CS 375 – Analysis of Algorithms

Professor Eric Aaron

Lecture – M W 1:00pm

Lecture Meeting Location: It’s complicated…

Business, Mar. 10

• Reading: Ch. 2 and Ch. 3
  – Although not all of it will show up in CS375 exercises, it is all good material to know

• See Divide & Conquer II slides for other Mar. 10 course business
Business, Mar. 15

- PS_DivConqII due already
  - I expect to go over part of it in class on Wednesday—please recall the course lateness policy and submit it before then!

- Intro to Asymptotic Complexity Problem Set out today, due date Mar. 24 (see assignment sheet)
  - It is likely there will be an additional Small Assignment before this PS is due

- Reading: Ch. 2 and Ch. 3
  - Although not all of it will show up in CS375 exercises, it is all good material to know
Business, Mar. 17

- Intro to Asymptotic Complexity Problem Set out
  - Due date Mar. 24 (see assignment sheet)

- Small Assignment assigned today
  - Due by beginning of class March 22
  - See CS375 Lecture Notes page for the SA3 assignment sheet
    - (Look under “Slides, Readings, etc.” for this course module)

- On both of the above assignments:
  - Please show all work regarding asymptotic complexity—do not (yet) use high-level approximations or shortcuts

- Reading: Ch. 2 and Ch. 3
  - Although not all of it will show up in CS375 exercises, it is all good material to know
Business, Mar. 22

• Grading update on PS_DivConqII
• Small Assignment 3 due already; discussed in class today

• Intro to Asymptotic Complexity Problem Set out
  – Due date EXTENDED: by beginning of class on March 29
• On the above assignment:
  – Please show all work regarding asymptotic complexity—do not (yet) use high-level approximations or shortcuts

• Reading: Ch. 2 and Ch. 3
  – Although not all of it will show up in CS375 exercises, it is all good material to know
Business, Mar. 24

• Grading update on Small Assignment 3
• PS DivConqII returned already

• Intro to Asymptotic Complexity Problem Set out
  – Due date EXTENDED: by beginning of class on March 29
• On the above assignment:
  – Please show all work regarding asymptotic complexity—do not (yet) use high-level approximations or shortcuts
• Reading: Ch. 2 and Ch. 3
  – Although not all of it will show up in CS375 exercises, it is all good material to know
And now, for something almost completely different

This slide is almost completely about Python
The Sorting Problem;
Insertion Sort

• There are many algorithms that solve the sorting problem

<table>
<thead>
<tr>
<th>Input: A sequence L of n numbers &lt;a₁, ..., aₙ&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: A permutation (reordering) &lt;b₁, ..., bₙ&gt; of the input sequence (perhaps leaving them unchanged) such that b₁ ≤ b₂ ≤ ... ≤ bₙ</td>
</tr>
</tbody>
</table>

  – What algorithms are you familiar with?
  – Sorting algorithms can be expressed *iteratively* or *recursively*

What’s the fastest sorting algorithm?
... and how would we even know?
... and what does “fastest” mean in context, anyway?

The Sorting Problem;
Insertion Sort

• There are many algorithms that solve the sorting problem

<table>
<thead>
<tr>
<th>Input: A sequence L of n numbers &lt;a₁, ..., aₙ&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: A permutation (reordering) &lt;b₁, ..., bₙ&gt; of the input sequence (perhaps leaving them unchanged) such that b₁ ≤ b₂ ≤ ... ≤ bₙ</td>
</tr>
</tbody>
</table>

  – What algorithms are you familiar with?
  – Sorting algorithms can be expressed *iteratively* or *recursively*

• One classic, efficient algorithm: *Insertion sort*
  – Based on its name, any ideas about how it might work on an array?

  How could we write this algorithm iteratively (in pseudocode)?
Insertion Sort: An Example

- Example of insertion sort working on an array (taken from CLRS, section 2.1)
  - Any questions on what the *insert* part of the algorithm does, and how it’s used to sort an array?

---

(a)  1 2 3 4 5 6
     5 2 4 6 1 3
       (b)  1 2 3 4 5 6
             2 5 4 6 1 3
             (c)  1 2 3 4 5 6
                    2 4 5 6 1 3

(d)  1 2 3 4 5 6
     2 4 5 6 1 3
       (e)  1 2 3 4 5 6
             1 2 4 5 6 3
             (f)  1 2 3 4 5 6
                    1 2 3 4 5 6
Insertion Sort

- An iterative insertion sort! Is it efficient?
  - Also, is it correct? Does this meet the specifications for the sorting problem?
  - (More on that later in the course, when we talk about loop invariants!)

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

Input: A sequence \( L \) of \( n \) numbers \( <a_1, ..., a_n> \)
Output: A permutation (reordering) \( <b_1, ..., b_n> \) of the input sequence (perhaps leaving them unchanged) such that \( b_1 \leq b_2 \leq ... \leq b_n \)

In the algo:
- \( j \) is the element to be inserted in order
- \( i \) ranges over elements of the previously sorted sub-array

Introduction to Time Complexity
Analysis of Algorithms

- Could use a timer or stopwatch (or clock... or calendar...) to measure how fast an algorithm is on a given size of input...
  - (called empirical analysis...)
  - But that doesn’t really measure the algorithm speed
  - How much clock time passes is dependent on things other than just the algorithm (processor speed, memory access speed, etc.)!
- Could count how many operations an algorithm does on a given size of input as a measure of how long it takes!
  - Assume some unit of time for each operation.
  - This gives a measure of time usage (i.e., speed) that is dependent upon the algorithm as coded, not external factors!
- (How would that counting work for the iterative insertion sort on an array of size \( n \)?)
Insertion Sort In Pseudocode:
Counting Operations

• What operations should we count?
  – Could count all of them! Good, thorough analysis!
• And how do we count them?

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

... but a lot of work!
Is it necessary?

In the algo:
• j is the element to be inserted in order
• i ranges over elements of the previously sorted sub-array

Introduction to Time Complexity
Analysis of Algorithms, cont.

• But even that kind of counting depends on how an algorithm is implemented
  – If the insertion sort idea is implemented with even minor differences…
  – Operation count could change… but the algorithm is the essentially the same, independent of minor coding details!
  – We don’t want to say the algorithm has different speeds just because of many slightly different implementations.
• We want to discuss algorithm time complexity at a level a little bit more abstract than just a literal count of operations
  – If somehow we could capture the essential character of how many operations insertion sort takes …
    • on input of a given size (e.g., an array of size n)
  – …without getting caught up in small details…
Insertion Sort In Pseudocode:
Counting Barometer Operations

- What operations should we count?
  - Could count all operations done—good, thorough analysis!
  - Or, to be accurate to within a constant factor, could find some fundamental barometer operation and count only those
  - To be a good barometer operation, the algorithm has to do at most a constant amount of other work for each time the barometer operation occurs
- And how do we count them?

<table>
<thead>
<tr>
<th>What would a good barometer operation be?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exactly how many of those operations are executed on input of size $n$?</td>
</tr>
</tbody>
</table>

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

Asymptotic Analysis / Big-O Notation

- With insertion sort, if we gloss over minor details, we can see the number of operations is on the order of $n^2$
  - i.e., it is $c*n^2 + (\text{lower order terms})$
  - … for some constant $c$
  - … where $n$ is the size of the input
- Definition: An algorithm runs in time $O(f(n))$ (read: “order of $f(n)$”) means:
  - There exist $c > 0, n_0 > 0$ s.t. …
  - … for all $n \geq n_0$, the running time of the algorithm is less than $c*f(n)$
  - (Basically, that means that for every input “big enough,” the running time is less than a constant times $f(n)$)
- This running time measure captures some essential characteristic of an algorithm
  - $O(n^2)$ algorithms differ from $O(n^3)$, from $O(n \log n)$, etc.
Asymptotic Complexity & Big-O Notation

• What does asymptotic complexity refer to? Why do we focus on it when studying algorithms?

  - Big-O notation: \textit{asymptotic upper bound} on functions
    - Definition: \( O(g(n)) = \{ f(n) | \exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0, 0 \leq f(n) \leq c \cdot g(n) \}\)
    - We say \( f(n) = O(g(n)) \) to indicate \( f(n) \) is in \( O(g(n)) \)

• We can then apply this to functions representing algorithm running times
• An algorithm runs in time \( O(g(n)) \) (read: “order of \( g(n) \)” ) if
  - There exist \( c > 0, n_0 > 0 \text{ s.t. } \ldots \)
  - \( \ldots \text{for all } n \geq n_0, \text{ the running time of the algorithm is less than } c \cdot g(n) \)
  - (Basically, that means that for every input “big enough,” the running time is less than a constant times \( g(n) \))
Using the Big-O Definition

- Definition: \( O(g(n)) = \{ f(n) \mid \exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0, 0 \leq f(n) \leq c \cdot g(n) \} \)

- Is each of the below statements true? If so, give an argument that it’s true; if not, explain why not.

1. \( 100n + 5 = O(n^2) \)
2. \( n^2/2 - 3n = O(n^2) \)
3. \( 100n^2 = O(n^2) \)
4. \( 100n^2 = O(n^3) \)
5. \( 0.01n^3 = O(n^2) \)
6. \( n \log n = O(\log^2 n) \)
7. \( 2^{n+1} = O(2^n) \)
8. \( 2^{2n} = O(2^n) \)

In general, when explaining why an existential (“\( \exists \)” ) statement is true, explicitly give some witness value(s) that make it true as part of the explanation.

Here, for example, if a statement is true, give specific values for \( c, n_0 \) that make it true.
Asymptotic Exercises

• In what Big-O class is the following?

We use the shorthand that an algorithm is in f(n) + O(g(n)) if the running time is f(n) + t(n) for a function t(n) in O(g(n)). Similarly for O(f(n)) * O(g(n)), or other arithmetic combinations.

– (n lg n + O(n) + O(1)) * O(n^2)

• Claim: f(n) = O(g(n)) implies 2^{f(n)} = O(2^{g(n)})

– Is that claim true or false? Explain what makes it true, or give a counterexample (a specific example that demonstrates it’s not always true).

Common complexity measures and how they relate to input sizes

• Algorithms are sometimes described by their time complexity. There are
  – Logarithmic algorithms
  – Quadratic algorithms
  – Exponential algorithms
  – Factorial algorithms
  – etc.

• To see which kind is fastest, see how these functions grow with increases in the input size:

<table>
<thead>
<tr>
<th>n</th>
<th>log_{10} n</th>
<th>n^2</th>
<th>2^n</th>
<th>n!</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1.00</td>
<td>100</td>
<td>1024</td>
<td>3628800</td>
</tr>
<tr>
<td>50</td>
<td>1.70</td>
<td>2500</td>
<td>1.13e15</td>
<td>3.04e64</td>
</tr>
<tr>
<td>100</td>
<td>2</td>
<td>10000</td>
<td>1.27e30</td>
<td>9.44e157</td>
</tr>
</tbody>
</table>

Please look through CLRS Ch. 3.2 for information about these functions, including pg. 56 for notation for logarithms.
Conventional Wisdom about Big-O Classes

- If two algorithms are in different big-O classes, then there seems to be something substantially different about their speeds
  - Even though, for some small values of $n$, an $O(2^n)$ algorithm could be faster than an $O(n^2)$ algorithm…
  - It is nonetheless true that $2^n$ grows faster than $n^2$…
  - Thus, an $O(2^n)$ algorithm is, in a relevant sense, inherently slower than an $O(n^2)$ algorithm

- For an $O(n)$ algorithm (called “linear”)
  - Doubling the input size does what to the running time?
  - Increasing input size by factor of 100 does what to running time?

- For an $O(n^2)$ algorithm (“quadratic”)
  - Doubling the input size does what to the running time?
  - Increasing input size by factor of 100 does what to running time?

- For an $O(2^n)$ algorithm (“exponential”)
  - Doubling the input size does what to the running time?

Big “Oh… there’s more?” Notation

- Theta notation: Asymptotically tight bound
  - Definition: $\Theta(g(n)) = (f(n)) \\exists c_1, c_2, n_0 > 0$ s.t. $\forall n \geq n_0$
    $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$

- Big-Omega notation: Asymptotic lower bound
  - Definition: $\Omega(g(n)) = (f(n)) \\exists c, n_0 > 0$ s.t. $\forall n \geq n_0$
    $0 \leq c g(n) \leq f(n)$

- What is the relationship among big-O, big-Omega, and Theta classes?
A Big-Symbols Theorem

• **Definition:** \( \theta(g(n)) = \{ f(n) \mid \exists c1, c2, n_0 > 0 \text{ s.t. } \forall n \geq n_0, 0 \leq c1 \cdot g(n) \leq f(n) \leq c2 \cdot g(n) \} \)

• **Definition:** \( \Omega(g(n)) = \{ f(n) \mid \exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0, 0 \leq c \cdot g(n) \leq f(n) \} \)

• **Theorem:** For any two functions \( f(n) \) and \( g(n) \), \( f(n) = \theta(g(n)) \) iff \( f(n) = O(g(n)) \) and \( f(n) = \Omega(g(n)) \).

Using the \( \theta, \Omega \) Definitions

• **Definition:** \( \theta(g(n)) = \{ f(n) \mid \exists c1, c2, n_0 > 0 \text{ s.t. } \forall n \geq n_0, 0 \leq c1 \cdot g(n) \leq f(n) \leq c2 \cdot g(n) \} \)

• **Definition:** \( \Omega(g(n)) = \{ f(n) \mid \exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0, 0 \leq c \cdot g(n) \leq f(n) \} \)

• Is each of the below statements true?

  1. \( 100n + 5 = \theta(n^2) \)
  2. \( 100n + 5 = \Omega(n^2) \)
  3. \( \frac{n^2}{2} - 3n = \theta(n^2) \)
  4. \( \frac{n^2}{2} - 3n = \Omega(n^2) \)
  5. \( 100n^2 = \theta(n^3) \)
  6. \( 0.01n^3 = \Omega(n^2) \)
  7. \( 2^{n+1} = \theta(2^n) \)
  8. \( 2^{2n} = \Omega(2^n) \)
<table>
<thead>
<tr>
<th>Design Paradigm</th>
<th>Analysis</th>
<th>Correctness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterative</td>
<td>Counting (Exact count of operations / space used)</td>
<td>Loop invariants</td>
</tr>
<tr>
<td>Recursive</td>
<td>Solving recurrences</td>
<td>Induction</td>
</tr>
</tbody>
</table>