Lecture 1

6.046/18.410J
Introduction to Algorithms
Profs. Erik Demaine & Madhu Sudan
http://theory.csail.mit.edu/classes/6.046/

Today: Introduction
- Course information
- What & why of algorithms
- Insertion sort
- Merge sort
- Asymptotic analysis
- Worst case vs. average case
- Recurrences

Abu Abdullah Muhammad ibn Musa al-Khwarizmi (~780–850 A.D.)
Persian mathematician - father of algebra
linear & quadratic equations; arithmetic
“al-kha-raz-mi”
Algorithm Design & Analysis:
- theoretical study of how to solve computational problems
  - sorting a list of numbers
  - finding shortest route from A to B in a map
  - scheduling when to work on your homework [tasks with deadlines]
  - ...

- solution = provably correct algorithm
- algorithm = mathematical abstraction of computer program

- tools for:
  - designing algorithms
  - analyzing & comparing algorithms' performance & resource usage
Typical goal: what's the fastest (correct) algorithm for a given problem?
- scalability of solution
- feasibility of problem
- "speed is the currency of computing"

- speed does not capture other important properties of computer programs:
  - simplicity
  - programmer time
  - maintainability
  - extensibility
  - modularity
  - reliability
  - robustness
  - user friendliness
  - etc.

- but speed is fun!
**Problem:** Sorting

**Input:** sequence \( a_1, a_2, \ldots, a_n \) of numbers

**Output:** permutation \( a'_1, a'_2, \ldots, a'_n \) such that \( a'_1 \leq a'_2 \leq \cdots \leq a'_n \).

*Example:* \( 8 \ 2 \ 4 \ 9 \ 3 \ 6 \rightarrow 2 \ 3 \ 4 \ 6 \ 8 \ 9 \)

---

**Insertion-Sort** (A, n)

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

---

**Invariant:** sorted \( A[1 \ldots j-1] \)
How to measure running time?
- depends on input (size of input; already sorted vs. reverse sorted)
- parameterize by size n of input
  1. **worst case:** (usually)
     \[ T(n) = \text{max. time of algorithm on any input of size n} \]
     - guarantee on running time
  2. **average case:** (sometimes)
     \[ T(n) = \text{expected time of algorithm over all inputs of size n} \]
     - requires assumption about distribution of inputs
  3. **best case:** (bogus!)
     - cheat with slow algorithm that's fast on some input
Machine independence:
- $T(n)$ defined in terms of "time of algorithm on input"
- but this depends on machine (speed, instruction set)
  & exact implementation (as instructions)

Big Idea: Asymptotic Analysis
- look at growth of $T(n)$ as $n \to \infty$
- ignore machine-dependent constants

$\Theta$ notation:
$\Theta(g(n)) = \{ f(n) : \text{there exist consts. } c_1, c_2, n_0$
  \text{ such that, for all } n \geq n_0, \quad
  0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \}$

"$f(n) = \Theta(g(n))$" means $f(n) \in \Theta(g(n))$
  asymmetric!

Roughly: drop lower-order terms
  ignore leading constants
  e.g. $3n^3 + 90n^2 - 5n + 6046 = \Theta(n^3)$
Asymptotic analysis:
- when $n$ gets large enough,
  $\Theta(n^2)$ algorithm always beats $\Theta(n^3)$ alg.
  (more scalable)

- but constants & lower-order terms play a role in practice
- nonetheless a powerful tool for analysis & comparison
Insertion sort analysis:
- worst case: (reverse-sorted input)
  \[ T(n) = \sum_{j=2}^{n} \Theta(j) = \Theta(n^2) \] — arithmetic series
- average case, all permutations equally likely:
  \[ T(n) = \sum_{j=2}^{n} \Theta(j/2) = \Theta(n^2) \]
- OK for small \( n \)
- sloooow for large \( n \)
**Merge sort** \(A[1 \cdots n]\):

1. if \(n=1\) then done
2. recursively sort \(A[1 \cdots \lfloor n/2 \rfloor]\) & \(A[\lfloor n/2 \rfloor +1 \cdots n]\)
3. merge the two sorted lists

**Merge subroutine:**

\[
\begin{array}{c|c|c|c|c|c|c|c}
20 & 12 & 20 & 12 & 20 & 12 & 20 & 12 \\
13 & 11 & 13 & 11 & 13 & 11 & 13 & 11 \\
7 & 9 & 7 & 9 & 7 & 9 & 9 & 9 \\
2 & 1 & 2 & 2 & 7 & 9 & 11 & 12 \\
\end{array}
\]

\(\Theta(n)\) time to merge \(n\) elements

"linear time"

**Merge-sort analysis:**

1. \(- \Theta(1)\) ← abuse of notation (need "w.r.t. \(n\)"
2. \(- 2 T(n/2)\) ← sloppiness — really \(T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil)\)
3. \(- \Theta(n)\)

\[T(n) = 2T(\lfloor n/2 \rfloor) + \Theta(n) \quad \text{if } n>1\]

\[T(1) = \Theta(1)\]

— generally assume \(T(n) = \Theta(1)\) for \(n \leq n_0\)
Lecture 2 covers solving recurrences. Here’s one method:

**Recursion tree:** \( T(n) = 2T(n/2) + cn \)

\[
T(n) = \begin{array}{c}
\text{cn} \\
T(n/2)\quad T(n/2) \\
\text{cn/2} \quad \text{cn/2} \\
T(n/4) \quad T(n/4) \quad T(n/4) \quad T(n/4)
\end{array}
\]

\[= \cdots =
\begin{array}{c}
\text{cn} \\
\text{cn/2} \quad \text{cn/2} \\
\text{cn/4} \quad \text{cn/4} \quad \text{cn/4} \quad \text{cn/4} \\
\end{array}
\]

\[
= \cdots \quad n \text{ leaves} \quad \Theta(n)
\]

**Total:** \( T(n) = cn \lg n + \Theta(n) \)

\[= \Theta(n \lg n)\]

- grows slower than \( \Theta(n^2) \)
- \( \Rightarrow \) mergesort faster than insertion sort for large enough \( n \)
- in practice, \( n > 30 \) or so
- (120 in my trivial Python test)