CS 375 – Analysis of Algorithms

Professor Eric Aaron

Lecture – M W 1:00pm

Lecture Meeting Location: Davis 117

Business

- There will be a class meeting Wednesday, Mar. 11
- HW8 in already
- HW9 out soon, due Wednesday, Mar. 11
- HW 7 back today; HW6 grading update

- Project1 assignment out today, due by 11:59pm Monday, 3/16
  - [In terms of lateness, please treat this as a hard deadline! …
    Want an extension?]
  - See “Project and Presentation Assignments” on course website
  - Teams for the project have been randomly assigned and are posted on
    the website
  - Auxiliary files also available for download from projects webpage
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. \( 0.01n^4 = O(n^2) \)
6. \( n \log n = O(n^2 \log n) \)
7. \( 2^{n+1} = O(2^n) \)
8. \( 2^n = O(2^n) \)

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.

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

- Definition: \( \theta(g(n)) = \{ f(n) \mid \exists c_1, c_2 > 0 \text{ s.t. } \forall n \geq n_0, 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \} \)

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

- Is each of the below statements true?

1. \( 100n + 5 = \theta(n^2) \)
2. \( 100n + 5 = \Omega(n^2) \)
3. \( n^2/2 - 3n = \theta(n^2) \)
4. \( n^2/2 - 3n = \Omega(n^2) \)
5. \( 100n^2 = \theta(n^3) \)
6. \( 0.01n^3 = \Omega(n^2) \)
7. \( 2^{n+1} = \theta(2^n) \)
8. \( 2^n = \Omega(2^n) \)
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
Insertion Sort: An Example

- Example of insertion sort working on an array (taken from CLRS, section 2.1)

**How would we argue correctness, for the pseudocode in this example?**

![Diagram of insertion sort example](image)

Algorithm Correctness Analysis: Loop Invariants

- To prove algorithms correct, can use *loop invariants* to analyze loops
  - Loop invariants are properties of loops that are true each time through the loop
    - *Initialization*: Property is true before the first iteration
    - *Maintenance*: If a property is true before an iteration, it is true (after that iteration) before the next iteration
    - *Termination*: When the loop terminates, the property (or the violation of the property) is useful in showing algorithm correctness
Insertion Sort and Loop Invariants

- From CLRS (which starts counting at 1), Insertion sort:

\[
\text{INSERTION-SORT}(A)
\]

\[
\begin{align*}
1 & \text{ for } j = 2 \text{ to } A.\text{length} \\
2 & \quad \text{key} = A[j] \\
3 & \quad // \text{ Insert } A[j] \text{ into the sorted sequence } A[1..j-1]. \\
4 & \quad i = j - 1 \\
5 & \quad \text{while } i > 0 \text{ and } A[i] > \text{key} \\
6 & \quad A[i + 1] = A[i] \\
7 & \quad i = i - 1 \\
8 & \quad A[i + 1] = \text{key}
\end{align*}
\]

- Outer loop invariant: sub-array \(A[1..j-1]\) consists of elements originally in \(A[1..j-1]\), but in sorted order
  - How can we explain the correctness of the invariant?
  - How does it help to show the correctness of the overall algorithm?

## Sorting Problem

Input: Sequence of numbers \(a_1, \ldots, a_n\)

Output: Permutation (reordering) \(b_1, \ldots, b_n\) of the input sequence (perhaps leaving them unchanged) such that \(b_1 \leq b_2 \leq \ldots \leq b_n\)

Insertion Sort and Loop Invariants

- From CLRS (which starts counting at 1), Insertion sort:

\[
\begin{align*}
\text{INSERTION-SORT}(A)
\end{align*}
\]

- Use invariant to show correctness:
  - Show invariant is true at loop initialization (How?)
  - Show maintenance step: invariant is true at next iteration, assuming that it's true at the start of this iteration (How?)
  - Use termination condition (What is it?) to show correctness of the overall algorithm (How?)
Insertion Sort and Loop Invariants

• From CLRS (which starts counting at 1), Insertion sort:

```
INSERTION-SORT(A)
1   for j = 2 to A.length
2       key = A[j]
3       // Insert A[j] into the sorted sequence A[1..j-1],
4       i = j - 1
5       while i > 0 and A[i] > key
6           A[i+1] = A[i]
7           i = i - 1
8       A[i+1] = key
```

• **Exercise:** What might an invariant be for the inner loop?

Another example: Bubble Sort

• (Yes, bubble sort is the actual name of this sorting algorithm)
• In pseudocode:

```
BubbleSort(A)
1.   for i = 1 to A.length - 1
2.     for j = A.length downto i+1
```

• How do we argue correctness (i.e., that it sorts A in non-decreasing order)?
  – What might the loop invariant be for the outer loop?
Another example: Bubble Sort

- Loop invariant for outer loop:

  Subarray A[1..i-1] consists of the i-1 smallest values of A, in sorted order, and A[i..n] consists of the remaining values of A (no constraint on order)

  BubbleSort(A)
  1. for i = 1 to A.length - 1
  2. for j = A.length downto i+1

- Is the invariant true at initialization (i = 1)?
- Is the invariant maintained by each iteration of the loop—i.e., is it true at the end of an iteration if we assume it’s true at the beginning?
- What does the invariant show is true at termination of the loop? How does that help us show the algorithm overall is correct?