CS 375 – Analysis of Algorithms

Professor Eric Aaron

Lecture – M W 1:00pm

Lecture Meeting Location: Davis 117

Business

• HW6 in already
• HW7 out soon, due Wednesday, March 4
• HW4, 5 back today
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;a_1, ..., a_n&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: A permutation (reordering) (&lt;b_1, ..., b_n&gt;) of the input sequence (perhaps leaving them unchanged) such that (b_1 \leq b_2 \leq ... \leq b_n)</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?
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 ≤ b_2 ≤ ... ≤ 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?

```plaintext
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 subarray

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?

What would a good barometer operation be?

Exactly how many of those
operations are executed on
input of size \( n \)?

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 \( c \cdot n^2 + \) (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 \( c \cdot f(n) \)
  – (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: asymptotic upper bound on functions
  - Definition: \( O(g(n)) = \{ f(n) \mid \exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0, 0 \leq f(n) \leq c \cdot g(n) \} \)
  - 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 \( c > 0, n_0 > 0 \text{ s.t. ...} \)
  - ...for all \( n \geq n_0 \), 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. \( n \log n = O((\log n)^2) \)
7. \( 2^{n+1} = O(2^n) \)
8. \( 2^{2n} = O(2^n) \)