Analysis of Algorithms I: Introduction

Xi Chen

Columbia University
Computational Problem: A well-defined input/output relationship. E.g., sorting, connected components, greatest common divisor (GCD), matrix multiplication.

Algorithm: A well-defined procedure that takes something (as input) and produces something (as output).

- Existed before computers: e.g., the Euclidean algorithm for GCD. [Section 31.2 of the textbook if interested]

An algorithm correctly solves a problem if, for every input instance, it halts with the correct output.
- Correctness: Provably correct in this course.
- Performance: (mostly) time complexity, and space complexity (or other computational resources).
- How to measure the running time of an algorithm?
  - the random-access machine (RAM) model
    [Section 2.2 of the textbook for more details]
  - cells storing integers and rational numbers
  - basic operations: arithmetic/data movement/control
  - count the number of basic operations
InsertionSort($A$), where $A = \langle a_1, \ldots, a_n \rangle$ is a sequence of integers:

1. Create an empty list $B$
2. For $i$ from 1 to $n$
   
   Enumerate the list $B$ backwards to find the first integer in $B$ smaller than $a_i$; insert $a_i$ right after that integer.
We use $T(A)$ to denote the number of basic operations it uses when the input is $A$, and we are interested in its worst-case time complexity: For $n \geq 1$, let

$$T(n) = \max_{\text{all } A \text{ of length } n} T(A).$$

Deriving exactly what $T(n)$ is can be very tedious, e.g., it depends on how we implement a list using a RAM.
In a certain implementation, assume that line 1 and line 2 take $c_1$ and $c_2$ steps each, where $c_1$ and $c_2$ are constants that are independent of the input size $n$. Also assume the $i$th iteration of the for-loop takes $c_3 k_i + c_4$ steps, where

- $c_3$: number of steps to enumerate backwards an integer in $B$;
- $c_4$: number of steps it takes for insertion;
- and $k_i$ is the number of integers we need to enumerate backwards to find an integer smaller than $a_i$.

Again, $c_3$ and $c_4$ are constants in a reasonable implementation.
From these assumptions, we have

\[ T(A) = c_1 + c_2 \cdot n + \sum_{i=1}^{n} (c_3 k_i + c_4) = c_1 + c_2 \cdot n + c_4 \cdot n + c_3 \sum_{i=1}^{n} k_i. \]

Different input instances yield different \( k_i \)'s. If \( A = \langle 1, 2, \ldots, n \rangle \) is already ordered nonincreasingly, then \( k_i = 1 \) for all \( i \). But when \( A' = \langle n, n-1, \ldots, 1 \rangle \), we have \( k_i = i \) for all \( i \). So

\[ T(A) = c_1 + c_2 \cdot n + c_4 \cdot n + c_3 \cdot n \]

\[ T(A') = c_1 + c_2 \cdot n + c_4 \cdot n + c_3 \cdot \sum_{i=1}^{n} i. \]

where \( \sum_{i=1}^{n} i = \frac{n(n+1)}{2} \). [Will use Induction to prove it next class]
We conclude that

\[ T(n) = T(A') = c_1 + c_2 \cdot n + c_4 \cdot n + c_3 \cdot \frac{n(n+1)}{2}, \]

because \( k_i \) can be no more than the length of the list \( B \), which is \( i \) in the \( i \)-th iteration of the for-loop.
Usually we make the following two simplifications in analysis:

- focus on the dominant term: keep $c_3 n^2 / 2$ only
- suppress the constant coefficient: keep $n^2$ only

More formally, we use the asymptotic notation: $T(n) = \Theta(n^2)$ (to be defined next).
Not worth the effort to keep the constant $c_3$ because

- An algorithm with $T(n) = 100n$ may not always perform better than an algorithm with $T(n) = n$ in practice, because the cost of the RAM basic operations vary among different machines.

- An algorithm with $T(n) = c_1 n$ always performs better than an algorithm with $T(n) = c_2 n^2$, when the input is large enough, no matter what the positive constants $c_1, c_2$ are.
We focus on the asymptotic performance to

- avoid the tedious analysis of the constants;
- understand the intrinsic (and machine-independent) complexity of an algorithm;
- concentrate on the dominant term when designing an algorithm because this decides its performance when the inputs are large.
But what if the hidden constant is really really large: E.g., for an algorithm with $T(n) = 10^{100}n$ to perform better than an algorithm with $T(n) = n^2$, $n$ needs to be $10^{100}$.

Fortunately the algorithms we cover in the course are well polished and have low hidden constants.
Let $f(n)$ and $g(n)$ are functions that map $n = 1, 2, \ldots$ to real numbers, then we let

$$O(g(n)) = \left\{ f(n) : \exists \text{ constants } c > 0 \text{ and } n_0 > 0 \right.$$ s.t. $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0 \right\}$

Check Figure 3.1 (b) of the textbook. Usually we use

$$f(n) = O(g(n)) \quad \text{to denote} \quad f(n) \in O(g(n))$$
Let $f(n)$ and $g(n)$ are functions that map $n = 1, 2, \ldots$ to real numbers, then we let

$$\Omega(g(n)) = \left\{ f(n) : \exists \text{ constants } c > 0 \text{ and } n_0 > 0 \right. \left. \text{ s.t. } 0 \leq g(n) \leq c \cdot f(n) \text{ for all } n \geq n_0 \right\}$$

Check Figure 3.1 (c) of the textbook. Usually we use

$$f(n) = \Omega(g(n)) \quad \text{to denote } \quad f(n) \in \Omega(g(n)).$$
Let \( f(n) \) and \( g(n) \) are functions that map \( n = 1, 2, \ldots \) to real numbers, then we let

\[
\Theta(g(n)) = \left\{ f(n) : \exists \text{ constants } c_1, c_2 > 0 \text{ and } n_0 > 0 \text{ s.t. } 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \text{ for all } n \geq n_0 \right\}
\]

Check Figure 3.1 (a) of the textbook. Usually we use

\[
f(n) = \Theta(g(n)) \quad \text{to denote} \quad f(n) \in \Theta(g(n)).
\]
Read Section 3.1 of the textbook to get comfortable about the asymptotic notation. Will be used in almost every lecture.

Back to the InsertionSort, we have $T(n) = O(n^2)$. To formally prove this, use limit from calculus:

$$\lim_{{n \to \infty}} \frac{T(n)}{n^2} = \frac{c_3}{2}$$

Let $\epsilon > 0$ be any constant. By the definition of limits, there exists a large enough $n_0$ such that

$$\frac{T(n)}{n^2} < \frac{c_3}{2} + \epsilon, \quad \text{for all } n \geq n_0.$$
Similarly $T(n) = \Omega(n^2)$ and thus, by Theorem 3.1 (Page 48, also an exercise in the first homework), $T(n) = \Theta(n^2)$. This finishes the asymptotic worst-case analysis of InsertionSort.