CSEN703
Analysis and Design of Algorithms

Course Material

- **Reference:** Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stien (2001). *Introduction to Algorithms (2nd edition)*. The MIT Press. ISBN 0-262-03293-7
- **Reference:** Richard Neapolitan and Kumarss Naimipour (2004). *Foundations of Algorithms Using Java Pseudocode*. Jones and Bartlett Publishers. ISBN 0-7637-2129-8
- **Reference:** Russ Miller and Quentin Stout (2002). *Parallel Algorithms for Regular Architectures: Meshes and Pyramids*. The MIT Press. ISBN 9780262279833
- **Internet:**

  `met.guc.edu.eg/courses/Winter2017_CSEN703.aspx`

Six Reasons Why You Should Study This Course!

1. Algorithms are bread-and-butter for computer science.
2. It teaches you how to be precise about algorithm efficiency.
3. It teaches you a number of clever algorithm design techniques
   - with fancy names like divide-and-conquer, dynamic programming, greedy algorithms.
4. It expands your horizons onto the realm of parallel algorithms.
5. It exposes you to algorithms used in diverse fields, whereby
   (a) you get exposed to various computational problems, and
   (b) you get to see how really clever people design algorithms.
6. Like all computer science courses, it is fun!

Three Things You Will *Not* Get Out of This Course!

1. You will not get a recipe for algorithm design in *all* applications.
   - There is no algorithm for algorithm design.
2. You will not learn about *all* algorithms that were ever designed.
   - Not even all the important/classical ones.
3. You will not learn about algorithms for *all* parallel models of computation.

Course Assessment

- Your grade in this course will be based on your scores in one final exam, one midterm exam, and a bunch of homework scores.
- Weights

Homeworks	15%
Quizzes	20%
Midterm Exam	25%
Final Exam	40%

Asymptotic Analysis

Lecture 1

September 9, 2017

Problems

- A problem is a pair (i, Q), where i is an **instance** and Q is a **question**.
- **Example**: $P_1 = (G, Q_1)$, where
  - G is a graph and
  - Q is "Is G connected?"
- A **decision problem** is a problem whose question is a yes/no question.
  - P_1 is a decision problem.
- **Example of a non-decision problem** : $P_2 = (S, Q_2)$, where
  - S is a sequence of integers and
  - Q_2 is "What is a permutation of S whose elements appear in ascending order?"

Algorithms

- Intuitively, an algorithm is a process (typically a sequence of steps; hence, procedure) whereby one may solve some problem.
  - The steps must be elementary/primitive.
- The algorithm takes the instance as input, and outputs an answer to the question.
  - The algorithm is said to *transform* the input into the output.
- If, for some instance, the process does not lead to an answer, then it is not an algorithm.
  - Thus, an algorithm never runs forever. (A procedure may.)
- **Why can't an algorithm be a simple table look-up procedure?**

Hardness

- Every problem has an intrinsic degree of **hardness**.
- There are two different (but related) ways of determining the hardness of a given problem, P_1:
  1. The amount of resources required by an algorithm that solves P_1.
     - **Remember:** Sometimes no such algorithm exists.
  2. The hardness of *reducing* P_1 to another problem P_2 + the hardness of P_2.
- The reduction takes place in two steps:
  1. Mapping an instance of P_1 to an instance of P_2.
  2. Mapping an answer to P_2's question to an answer to P_1's question.

Example

- $P_1 = (S, Q_1)$, where
  - S is a sequence of integers and
  - Q_1 is "What is a permutation of S whose elements appear in ascending order?"
- $P_2 = (S, Q_2)$, where
  - S is a sequence of integers and
  - Q_2 is "What is a permutation of S whose elements appear in descending order?"
- Evidently, P_1 is reducible to P_2 (and vice versa).
  1. The instance mapping is the identity mapping.
  2. The answer mapping is the sequence reversal mapping.
- Thus, P_1 is as hard as solving P_2 and reversing the answer.

A Catch

- But, instead of going through the reduction, we can directly come up with an algorithm for P_1.
- Thus, we have two algorithms (and possibly more) for P_1.
  - Note that the composition of P_2's algorithm with the reduction defines an algorithm for P_1.
- Which algorithm determines the hardness of P_1?
  - The one that provides the smallest measure.
- This measure of the amount of resources an algorithm needs is its **complexity**.
- But now we need to be precise about what we mean by "complexity" and about how to determine complexity of algorithms in a consistent manner.

Models of Computation

- To precisely discuss algorithm complexity, we need to explicate our assumptions about the model of computation used.
- For the purpose of complexity analysis, models of computation are either **sequential** or **parallel**.
- Such models consist of one (if sequential) or more (if parallel) processors and some memory space.
- The processor has a repertoire of **basic operations** that it can directly perform. All basic operations are performed in **the same amount of time**.
- The memory space is a (possibly partitioned) sequence of **fixed-size locations**, each with a **unique address**.
- Models vary primarily with respect to the mode whereby processor(s) access the memory space.

The Random Access Machine

- The **RAM** is the standard sequential model of computation.
  - In complexity theory, the multi-tape Turing machine is the standard. They are equivalent, though.
- The processor can access **any memory location**, in the **same amount of time**.
- Typically, RAM basic operations are the primitives of a typical high-level programming language:
  - arithmetic operations, logical operations, comparisons, assignments, etc.
- Parallel models will be discussed in the second half of the course.

Classification of Complexity Measures

- There are two orthogonal dimensions along which we can classify complexity measures:
  1. **Resource Type:** Time, Space, Work.
  2. **Analysis Technique:** Worst-case, Average-case, Best-case.
- Thus, we can talk about *worst-case time complexity*, *average-case space complexity*, *best-case work complexity*, etc.
- All complexity measures are defined as functions of the **input size**. (More on this below.)
- In this course, we are primarily interested in **worst-case time complexity**.

Resource Types

- **Space Complexity:** The **number of memory locations accessed** by the algorithm.
- **Work Complexity:** The **number of basic operations performed** by the algorithm.
- **Time Complexity:** The **number of time units used** up by the algorithm. A time unit is the fixed amount of time it takes to perform a basic operation.
- For sequential models, work complexity = time complexity.

Analysis Techniques

- **Worst-case:** The **maximum** amount of a resource needed, taken over the set of all instances, for a given input size.
- **Average-case:** The **mean** of the amount of a resource needed, taken over the set of all instances, for a given input size. A probability distribution over the set of instances is required.
- **Best-case:** The **minimum** amount of a resource needed, taken over the set of all instances, for a given input size.

Dependence on the Input

- Complexity defined as a function of the **value/magnitude** of the input is not very helpful.
- Cluster inputs into classes of *same-size* inputs.
- Thus, define complexity as a function of the **input size**.
- Dependence on the input's value features in the distinction between worst-case, average-case, and best-case complexity.
- Where $r \in \{t, s, w\}$ and $C \in \{W, A, B\}$, $C_r(n)$ denotes the C-case r complexity for input size n.
  - When the subscript is omitted, assume it is t.
  - Typically, $T(n)$ is used as a shorthand for $W_t(n)$.

A Note on "Input Size"

- Strictly speaking, the input size is the length of a *reasonable* string encoding of the input.
  - A reasonable encoding is any non-unary encoding; unary encodings are *expensive*.
- In most cases, however, a more intuitive notion of size is equivalent.
  - The number of elements in an array.
  - The number of nodes and arcs of a graph.
  - The dimensionality of a matrix.
- In other cases, the exact size of the input is significant.
  - The worst-case time complexity of the obvious prime-checking algorithm is linear in the *value* of the input, but exponential in the *size* of the input.

Asymptotic Analysis

- It is often complex to get the exact complexity of an algorithm.
- A convenient estimation involves describing the function $C_r(n)$ by its asymptotic behavior—its behavior on large values of n.
- This kind of complexity analysis is called **asymptotic analysis**.
- The assumption is that it is the behavior of programs on large inputs that really matters.
- Asymptotic analysis is convenient since we only retain higher order terms without worrying about lower order terms or constant coefficients.

Asymptotic Notation: O, Ω, and Θ

- Let $f, g : \mathbb{N} \longrightarrow \mathbb{R}^+$ be two functions.
- $f(n) = O(g(n))$ if

  there are $c \in \mathbb{R}^+$ and $n_0 \in \mathbb{Z}^+$ such that

  for every $n \in \mathbb{Z}^+$, $n \geq n_0 \rightarrow f(n) \leq cg(n)$.
- When $f(n) = O(g(n))$, we say that $g(n)$ is an **asymptotic upper bound** for $f(n)$.
- $f(n) = \Omega(g(n))$ iff $g(n) = O(f(n))$
- When $f(n) = \Omega(g(n))$, we say that $g(n)$ is an **asymptotic lower bound** for $f(n)$.
- $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$.

Asymptotic Notation: o and ω

- Let $f, g : \mathbb{N} \longrightarrow \mathbb{R}^+$ be two functions.
- $f(n) = o(g(n))$ if

  for every $c \in \mathbb{R}^+$, there is an $n_0 \in \mathbb{Z}^+$ such that

  for every $n \in \mathbb{Z}^+$, $n \geq n_0 \rightarrow f(n) < cg(n)$.
- Note: $f(n) = o(g(n)) \rightarrow f(n) = O(g(n))$
- $f(n) = \omega(g(n))$ iff $g(n) = o(f(n))$
- Note: $f(n) = \omega(g(n)) \rightarrow f(n) = \Omega(g(n))$

Examples

- $n = o(n^2)$.
- $n = o(2^n)$.
- $\log n = o(n)$. (Does the base of the logarithm matter?)
- $n \log n = o(n^2)$.
- $\log \log n = o(\log n)$

In general

- $n^i = o(n^j)$, for $i < j$.
- $n^i = o(a^n)$, for $a > 1$.
- $f(n) = O(f(n))$. But $f(n) \neq o(f(n))$.

Example

- $3 + \cos(n) = \Theta(1)$.

Example

- $3 + \cos(n) = \Theta(1)$.

  Proof. We know that, for all n,

  $$-1 \leq \cos(n) \leq 1.$$

  It follows that, for all n,

  $$2 \leq 3 + \cos(n) \leq 4.$$

  Thus,

  $$3 + \cos(n) = O(1) \text{ (with } c = 4 \text{ and } n_0 = 1)$$

  and

  $$3 + \cos(n) = \Omega(1) \text{ (with } c = 2 \text{ and } n_0 = 1).$$

  Hence, $3 + \cos(n) = \Theta(1)$.

The Limit Test

- Let $L = \lim_{n \to \infty} \frac{f(n)}{g(n)}$.
  1. $L = 0 \rightarrow f(n) = o(g(n))$.
  2. $L = \infty \rightarrow f(n) = \omega(g(n))$.
  3. $L \in \mathbb{R}^+ \rightarrow f(n) = \Theta(g(n))$.
  4. L is undefined $\rightarrow$ test fails.
- Compute the limit by pretending that the domain of f and g is $\mathbb{R}$.
- Use the limit test to verify the examples of the previous two slides.
- **Example**: Show that $n^2 + 3n + 6 = \Theta(n^2)$.

Summations

- Interested in summations of the form: $\sum_{i=1}^{n} f(i)$.
- Two methods to get asymptotic bounds on the summation:
  1. Find an equivalent closed-form expression. (Not always easy.)
  2. Use integration.
     - For a non-decreasing f,
     $$\int_0^n f(x)dx \leq \sum_{i=1}^{n} f(i) \leq \int_1^{n+1} f(x)dx$$
     - **Why?**
     - Note that, strictly-speaking, the functions in the integrals cannot be f.

Example

- Prove that $\sum_{i=1}^{n} i = \Theta(n^2)$.
  - Closed form (using a combinatorial argument):
    $$\frac{(n+1)n}{2} = \Theta(n^2)$$
  - Use integration. (Check the following slide.)

Example: Proof Using Integration

$$\int_0^n x dx \leq \sum_{i=1}^{n} i \leq \int_1^{n+1} x dx$$

$$\rightarrow \quad \frac{x^2}{2}\Big|_0^n \leq \sum_{i=1}^{n} i \leq \frac{x^2}{2}\Big|_1^{n+1}$$

$$\rightarrow \quad \frac{n^2}{2} \leq \sum_{i=1}^{n} i \leq \frac{(n+1)^2}{2} - \frac{1}{2}$$

$$\rightarrow \quad \frac{n^2}{2} \leq \sum_{i=1}^{n} i \leq \frac{(n+1)^2}{2} = \frac{n^2}{2} + n + \frac{1}{2}$$

$$\rightarrow \quad \frac{n^2}{2} \leq \sum_{i=1}^{n} i \leq \frac{n^2}{2} + n^2 + n^2, \text{ for } n \geq 1$$

$$\rightarrow \quad \frac{n^2}{2} \leq \sum_{i=1}^{n} i \leq \frac{5n^2}{2}, \text{ for } n \geq 1$$

$$\rightarrow \quad \sum_{i=1}^{n} i = \Theta(n^2)$$

Points to take home

- Problems.
- Algorithms.
- Resource types.
- Analysis techniques.
- Asymptotic analysis: $O, \Omega, \Theta, o, \omega$.
- The limit test.
- Summations.

Next time

- Divide-and-Conquer.