Information about the Lecturer

- Dr. Mordecai Golin
- Office: 3559
- Email: golin@cs.ust.hk
- Office hours: Just drop by or send email for appointment.
Textbook and Lecture Notes


References: Recommendations

1. Dave Mount: Lecture Notes
   Available on course web page


About COMP 271

A continuation of COMP 171, with advanced topics and techniques. Main topics are:


2. Analysis of algorithms (goes hand in hand with design).

3. Graph Algorithms.


Prerequisite: Discrete Math. and COMP 171
We assume that you know

- Sorting: Quicksort, Insertion Sort, Mergesort, Radix Sort (with analysis). Lower Bounds on Sorting.

- Big $O()$ notation and simple analysis of algorithms

- Heaps


- Balanced Binary Search Trees (dictionaries)

- Hashing
Tentative Syllabus

- *Introduction & Review*

- *Maximum Contiguous Subarray:* case study in algorithm design

- *Divide-and-Conquer Algorithms:* Mergesort, Polynomial Multiplication, Randomized Selection

- *Graphs:*
  - Review: Notation, Depth/Breadth First Search
  - Cycle Finding & Topological Sort
  - Minimum Spanning Trees: Kruskal’s and Prim’s algorithms
  - Dijkstra’s shortest path algorithm

- *Dynamic Programming:* Knapsack, Chain Matrix Multiplication, Longest Common Subsequence, All Pairs Shortest Path

- *Greedy algorithms:* Activity Selection, Huffman Coding

- *Complexity Classes:* Nondeterminism, the classes P and NP, NP-complete problems, polynomial reductions
Other Information

- Lecture and tutorial schedule, TAs.
  No Tutorials this week.
  Tutorials start on **February 14, 2003**.

- Question banks: To help you review

- Assignments: 4, each worth 5% of grade
  Midterm: worth 35% of grade
  Final exam (comprehensive): worth 45% of grade.

- Final Grade. Will be curved based on class performance. Guaranteed Grades:
  Average of $\geq 95 \Rightarrow A$
  Average of $\geq 85 \Rightarrow B$
  Average of $\geq 75 \Rightarrow C$
  Average of $\geq 65 \Rightarrow D$
Classroom Etiquette

- No pagers and cell phones – switch off in classroom.

- Latecomers should enter QUIETLY.

- No loud talking during lectures.

- But please ask questions and provide feedback.
Lecture 1: Introduction

**Computational Problems and Algorithms**

**Definition:** A computational problem is a specification of the desired input-output relationship.

**Definition:** An instance of a problem is all the inputs needed to compute a solution to the problem.

**Definition:** An algorithm is a well defined computational procedure that transforms inputs into outputs, achieving the desired input-output relationship.

**Definition:** A correct algorithm halts with the correct output for every input instance. We can then say that the algorithm solves the problem.
Example of Problems and Instances

**Computational Problem: Sorting**

- **Input:** Sequence of $n$ numbers $\langle a_1, \cdots, a_n \rangle$.

- **Output:** Permutation (reordering)
  
  $\langle a'_1, a'_2, \cdots, a'_n \rangle$

  such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

**Instance of Problem:** $\langle 8, 3, 6, 7, 1, 2, 9 \rangle$
**Example of Algorithm: Insertion Sort**

In-Place Sort: uses only a fixed amount of storage beyond that needed for the data.

**Pseudocode:** $A$ is an array of numbers

for $j = 2$ to length($A$)  
{   
key = $A[j]$;  
$i = j - 1$;  
while ($i \geq 1$ and $A[i] > key$)  
{   
$i = i - 1$;  
}  
$A[i + 1] = key$;  
}

**Pause:** How does it work?
**Insertion Sort: an Incremental Approach**

To sort a given array of length $n$, at the $i$th step it sorts the array of the first $i$ items by making use of the sorted array of the first $i - 1$ items in the $(i - 1)$th Step.

**Example:** Sort $A = \langle 6, 3, 2, 4 \rangle$ with Insertion Sort.

**Step 1:** $\langle 6, 3, 2, 4 \rangle$

**Step 2:** $\langle 3, 6, 2, 4 \rangle$

**Step 3:** $\langle 2, 3, 6, 4 \rangle$

**Step 4:** $\langle 2, 3, 4, 6 \rangle$
Analyzing Algorithms

Predict resource utilization

1. Memory (space complexity)

2. Running time (time complexity)

Remark: Really depends on the model of computation (sequential or parallel). We usually assume sequential.
Analyzing Algorithms – Continued

**Running time:** the number of *primitive operations* used to solve the problem.

**Primitive operations:** e.g., addition, multiplication, comparisons.

**Running time:** depends on problem instance, often we find an upper bound: $F(\text{input size})$

**Input size:** rigorous definition given later.

1. **Sorting:** number of items to be sorted

2. **Multiplication:** number of bits, number of digits.

3. **Graphs:** number of vertices and edges.
Three Cases of Analysis

Best Case: constraints on the input, other than size, resulting in the fastest possible running time.

Worst Case: constraints on the input, other than size, resulting in the slowest possible running time. Example. In the worst case Quicksort runs in $\Theta(n^2)$ time on an input of $n$ keys.

Average Case: average running time over every possible type of input (usually involve probabilities of different types of input). Example. In the average case Quicksort runs in $\Theta(n \log n)$ time on an input of $n$ keys. All $n!$ inputs of $n$ keys are considered equally likely.

Remark: All cases are relative to the algorithm under consideration.
Three Analyses of Insertion Sorting

The number of comparisons needed is equal to

\[
\underbrace{1 + 1 + 1 + \cdots + 1}_{n-1} = n - 1 = \Theta(n).
\]

The number of comparisons needed is equal to

\[
1 + 2 + \cdots + (n - 1) = \frac{n(n - 1)}{2} = \Theta(n^2).
\]

**Average Case:** $\Theta(n^2)$ assuming that each of the $n!$ instances are equally likely.
Big Oh

\[ f(n) = O(g(n)) : g(n) \text{ is an asymptotically upper bound for } f(n). \]

\[ O(g(n)) = \{ f(n) : \exists \ c > 0 \text{ and } n_0 > 0 \]
\[ \text{such that } 0 \leq f(n) \leq cg(n) \]
\[ \text{for all } n \geq n_0 \}. \]

Remark: “\( f(n) = O(g(n)) \)” means that
\[ f(n) \in O(g(n)). \]

Examples:
(1) \( n^2 + 2n = O(n^2). \)
(2) \( 200n^2 - 100n = O(n^2). \)
(3) \( n \log_2 n = O(n^2). \)
(4) \( n^2 \log_2 n \neq O(n^2). \)
(5) \( \forall a, b > 1, \log_a n = O(\log_b n). \)
Big Omega

\[ f(n) = \Omega(g(n)) : \text{ } g(n) \text{ is an asymptotically lower bound for } f(n). \]

\[ \Omega(g(n)) = \{ f(n) : \exists \ c > 0 \text{ and } n_0 > 0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \} \]

Examples:
(1) \( 200n^2 - 100n = \Omega(n^2) = \Omega(n) = \Omega(1) \).
(2) \( n^2 = \Omega(n) \).
(3) \( n^2 \neq \Omega(n^2 \log n) \).
(3) Does \( f(n) = O(g(n)) \) imply \( g(n) = \Omega(f(n)) \)?
(4) Does \( f(n) = \Omega(g(n)) \) imply \( g(n) = O(f(n)) \)?
Big Theta

\[ f(n) = \Theta(g(n)) \]: \( g(n) \) is an \textit{asymptotically tight bound} for \( f(n) \).

\[ \Theta(g(n)) = \{ f(n) : \exists \ n_0 > 0, \ c_1 > 0 \text{ and } c_2 > 0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \} \].

Examples:
(1) \( 5n^2 - 2n + 5 = \Theta(n^2) \)
(2) \( 5n^2 - 2n + 5 = \Theta(n^2 + \log n) \)
Note that if
\[ f(n) = \Theta(g(n)), \]
then
\[ f(n) = \Omega(g(n)) \quad \text{and} \quad f(n) = O(g(n)). \]

\[ \bullet \quad \bullet \quad \bullet \quad \bullet \]

In the other direction, if
\[ f(n) = \Omega(g(n)) \quad \text{and} \quad f(n) = O(g(n)), \]
then
\[ f(n) = \Theta(g(n)). \]

\[ \bullet \quad \bullet \quad \bullet \quad \bullet \]

\[ f(n) = \Theta(g(n)) \iff f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n)) \]
Some thoughts on Algorithm Design

- *Algorithm Design*, as taught in this class, is mainly about designing algorithms that have small big $O()$ running times.

- “All other things being equal”, $O(n \log n)$ algorithms will run more quickly than $O(n^2)$ ones and $O(n)$ algorithms will beat $O(n \log n)$ ones.

- Being able to do good algorithm design lets you identify the *hard parts* of your problem and deal with them effectively.

- Too often, programmers try to solve problems using brute force techniques and end up with slow complicated code! A few hours of abstract thought devoted to algorithm design could have speeded up the solution substantially *and* simplified it.
Note: After algorithm design one can continue on to *Algorithm tuning* which would further concentrate on improving algorithms by cutting down on the *constants* in the big $O()$ bounds. This needs a good understanding of both algorithm design principles and efficient use of data structures. In this course we will not go further into algorithm tuning. For a good introduction, see chapter 9 in *Programming Pearls, 2nd ed* by Jon Bentley.