Week 1: Introduction & Computational Models

Instructor: Warut Suksompong

National University of Singapore

CS3230
Semester 2, 2021–22

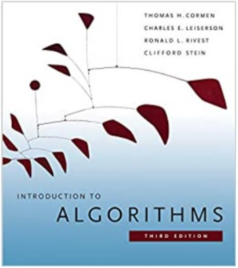
Introduction

- Module objectives:
  - To study algorithms in a formal way (through the lens of mathematics)
  - To learn tools to analyze the performance of algorithms
  - To learn techniques to design an efficient algorithm

- After the module, students should be able to
  - Perform analysis of the asymptotic performance of algorithms.
  - Design efficient algorithms to solve problems.
  - Able to comment on correctness of designed algorithms.
  - Comment on (inherent) hardness of a problem.

Introduction

- Prerequisites:
  - CS2010 or CS2020 or CS2040/C/S Data Structures and Algorithms
  - CS1231/S or MA1100 Discrete Structures
- Textbook:
  - CLRS: Introduction to Algorithms, 3rd edition, by Cormen, Leiserson, Rivest & Stein, 2009



Logistics

- Lecture: Thursday 14:00–16:00 via Zoom, recorded
  - I will stick around to answer questions after lecture.
  - Lecture notes and slides will be posted on LumiNUS.
  - My email address: warut@comp.nus.edu.sg
  - Due to the large number of students, please don't email me questions about course material. Post them on the LumiNUS forum instead.
- Tutorial: 1 hour each week, starting from week 3
  - 18 slots (10 F2F, 8 via Zoom)
  - Schedule and list of tutors in "Tutorials" folder on LumiNUS (soon).
  - Each tutor will also hold one hour of office hours per week.
- LumiNUS forum: Ask questions here!
  - I will monitor it together with TAs.
  - The posts will show your "nickname". You can set your nickname in your LumiNUS user profile.

Tentative Schedule

Week	Date	Topic
1	13 Jan	Intro & computational models
2	20 Jan	Asymptotic analysis
3	27 Jan	Iteration, recursion & divide-and-conquer
4	3 Feb	Average-case analysis & randomized algorithms
5	10 Feb	Hashing
6	17 Feb	Pattern matching & Streaming
7	3 Mar	**Midterm (during lecture slot)**
8	10 Mar	Amortized analysis
9	17 Mar	Dynamic programming
10	24 Mar	Greedy algorithms
11	31 Mar	Reductions & computational complexity
12	7 Apr	Reductions & computational complexity (cont.)
13	14 Apr	No class (NUS Well-Being Day)

Assessment

- Assignments (36%):
  - 12 assignments, one for each lecture (last assignment will be a revision)
  - Each assignment consists of 3 questions.
  - There is one question graded for correctness in every odd-numbered assignment $(1, 3, 5, 7, 9, 11)$. All other questions are graded for effort.
  - Each correctness-based question worth 3.5% (graded out of 7 points)
  - Each effort-based question worth 0.5% (graded out of 1 point)
  - Total $= 6 \cdot 3.5\% + 30 \cdot 0.5\% = 36\%$
- Assignment released on lecture day (Thursday), due 11:59pm Sunday of the following week (except Assignment 12)
- Assignment schedule + grader list posted on LumiNUS (in the "Assignments" folder)
- No late assignment will be accepted.
- For grading enquiries, please check directly with the relevant grader.

Assessment

- Continuous assessment (40%):
  - Assignments (36%)
  - Tutorial attendance (4%)
  - Bonus points (up to 6%)
    - Free for everyone! (2%)
    - Two programming assignments (total of 4%)
  - The total you earn from continuous assessment cannot exceed 40%.
- Exams (60%):
  - Midterm (30%): **3 March** (during lecture slot), **14:00–16:00**
  - Final exam (30%): **26 April** (Tuesday), **17:00–19:00**
  - Mode of both exams will be online.

Problems v Algorithms

- Problems provide the what. Algorithms provide the how.
- Example of a computational problem: Multiplication
  - Input: Two numbers x and y
  - Output: The product $x \cdot y$
- An algorithm is a well-defined procedure for finding a correct solution to the input.
- There can be many algorithms for a particular problem.
- The grade-school algorithm for multiplication is just one algorithm for the problem (in fact, not the best one when the input is large!)

Algorithms

- In this module, we will focus on two key aspects of algorithms: correctness and efficiency (i.e., running time).
- For correctness, we typically want algorithms to be correct on every valid input.
- This is known as worst-case correctness.
- Sometimes, relaxed notions of correctness are considered:
  - Correct on a random input
  - Correct on every input with high probability
  - Approximately correct

Algorithms

- The running time measures the number of steps executed by an algorithm as a function of the input size.
  - What each step is depends on the computational model used.
- We must specify what the notion of input size is.
  - If input is an array, its size is typically the length of the array.
  - If input is a number, its size is typically the length of its binary representation.
  - If input is a graph, its size is typically the number of vertices and edges in the graph.
- The (worst-case) running time of an algorithm is the maximum number of steps executed when run on an input of size n.
- Besides correctness and running time, other considerations include simplicity, space usage, energy consumption, parallelism, and fairness/ethics.

Comparison Model

- The input is an array of n numbers.

3	7	5	8	4

- The algorithm can compare any two elements in one time unit: Is $x > y$, $x < y$, or $x = y$?
- No other operations on the elements (e.g., addition, subtraction) are allowed.
- Running time $=$ total number of comparisons made
- The array can be manipulated (e.g., permuted or broken into subarrays) at no cost.

Comparison Model: Maximum

- Problem: Given an array A of n distinct elements (denoted by $A_1, \ldots, A_n$), find the largest element in A.
- Here is one algorithm:

```
RunThru(A)
cur = 1
n = A.length
for i = 2 to n
        if A_i > A_cur
                cur = i
return A_cur
```

- RunThru makes exactly $n - 1$ comparisons for any input.

Maximum

- **Claim:** Every algorithm solving the Maximum problem must make $\geq n-1$ comparisons!
- Let's look at some small cases first.
  - $n = 2$: Need to do 1 comparison.

?	?

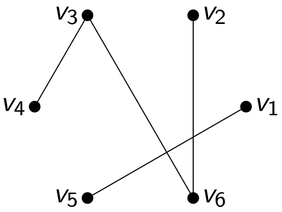
  - $n = 3$: If the algorithm makes just 1 comparison (say, between the first and second elements), then the third element is left untouched.

?	?	?

    The third element could be either very large (in which case it is the maximum), or very small (in which case it is not the maximum). So 2 comparisons are needed.

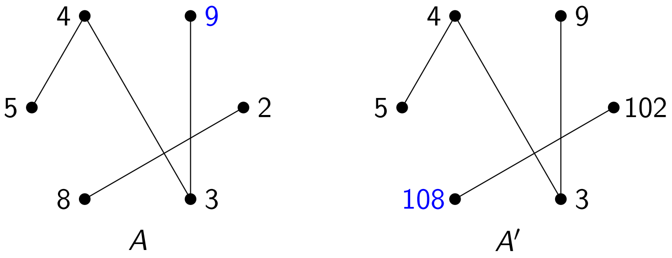
Maximum

- To prove the claim for general n, fix an algorithm $\mathcal{M}$ that solves the Maximum problem on all inputs using $< n - 1$ comparisons.
- Take an input array A on which $\mathcal{M}$ makes $< n - 1$ comparisons.
- Construct a graph G on n nodes $v_1, \ldots, v_n$, where there is an edge between nodes v_i and v_j iff $\mathcal{M}$ compares A_i and A_j.



Maximum

- Since G has $< n - 1$ edges, it is disconnected.



- Let A_i be the maximum element of A.
- Consider a different input A', where all numbers in a different connected component than A_i are increased by a huge amount.
- $\mathcal{M}$ cannot distinguish between A and A', a contradiction.

Maximum

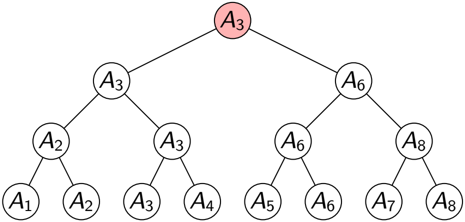
- This proof is an example of an adversary argument.
- The input is decided on-the-fly by an adversary who keeps its options open about what the actual input is.
- The adversary makes sure that if the algorithm makes too few comparisons, then:
  - There are two different inputs which are consistent with the results of these comparisons.
  - And yet, the solutions for the two inputs are different.

Second Largest

- What about finding the second largest element?
- One way is to find the maximum first using our previous algorithm, then use this algorithm again to find the maximum among the remaining $n-1$ numbers.
- This requires $(n-1)+((n-1)-1)=2n-3$ comparisons.
- Is this the best we can do?
- No!
- Charles Lutwidge Dodgson (better known as Lewis Carroll, author of *Alice in Wonderland*), came up with an algorithm requiring fewer comparisons.

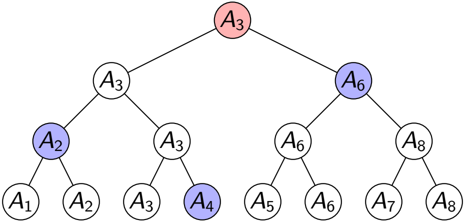
Second Largest

- Instead of finding the maximum using the previous algorithm, we do so by comparing elements using a knockout tournament structure.



- The winner of this "tournament" is the maximum element.
- Since every non-winner has "lost" exactly once, this algorithm also solves the Maximum problem using $n - 1$ comparisons.

Second Largest



- Observe that the second-largest element must have lost to the winner.
- We can therefore find the maximum among the $\lceil \lg n \rceil$ elements that lost to the winner using $\lceil \lg n \rceil - 1$ comparisons ($\lg$ denotes $\log_2$).
- The total number of comparisons is therefore $(n-1) + (\lceil \lg n \rceil - 1) = n + \lceil \lg n \rceil - 2$.
- This is known to be optimal!

Sorting

- For the Sorting problem, we want to order all elements in the array A of distinct numbers.
- For simplicity, assume that n is a power of two.
- **Claim:** There is a sorting algorithm that requires $\leq n \lg n - n + 1$ comparisons.
- For example, the Merge Sort algorithm (covered in Lecture 3).
- The algorithm divides A into two equal halves and merges them into one sorted array.
- If each half contains $n/2$ elements, the merging step takes $n - 1$ comparisons.

Sorting

- **Claim:** Every sorting algorithm must make $\geq \lg(n!)$ comparisons.
- Initially, there are $n!$ permutations of the set $\{1, \ldots, n\}$ that the adversary could choose as the array A. Call this set $\mathcal{U}$.
- Each permutation in $\mathcal{U}$ needs to be ordered differently to get sorted.
- When a query comes in ("Is $A_i > A_j$?"), the adversary checks whether $\mathcal{U}_{\text{yes}} = \{A \in \mathcal{U} : A_i > A_j\}$ is of size at least $|\mathcal{U}|/2$.
  - If so, it replies Yes to the algorithm and sets $\mathcal{U}$ to be $\mathcal{U}_{\text{yes}}$.
  - Else, it replies No and sets $\mathcal{U}$ to be $\mathcal{U} \setminus \mathcal{U}_{\text{yes}}$.
- If the algorithm makes $< \lg(n!)$ comparisons, $\mathcal{U}$ will still contain at least two permutations, since its size decreases by at most half with each comparison.
- The algorithm will order these two permutations in the same way, and will be wrong on at least one of them.

Sorting

- How big is $\lg(n!)$ then?
- It is known that $n! \geq (n/e)^n$, so

$$\lg(n!) \geq n \lg\left(\frac{n}{e}\right) = n \lg n - n \lg e \approx n \lg n - 1.44n.$$

- This means that roughly $n \lg n$ comparisons are both required and sufficient for sorting n numbers.

Query Model: Strings

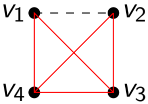
- Suppose the input is a string of n bits (each bit is 0 or 1).
- With each query, the algorithm can find out one bit of the string.
- Consider the problem of deciding whether an input string is the all-0 string or not.
- By checking whether each bit is 0, n queries suffice.
- **Claim:** n queries are also necessary.
  - Suppose $\mathcal{M}$ is an algorithm making $< n$ queries.
  - Consider an adversary that replies to each of $\mathcal{M}$'s queries with 0.
  - After $\mathcal{M}$ halts, there is still at least one unqueried bit, say the i-th bit.
  - The input may be all-0, or it may be 0 in all bits except the i-th bit.

Query Model: Graphs

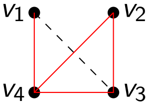
- The input is the (symmetric) adjacency matrix of an n-node undirected graph.
- With each query, the algorithm can find out one entry of the matrix (i.e., whether an edge is present between two chosen nodes or not).
- Consider the problem of deciding whether a graph is connected or not.
- **Claim:** $\binom{n}{2}$ queries are necessary.
  - Suppose $\mathcal{M}$ is an algorithm making $< \binom{n}{2}$ queries.
  - When $\mathcal{M}$ makes a query, the adversary tries not adding this edge, but adding all remaining unqueried edges.
  - If the resulting graph is connected, the adversary replies 0 (i.e., edge does not exist).
  - Else, the adversary replies 1 (i.e., edge exists).

Query Model: Graphs

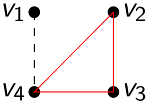
- **Example:** Consider a graph with 4 nodes v_1, v_2, v_3, v_4.
- First query: $(v_1, v_2) \to$ 0 (edge does not exist)



- Second query: $(v_1, v_3) \to$ 0 (edge does not exist)



- Third query: $(v_1, v_4) \to$ 1 (edge exists)

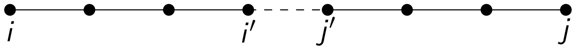


Query Model: Graphs

- At every stage, setting all unqueried entries to 1 will make the graph connected.
- At the end, since $\mathcal{M}$ made $< \binom{n}{2}$ queries, at least one entry of the adjacency matrix is unqueried.
- The adversary considers the graph G_0 obtained by setting all unqueried entries to 0, and the graph G_1 obtained by setting all unqueried entries to 1.
- By the first bullet point above, G_1 is connected.
- **Claim:** G_0 is disconnected.
- This claim suffices to finish the proof.

Query Model: Graphs

- **Claim:** G_0 is disconnected.
- Let (i, j) be an unqueried pair of nodes.
- Suppose for contradiction that there is a path between i and j in G_0.
- The adversary replied 1 to all edges on this path.
- Let (i', j') be the edge on this path that was queried last. Consider the graph when the adversary receives this query.



- Even if the adversary answers 0 on the edge (i', j'), when it sets all unqueried edges (including (i, j)) to 1, the graph must be connected.
- So the adversary should have answered 0 on (i', j'), a contradiction!