Algorithms and Theory of Computation

Lecture 1: Introduction, Basics of Algorithms

Xiaohui Bei

MAS 714

August 13, 2018
Administration

- **Lectures**
  - Monday 10:30am - 12:30pm  SPMS-TR+12
  - Tuesday 9:30am - 11:30am  SPMS-TR+12

- **Tutorials**
  - Tuesday 10:30am - 11:30am  biweekly

- Website: [http://www3.ntu.edu.sg/home/xhbei/MAS714.html](http://www3.ntu.edu.sg/home/xhbei/MAS714.html)

- E-mail: xhbei@ntu.edu.sg  Office: MAS-05-46
Gradings

Homework: 20%, Midterm 20%, Final: 60%

Homework

- Submitted by e-mail (as PDF files obtained from \LaTeX or Word sources), or by pushing a hard copy under my office door.
- Solutions will be discussed on tutorials.

Homework policy:

- You are allowed (and encouraged) to discuss with your peers on the questions and solutions. But everyone needs to write and submit their own solutions.
- No cheating behavior will be tolerated.

NTU academic integrity policy can be found at http://www.ntu.edu.sg/ai.
Recommended Reading

- Kleinberg, Tardos: *Algorithm Design*
- Cormen, Leiserson, Rivest, Stein: *Introduction to Algorithms*
- Sipser: *Introduction to the Theory of Computation*
Course Structure

1. Algorithm Design
   - graph algorithms
   - greedy
   - divide and conquer
   - dynamic programming
   - linear programming
   - network flow

2. Automata Theory
   - regular languages
   - finite state machines

3. Computability Theory
   - Turing machines
   - undecidability
   - P and NP
   - NP-completeness
An **algorithm** is a procedure for performing a computation. Start from an initial state and an **input** (perhaps empty), eventually produce an **output**.

An algorithm consists of **primitive steps/instructions** that can be executed mechanically.

What is a primitive step? Depends on the model of computation.
- C++ commands

A **computer** is a device that can be programmed to carry out primitive steps.
- can then implement an entire algorithm by keeping track of state
- model vs. device
Computers can be Humans!

Figure: Women at work tabulating during World War II (Shorpy)
How to measure the performance of an algorithm?

- **Correctness**: a must!
  - the definition of “correctness” can be discussed

- Two most important measures are **time** and **space**.

- **Time**: number of primitive steps needed to perform the computation.

- **Space**: the amount of storage needed during the computation.
Algorithm Example

Number Addition

Given two $n$-digit numbers $x$ and $y$, compute $x + y$.

Procedure

\[
\begin{array}{cccccc}
1 & 1 \\
3 & 8 & 7 & 3 & 4 \\
\hline
+ & 8 & 4 & 0 & 7 & 5 \\
\hline
1 & 2 & 2 & 8 & 0 & 9 \\
\end{array}
\]

Algorithm explained:

- write numbers under each other
- add number position by position moving a “carry” forward
Algorithm Analysis

Primitive steps:
- add two digits
- read and write

Time complexity:
- algorithm requires $O(n)$ primitive steps.

Space complexity:
- algorithm requires $O(n)$ storage space.
Model of Computation

An “idealized mathematical construct” that describes the primitive instructions and other details.

- Turing Machines
- Circuits
- Random Access Machine (RAM)
- etc.

Random Access Machine (RAM)

- read/write from registers
- arithmetic operation on registers
- indirect addressing

Within this course, basically, pseudocode/informal language.
Some Other Examples

Number Multiplication

Given two $n$-digit numbers $x$ and $y$, compute $x \cdot y$.

Procedure

\[
\begin{array}{c}
8 7 3 \\
\times 4 7 5 \\
\hline
4 3 6 5 \\
6 1 1 1 \\
3 4 9 2 \\
\hline
4 1 4 6 7 5 \\
\end{array}
\]
Number Multiplication

- Number of primitive steps: $O(n^2)$.
- Space: $O(n^2)$.
- Can we do better? Yes, but highly nontrivial.
  - best known algorithm time: $O(n \log n \cdot 2^{O(\log^* n)})$ [Furer 2008]
  - previous best time: $O(n \log n \log \log n)$ [Schonhage-Strassen 1971]
  - Conjecture: an $O(n \log n)$ time algorithm exists.
- We don’t even understand multiplication well.
### Independent Set

<table>
<thead>
<tr>
<th><strong>Input</strong></th>
<th>an undirected graph $G = (V, E)$.</th>
</tr>
</thead>
<tbody>
<tr>
<td>A set of nodes $S \subseteq V$ is <strong>independent</strong> if no two nodes in $S$ are jointed by an edge.</td>
<td></td>
</tr>
<tr>
<td><strong>Problem</strong></td>
<td>find an independent set that is as large as possible.</td>
</tr>
</tbody>
</table>

Harder: no efficient algorithm is known.

- conjecture: no such algorithm exists

On the good side: checking the validity of a solution is easy.
Hex (Board Game)

**Hex**

- **Rules**: Players take turn placing a stone (of their color) on an unoccupied cell.
- **Goal**: Form a path of their own stones connecting the opposing side of the board marked by their colors.

**Problem**: determine whether a position is a winning position.

Even harder: PSPACE-Complete

- even checking a solution is hard
Posts' Correspondence Problem

Input: two sequences of strings.
  - e.g. A = [a, ab, bba], B = [baa, aa, bb]

Problem: Find a sequence of indices such that the corresponding concatenated strings are the same.
  - e.g. (3, 2, 3, 1) \Rightarrow bba \ ab \ bba \ a = bb \ aa \ bb \ baa

Even harder: Undecidable
  - no algorithm exists!
Algorithm design is not a piece of cake.

Difficulty varies in different problems.
Basics of Algorithm Analysis
What is a good algorithm?

“No, Thursday’s out. How about never—is never good for you?”

Figure: All Rights Reserved http://www.cartoonbank.com
What qualifies as an efficient algorithm?

- Most important factor: running time.
- Many possibilities: $n$, $n \log n$, $n^2$, $n^3$, $n^{100}$, $2^n$, $n!$, etc...

Brute force: enumerate every possible solutions that check their validity.
- Usually takes at least $2^n$ time, is not considered as efficient.
### Running Time Examples

The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds $10^{25}$ years, we simply record the algorithm as taking a very long time.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n$</th>
<th>$n \log_2 n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$1.5^n$</th>
<th>$2^n$</th>
<th>$n!$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n = 10$</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$4$ sec</td>
</tr>
<tr>
<td>$n = 30$</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$18$ min</td>
<td>$10^{25}$ years</td>
</tr>
<tr>
<td>$n = 50$</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$11$ min</td>
<td>$36$ years</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 100$</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$1$ sec</td>
<td>$12,892$ years</td>
<td>$10^{17}$ years</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 1,000$</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$1$ sec</td>
<td>$18$ min</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 10,000$</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$2$ min</td>
<td>$12$ days</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 100,000$</td>
<td>$&lt; 1$ sec</td>
<td>$2$ sec</td>
<td>$3$ hours</td>
<td>$32$ years</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 1,000,000$</td>
<td>$1$ sec</td>
<td>$20$ sec</td>
<td>$12$ days</td>
<td>$31,710$ years</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
</tbody>
</table>
Polynomial Running Time

Efficient Algorithms

An algorithm called efficient if it has a polynomial running time.

What is so good about polynomial running time?

- Robust, mathematically sound.
- When input size doubles, the running time only increases by some constant factor $C$.
- Works well in practice.

Drawbacks:

- Polynomials with large constants/exponents are not that practical.
- $20n^{100}$ or $n^{1+0.02\ln n}$?
Type of Analyses

Running Time

How to define the running time of an algorithm?

- **Worst-case.** Running time guarantee for any input of size $n$.  
- **Average-case.** Expected running time for a random input of size $n$.  
- **Probabilistic.** Expected running time of a randomized algorithm.  
- **Amortized.** Worst-case running time for any sequence of $n$ operations.