Design and Analysis of Algorithms

Comp 271

Department of Computer Science, HKUST
Information about the Lecturer

• Prof. Dekai WU

• Office: Rm 3539

• Email: dekai@cs.ust.hk

• http://www.cs.ust.hk/~dekai/271

• Office hours: Just drop by or send email for appointment.
Textbook and Lecture Notes


Lecture Slides: Available on course webpage

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.


5. String matching.


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-Oh 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: Polynomial Multiplication, Randomized quicksort, Randomized Selection and Deterministic Selection

• Graphs:
  – Depth-First Search - Applications of DFS (Articulation Points, and Biconnected Components)
  – Minimum Spanning Trees: Kruskal's and Prim's algorithms
  – Dijkstra's shortest path algorithm

• Dynamic Programming: 0-1 Knapsack, Chain Matrix Multiplication, Longest Common Subsequence, All Pairs Shortest Path

• Greedy algorithms: Fractional Knapsack, Huffman Coding

• Algorithm Examples: Fast Fourier Transformation (FFT) and String-Matching Algorithms

• Complexity Classes: The classes P and NP, NP-complete problems, polynomial reductions
Other Information

- Assignments: 4–5, worth a total of 20% of grade.
- Midterm: worth 35% of grade.
- Final exam (comprehensive): worth 45% of grade.
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:** \( A[1 \ldots n] \) is an array of numbers

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

**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.
Analytical Time Complexity Analysis

- We would like to compare efficiencies of different algorithms for the same problem, instead of different programs or implementations. This removes dependency on machines and programming skill.

- It becomes meaningless to measure absolute time since we do not have a particular machine in mind. Instead, we measure the number of steps. We call this the time complexity or running time and denote it by $T(n)$.

- We would like to estimate how $T(n)$ varies with the input size $n$. 
Big-Oh

If $A$ is a much better algorithm than $B$, then it is not necessary to calculate $T_A(n)$ and $T_B(n)$ exactly. As $n$ increases, since $T_B(n)$ will grow much more rapidly, $T_A(n)$ will always be less than $T_B(n)$ for large enough $n$.

Thus, it suffices to measure the *growth rate* of time complexity to get a rough comparison.

$$f(n) = O(g(n))$$:

There exists constant $c > 0$ and $n_0$ such that $f(n) \leq c \cdot g(n)$ for $n \geq n_0$. 
When estimating the growth rate of $T(n)$ using big-Oh:

- Ignore the low order terms.
- Ignore the constant coefficient of the most significant term.
- The remaining term is the estimate.
For example,

- \( n^2/2 - 3n = O(n^2) \)
- \( 1 + 4n = O(n) \)
- \( \log_{10} n = \frac{\log_2 n}{\log_2 10} = O(\log_2 n) = O(\log n) \)
- \( \sin n = O(1) \), \( 10 = O(1) \), \( 10^{10} = O(1) \).
- \( \sum_{i=1}^{n} i^2 \leq n \cdot n^2 = O(n^3) \)
- \( \sum_{i=1}^{n} i \leq n \cdot n = O(n^2) \)
- \( 2^{10n} \) is not \( O(2^n) \)
- \( 7n^2 + 10n + 3 = O(n^2) = O(n^3) = O(n^4) \)
**Big Omega and Big Theta**

\[ f(n) = \Omega(g(n)) \text{ (big-Omega):} \]

There exists constant \( c > 0 \) and \( n_0 \) such that 
\[ f(n) \geq c \cdot g(n) \text{ for } n \geq n_0. \]

\[ f(n) = \Theta(g(n)) \text{ (big-Theta):} \]

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

- *Algorithm Design*, as taught in this class, is mainly about designing algorithms that have small big-Oh 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 cut 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.