Algorithm Analysis 101

Data Structures and Algorithms

Nanjing University, Fall 2021
郑朝栋

Insertion Sort

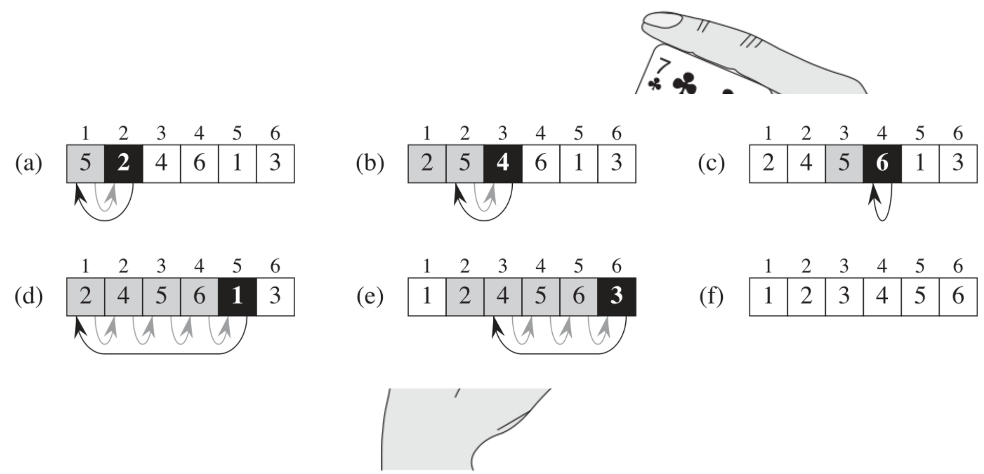
Integer Sorting Problem:

Input: a sequence of n integers $< a_1, a_2, \cdots, a_n >$
Output: a reordering $< a'_1, a'_2, \cdots, a'_n >$ of input where $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

Algorithm design strategy 0: wisdom from daily l

Insertion-Sort (A)

```
for j = 2 to A.length
   key = A[j]
   i = j - 1
   while (i>0 and A[i]>key)
      A[i+1] = A[i]
      i = i - 1
   A[i+1] = key
```



Assume we have $j-1$ sorted cards, then put the j^{th} card into its correct position.

Correctness of Insertion Sort

Insertion-Sort (A)

```
for j = 2 to A.length
   key = A[j]
   i = j – 1
   while (i>0 and A[i]>key)
      A[i+1] = A[i]
      i = i – 1
   A[i+1] = key
```

An algorithm is **correct** if for **every** input instance of the considered problem, the algorithm **halts** with the **correct** output.

The algorithm terminates within finite steps on every instance. **(WHY?**

The algorithm outputs correct result on every instance. **(WHY?!)**

Correctness of Insertion Sort

Insertion-Sort (A)

```
for j = 1 to n
   key = A[j]
   i = j – 1
   while (i>0 and A[i]>key)
      A[i+1] = A[i]
      i = i – 1
   A[i+1] = key
```

The algorithm outputs correct result on every instance.

Claim: By the end of the j^{th} iteration, the elements in subarray $A[1, \cdots, j]$ are in sorted order.

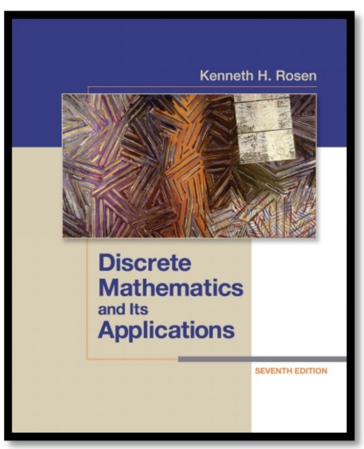
Often called a "**loop invariant**", which gives helpful properties when loop exits.

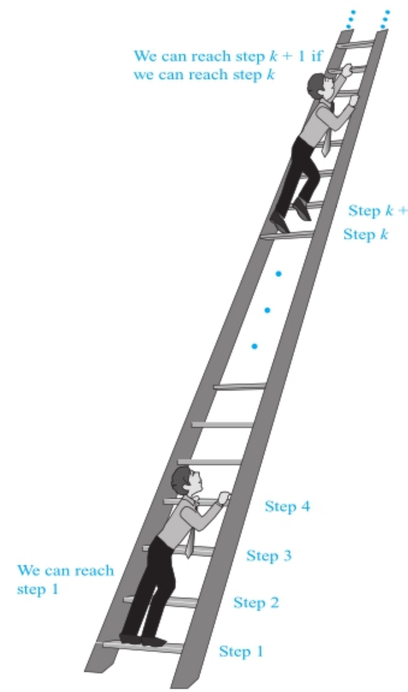
Proof of the above claim via **mathematical induction**:

- **[Basis]** By the end of the first iteration, $A[1]$ is in sorted order.
- **[Inductive Step]** Assume by the end of the j^{th} iteration, the elements in subarray $A[1, \cdots, j]$ are in sorted order; then by the end of the $(j+1)^{\text{th}}$ iteration, the elements in subarray $A[1, \cdots, j+1]$ are in sorted order.

Proving the correctness of algorithms

- Some methods and strategies: proof by cases, proof by contraposition, proof by contradiction, etc.
- When loops and/or recursions are involved: often (if not always) use mathematical induction.
- Review your discrete math book if you f
  unfamiliar with above terms…
  [Rosen] Ch.1 (1.7, 1.8) and Ch.5 (5.1, 5.2)





Efficiency of Insertion Sort

Insertion-Sort (A)

```
for j = 2 to A.length
   key = A[j]
   i = j – 1
   while (i>0 and A[i]>key)
      A[i+1] = A[i]
      i = i – 1
   A[i+1] = key
```

Time complexity: how much time is needed before halting.

Space complexity: how much memory (usually excluding input) is required for successful execution.

Other performance measures…

Observation: larger inputs often demands more time.
Cost of an algorithm should be a function of *input size*.

Observation: same (high-level) algorithm on same input can have different running times on different machines.
Cost of an algorithm should be measured on a specific *model of computation*.

Running time in the RAM model

Random-Access-Machine (**RAM**):
relatively simple, yet generic and representative.

- One processor which executes instructions one by one.
- Memory cells supporting random access, each of limited size.
- RAM model supports common instructions.
    - Arithmetic, logic, data movement, control, …
- RAM model supports common data types.
    - Integers, floating point numbers, …
- RAM model does *not* support complex instructions or data types (directly).
    - Vector operations, graphs, …

Given an algorithm and an input, running time in the RAM model:
Number of instructions executed before the algorithm halts.

Time complexity of Insertion Sort

Insertion-Sort (A)	cost	# of times
for j = 2 to A.length	c_1	n
key = A[j]	c_2	$n-1$
i = j – 1	c_3	$n-1$
while (i>0 and A[i]>key)	c_4	$\sum_{j=2}^{n} t_j$
A[i+1] = A[i]	c_5	$\sum_{j=2}^{n} (t_j - 1)$
i = i – 1	c_6	$\sum_{j=2}^{n} (t_j - 1)$
A[i+1] = key	c_7	$n-1$

Assume $A.length = n$, then total running time $T(n)$ is:

$$c \cdot n + c' \cdot \left(\sum_{j=2}^{n} t_j\right) - c''$$

If $t_j = 1$, then $T(n) \approx cn + c'n - c''$ **Best case:** A is sorted

If $t_j = j$, then $T(n) \approx cn + (c'/2)n^2 - c''$ **Worst case:** A is reversely so

Average case???

Asymptotic Time Complexity

Insertion-Sort (A)

```
for j = 2 to A.length
   key = A[j]
   i = j – 1
   while (i>0 and A[i]>key)
      A[i+1] = A[i]
      i = i – 1
   A[i+1] = key
```

$T(n) = \mathrm{O}(n^2)$

Runtime $T(n) = c \cdot n + c' \cdot \left(\sum_{j=2}^{n} t_j\right) - c''$

Best case: $T(n) = \Theta(n)$

$t_j = 1$ and $T(n) \approx cn + c'n - c''$

Worst case: $T(n) = \Theta(n^2)$

$t_j = j$ and $T(n) \approx cn + (c'/2)n^2 - c''$

Suppose there is another sorting algorithm with runtime $T(n) = d \cdot n \cdot \lg$

Which algorithm is better? $T(n) = \Theta(n \lg n)$

Constant coefficients are not that important (when n is large)

Lower-order terms are not that important (when n is large).

Asymptotic Notation: O

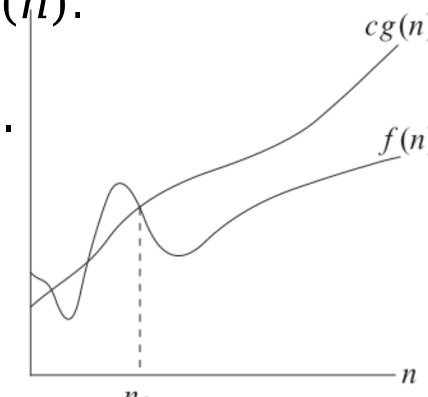
Use **asymptotic notations** to describe asymptotic efficiency of algori
(Ignore constant coefficients and lower-order terms.)

Given a function $g(n)$, we denote by $O(g(n))$ the following **set of functions**:
$\{f(n): \text{exists } c > 0 \text{ and } n_0 > 0, \;\; \text{such that } f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0\}$.

$f(n) = O(g(n))$: ~~$f(n)$ is **asymptotically smaller**~~ than $g(n)$.

$f(n) \in O(g(n))$: $f(n)$ is **asymptotically at most** $g(n)$.

O-notation gives an **asymptotic upper bound**.



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

Insertion Sort as an example:

- Best case: $T(n) \approx cn + c'n - c''$ $T(n) = O(n)$
- Worst case: $T(n) \approx cn + (c'/2)n^2 - c''$ $T(n) = O(n^2)$

Q: Is $n^3 = O(n^2)$? How to prove your answer?

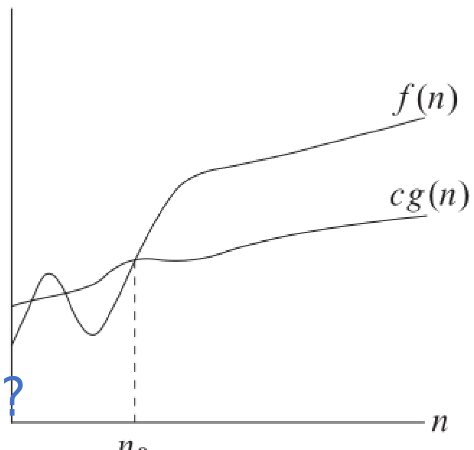
Asymptotic Notation: Ω

Given a function $g(n)$, we denote by $\Omega(g(n))$ the following **set of functions**:
$\{f(n)\text{: exists } c > 0 \text{ and } n_0 > 0, \text{ such that } f(n) \geq c \cdot g(n) \text{ for all } n \geq n_0\}$.

$f(n) = \Omega(g(n))$: ~~$f(n)$ is **asymptotically larger**~~ than $g(n)$.

$f(n) \in \Omega(g(n))$: $f(n)$ is **asymptotically at least** $g(n)$.

Ω-notation gives an **asymptotic lower bou**



$f(n) = \Omega(g(n))$

Insertion Sort as an example: correct but not helpful

- Best case: $T(n) \approx cn + c'n - c''$ $\cancel{T(n) = \Omega(n)}$
- Worst case: $T(n) \approx cn + (c'/2)n^2 - c''$ $\cancel{T(n) = \Omega(n^2)}$

Q: The time complexity of Insertion Sort is $\Omega(n^2)$?

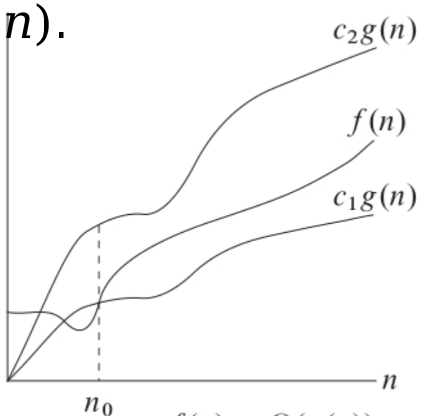
Asymptotic Notation: Θ

Given a function $g(n)$, we denote by $\Theta(g(n))$ the following **set of functions**:
$\{f(n)\text{: exists } c_1 > 0,\ c_2 > 0, \text{ and } n_0 > 0,$
$\text{such that } c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n) \text{ for all } n \ge n_0\}.$

$f(n) = \Theta(g(n))$: $f(n)$ is **asymptotically equal** to $g(n)$.

Θ-notation gives an **asymptotic tight bound**.

Given two functions $f(n)$ and $g(n)$, $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.



$f(n) = \Theta(g(n))$

Insertion Sort as an example:

- Best case: $T(n) \approx cn + c'n - c''$
- Worst case: $T(n) \approx cn + (c'/2)n^2 - c''$

Q: The time complexity of Insertion Sort is $\Theta(n^2)$?
The worst-case time complexity of Insertion Sort is $\Theta(n^2)$?

Asymptotic Notation: o

Given a function $g(n)$, we denote by $O(g(n))$ the following set of functions:
$\{f(n): \text{exists } c > 0 \text{ and } n_0 > 0, \text{ such that } f(n) \le c \cdot g(n) \text{ when } n \ge n_0\}$.

$f(n) \in O(g(n))$: $f(n)$ is **asymptotically at most** $g(n)$, or "$f(n) \le g(n)$" asy

How to define: $f(n)$ is **asymptotically (strictly) smaller than** $g(n)$

Given a function $g(n)$, we denote by $o(g(n))$ the following set of functions:
$\{f(n): \text{for } \mathbf{any}\ c > 0, \text{ exists } n_0 > 0, \text{ such that } f(n) < c \cdot g(n) \text{ when } n \ge n_0\}$.

Alternatively, we say $f(n) \in o(g(n))$ iff $\lim_{n\to\infty} f(n)/g(n) = 0$.

Observe that o is the "negation" of Ω!

Q: Is $n^2 \in o(n^2)$ true? Is $n \lg n \in o(n^2)$ true?

Asymptotic Notation: ω

Given a function $g(n)$, we denote by $\Omega(g(n))$ the following set of functions:
$\{f(n): \text{exists } c > 0 \text{ and } n_0 > 0, \text{ such that } f(n) \geq c \cdot g(n) \text{ for all } n \geq n_0\}$.

$f(n) \in \Omega(g(n))$: $f(n)$ is **asymptotically at least** $g(n)$, or "$f(n) \geq g(n)$" asy

How to define: $f(n)$ is **asymptotically (strictly) larger than** $g(n)$?

Given a function $g(n)$, we denote by $\omega(g(n))$ the following set of functions:
$\{f(n): \text{for } \mathbf{any}\ c > 0, \text{ exists } n_0 > 0, \text{ such that } f(n) > c \cdot g(n) \text{ when } n \geq n_0\}$.

Alternatively, we say $f(n) \in \omega(g(n))$ iff $\lim_{n\to\infty} f(n)/g(n) = \infty$.

Observe that ω is the "negation" of O!

Q: Now that we have $O,\ \Omega,\ \Theta$ and o, ω, do we have small θ?

Some properties of asymptotic notations

- Reflexivity
  - E.g., $f(n) \in O(f(n))$; but $f(n) \notin o(f(n))$.
- Transitivity
  - E.g., if $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$, then $f(n) \in O(h(n))$.
- Symmetry
  - $f(n) \in \Theta(g(n))$ iff $g(n) \in \Theta(f(n))$.
- Transpose symmetry
  - E.g., $f(n) \in O(g(n))$ iff $g(n) \in \Omega(f(n))$.

Comparing some common functions

- $\Theta(1)$, constant
- $\Theta(\lg n)$, logarithm
- $\Theta(n)$, linear
- $\Theta(n^c)$, polynomial
- $\Theta(2^n)$, exponential

<u>Handy tools:</u>

- **L'Hôpital's rule** for comparison of two functions.
- **Stirling’s approximation** to deal with factorials.

Reading

- [CLRS] Ch.2 (2.1, 2.2), Ch.3
- [Rosen] Ch.1 (1.7, 1.8) and Ch.5 (5.1, 5.2)