$ instead which holds by rule 2

O-notation — example

- List the following functions based on their asymptotic value in increasing order
- $(\log n)^{1/3}$ $\quad 2^{\sqrt{\log n}}$ $\quad \log \log n$
- Ordering is $\quad \log \log n \quad (\log n)^{1/3} \quad 2^{\sqrt{\log n}}$

O-notation — example

- Proof:
- Part 1: $\log \log n = O((\log n)^{1/3})$
- Change of variable $m = (\log n)^{1/3}$
- So we need to prove $\log(m^3) = O(m)$
- But we also have $\log(m^3) = 3 \log m = O(\log m)$
- Since we know $\log m = O(m)$, by transitivity, $\log(m^3) = O(m)$

O-notation — example

- Proof:
- Part 2: $(\log n)^{1/3} = O(2^{\sqrt{\log n}})$
- Change of variable: $m = \sqrt{\log n}$
- So $(\log n)^{1/3} = (\sqrt{\log n})^{2/3} = m^{2/3}$
- We need to prove that $m^{2/3} = O(2^m)$
- But this already holds by rule 2

$(a^b)^c = a^{b \cdot c}$

Ω-notation

- Ω-notation is informally "the $\geq$ operator in the limit"
- So $f(n) = \Omega(g(n))$ if and only if $g(n) = O(f(n))$
- For instance, $n^2 = \Omega(n)$
- So we can use all the previous rules we had for O-notation to get new results for Ω-notation

Θ-notation

- Θ-notation is informally "the $=$ operator in the limit"
- $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
- For instance, $100n = \Theta(n)$ but $n^2 \neq \Theta(n)$
- Again, we can apply previous rules to prove a bound in Θ-notation
- Example: Prove $50n = \Theta(2^{\log n})$
- Firstly $50n = \Theta(n)$ and secondly $2^{\log n} = n$

o, ω-notations

- O, Ω-notation are analogues of " $\leq$, $\geq$ " operators:
  - They allow for "equality"
  - We can have $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ at the same time

o, ω-notations

- What about “strict inequality”?
  - o-notation (“the < operator in limit”):
    - when $f(n) = O(g(n))$ but $f(n) \neq \Omega(g(n))$
    - Alternatively $\lim_{n \to +\infty} \dfrac{f(n)}{g(n)} = 0$
  - ω-notation (“the > operator in limit”):
    - when $f(n) = \Omega(g(n))$ but $f(n) \neq O(g(n))$
    - Alternatively $\lim_{n \to +\infty} \dfrac{f(n)}{g(n)} = +\infty$

o, ω-notations

- Useful for showing an algorithm is strictly faster/slower than another one

An Example of Algorithm Design Process

Problem

- We have n people in a party for some **odd** number n
- We know that strictly more than half of these people belong to some hidden community
- We can ask two people in the party to **greet** each other:
  - If **both** people belong to this hidden community then they greet each other warmly
  - Otherwise, they say they do not each other
- Design an algorithm that finds every member of this hidden community using smallest number of greetings possible

Where to start?

- Gain intuition about the problem
- See if you can solve a "puzzle" version of this problem for some reasonably small value of n
- We have 15 people in a party and 8 of them belong to a hidden community
- Can we find all people in this hidden community?
  - There is a simple solution with 105 greetings
  - Can you solve the problem with ≤ 25 greetings?