CS 375 – Analysis of Algorithms

Professor Eric Aaron

Lecture – M W 1:00pm

Lecture Meeting Location: Davis 117

Business

- Thank you again to those who participated in class discussion Oct. 5
  - Please come talk with me about any aspect of that you'd like to discuss!
- Grading update:
  - PS1 returned
  - PS2 in progress
  - Project1 returned for groups that submitted on time / following instructions
    - Slight delay for others—coming soon!
- A note about grading in CS375 in general, and on Proj1 in particular
  - Median grade on Proj1: 41/50. Scales to at least a B+.
  - So, about Sorting Method 3….
- SA3 due already
  - **NOTE:** I found a typo in the hint for part *c* of the invariants-explanation
    - Assignment said loop ended with $i = n+1$, instead of $i = n$
    - Revised SA3 now posted with a revised hint, but no other differences
- PS2 out, due Oct. 24

Business: Project 2

- Project 2 out *very soon*
  - Multi-stage project, with final due date in early November
- **Project 2 is to be done in teams of 4**
  - IMPORTANT: By end of day Wednesday, Oct. 19, one person from each team should email me *and everyone on the team* to let me know they're teaming up
  - If you'd like my help finding a team for you, please let me know!
- Parts of Project 2:
  1. **Design Exhaustive Search Algorithms**: Your team will collectively design exhaustive search algorithms for a variety of problems.
  2. **Improve Time Efficiency**: Your team will pick one of the problems and make your exhaustive search algorithm more efficient.
  3. ***Reduction***: For the same problem chosen for part 2 above, you will *reduce* that problem to one of the other problems given in part 1. (This is a new topic for us—more soon!)
  4. **Create and Give a Presentation**: Your team will present work from the previous three parts of the assignment, *using loop invariants* where appropriate to explain the correctness of your algorithms.

Using Loop Invariants in Algorithm Design and Explanations

- Loop invariants state properties known to be true for each iteration of a loop
  - Can think of it as a property known to be true immediately before and immediately after each iteration
- In our examples, we've understood loop invariants in terms of *before* and *after* conditions
  - *Before*: What is helpful and known to be true before a given iteration of the loop
  - *After*: What is helpful and known to be true after a given iteration of the loop, which establishes the truth of the *Before* condition before the next iteration
- Key point: Loop invariant gives a property that's true *after the loop is finished* that helps explain algo correctness

To (informally) use loop invariants to help explain algo correctness:
- **Explain how the invariant is true before the first iteration of the loop**
- **Explain how the invariant is true after each following iteration**
- **Explain how the invariant property shows that the algo meets its specifications**

A Logical Digression: Logical Reasoning About Empty Stuff

- How does it work if we're asked if something is true for *all* elements of an *empty structure*?
- Examples:
  - Say L is a list of strings, and L is empty. Is it true that every string on the list has length greater than 375?
  - Say S is a set of numbers, and S is empty. Is it true that every number in S is equal to 375?
  - The answer to these questions is **Yes, it is true**. We say it is ***vacuously true***.
- Part of the way *propositional logic* works is that when we ask if some property is true of *every element in some empty structure* …
- … the answer is always yes! Because there are no elements to reason about, we can *vacuously* say *anything* is true of anything in that empty set

This follows from the same idea that once it is shown that False == True, the system is incoherent, so we can say anything is True in that system

A Logical Digression: *Vacuous Truth*

- How does it work if we're asked if something is true for *all* elements of an *empty structure*?
- More Examples:
  - Say L is a list of numbers, and L is empty. Is it true that L is *in sorted order*?
  - Say L is a list of numbers, and L is empty. Is it true that L is *out of sorted order*?
- The answer to these questions is **Yes—they are *vacuously true***

This can be useful as part of using loop invariants, especially when showing a property is true in some boundary case—either *before the first iteration* of a loop or *after the last iteration* of a loop

It's completely understandable if you're not feeling totally happy about this—dealing with logical incoherence is tricky!

It turns out, though, this convention of vacuous truth can be really useful, and it gets very intuitive after a while!

Example of Reasoning with Vacuous Truth: Loop Invariants and Bubble Sort

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

What's the time complexity of this algorithm?

```
BubbleSort(A)
    1. for i  = 1 to A.length - 1
    2.   for j = A.length downto i+1
    3.     if A[j] < A[j-1]
    4.       swap A[j] with A[j-1]
```

- Do you understand how the algo works? (Try it on $A=[7,5,3]$)
- How do we use loop invariants to show correctness (i.e., that it sorts A in non-decreasing order)?

Example of Reasoning with Vacuous Truth: Loop Invariants and Bubble Sort ($O(n^2)$)

- 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
    3.     if A[j] < A[j-1]
    4.       swap A[j] with A[j-1]
```

Sorting Problem

Input: Sequence of numbers $\langle a_1, \ldots, a_n \rangle$

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

- Use invariant to show correctness:
  1. Show invariant is true before first loop iteration (How?)
  2. Show pseudocode ensures invariant is true after each successive iteration, *assuming that it's true at the start of the iteration* (How?)
  3. Show when loop is done, algorithm meets specifications (How?)

Example of Reasoning with Vacuous Truth: Loop Invariants and Bubble Sort ($O(n^2)$)

- 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
    3.     if A[j] < A[j-1]
    4.       swap A[j] with A[j-1]
```

Sorting Problem

Input: Sequence of numbers $\langle a_1, \ldots, a_n \rangle$

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

- Use invariant to show correctness:
  1. Show invariant is true before first loop iteration (How?)

- **Before the first iteration, *i* is set to 1. For the invariant, we look at A[1..(i-1)] = A[1..0].**
- **By convention, A[1..0] is an *empty array*.**
- **So, it *vacuously* contains the (i-1) smallest values of A, in sorted order!**

Example of Reasoning with Vacuous Truth: Loop Invariants and Bubble Sort ($O(n^2)$)

- 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
    3.     if A[j] < A[j-1]
    4.       swap A[j] with A[j-1]
```

Sorting Problem

Input: Sequence of numbers $\langle a_1, \ldots, a_n \rangle$

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

- Use invariant to show correctness:

Super Important Note: Show that the *entire* invariant is true, not just part of it!
- **A[1..0] *vacuously* contains the (i-1) smallest values of A, in sorted order.**
- **We still need to explain how A[1..n] consists of the remaining values of A (no constraint on order)… but that's not that difficult**

Explaining Correctness: Loop Invariants and Bubble Sort ($O(n^2)$)

- 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
    3.     if A[j] < A[j-1]
    4.       swap A[j] with A[j-1]
```

Sorting Problem

Input: Sequence of numbers $\langle a_1, \ldots, a_n \rangle$

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

- Use invariant to show correctness:
  1. Show invariant is true before first loop iteration (**Done!**)
  2. Show pseudocode ensures invariant is true after each successive iteration, *assuming that it's true at the start of the iteration* (How?)
  3. Show when loop is done, algorithm meets specifications (How?)

Explaining Correctness, step 2: Loop Invariants and Bubble Sort ($O(n^2)$)

- 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
    3.     if A[j] < A[j-1]
    4.       swap A[j] with A[j-1]
```

Sorting Problem

Input: Sequence of numbers $\langle a_1, \ldots, a_n \rangle$

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

  2. Show pseudocode ensures invariant is true after each successive iteration, *assuming that it's true at the start of the iteration*

- **There are many right ways to do this step—it's all about explaining how the algo works!**
- **Refer directly to the pseudocode in your explanation, citing specific lines of pseudocode**
- **How might we structure this explanation?**

Explaining Correctness, step 2: Loop Invariants and Bubble Sort ($O(n^2)$)

- 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)

Sorting Problem

Input: Sequence of numbers $\langle a_1, \ldots, a_n \rangle$

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

```
BubbleSort(A)
1. for i  = 1 to A.length - 1
2.   for j = A.length downto i+1
3.     if A[j] < A[j-1]
4.       swap A[j] with A[j-1]
```

2. Show pseudocode ensures invariant is true after each successive iteration, *assuming that it's true at the start of the iteration*

- **One possible explanation idea:**
  - **Each iteration of the inner loop ensures, *after the swap*, that A[j-1] is smaller than A[j]**
  - **For each i, at the end of the inner loop in lines 2-4, the smallest value remaining in A[i..n] has been swapped down the array into position A[i]**

Explaining Correctness, step 3: Loop Invariants and Bubble Sort ($O(n^2)$)

- 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)

Sorting Problem

Input: Sequence of numbers $\langle a_1, \ldots, a_n \rangle$

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

```
BubbleSort(A)
1. for i  = 1 to A.length - 1
2.   for j = A.length downto i+1
3.     if A[j] < A[j-1]
4.       swap A[j] with A[j-1]
```

3. Show when loop is done, algorithm meets specifications

- **There are many right ways to do this step, too!**
- **Refer directly to the invariant property and to the specifications in your explanation—referring to specifications is essential for showing algo correctness**

Explaining Correctness, step 3: Loop Invariants and Bubble Sort ($O(n^2)$)

- 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)

Sorting Problem

Input: Sequence of numbers $\langle a_1, \ldots, a_n \rangle$

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

```
BubbleSort(A)
    1. for i  = 1 to A.length - 1
    2.   for j = A.length downto i+1
    3.     if A[j] < A[j-1]
    4.       swap A[j] with A[j-1]
```

3. Show when loop is done, algorithm meets specifications

- **One possible explanation idea (with important details omitted!):**
  - **Because the invariant is true at the end of the outer loop, when *i = n*, we know *A[1..n-1]* contains the *n-1* smallest values of *A* in sorted order**
  - **Then, A[n] must be the largest value in A [do you see why?], and… [can you fill this in?]**
  - **So, the elements in A are in sorted order [do you see why?], which meets the specifications of the sorting problem, so the algorithm is correct**

Algorithm Correctness and Loop Invariants: A Recap

See CLRS 2.1

- To explain correctness of iterative algos, look at the loop
- Loop invariants can help! Three steps to using loop invariants:
  1. Show the invariant is true before the first iteration
  2. Show the invariant stays true after each successive iteration, assuming it was true before that iteration
  3. Show the algo meets specifications, using what the invariant says is true after the loop is done

These three steps correspond to three parts mentioned in our CLRS textbook:
1. ***Initialization:* Property is true before the first iteration**
2. ***Maintenance:* If a property is true before an iteration, it is true after that iteration / before the next iteration**
3. ***Termination:* When the loop terminates, the property is useful in showing algorithm correctness**

You do not need to use these three terms from CLRS for CS375 when discussing invariants, although you can if you want to—I just wanted to connect our approach in this slides to our textbook

The green-shaded ones are examples of *polynomial time* classes—upper bounded by n^k for some constant *k*. Problems solvable in polynomial time are considered *tractable*. (More about this later in the semester!)

Time Complexity Classes Illustrated!

Complexity Class	*What we call it*	*Example algorithms / objects*
O(1)	Constant	Print "Hello, World!"; stack operations [and much, much more—be careful!]
O(lg n)	Log time	Binary search
O(n)	Linear	Exhaustive search of an array (linear search); Merge (as used in Mergesort)
O(n lg n)	n lg n	Mergesort; Heapsort [Recall: sorting can be done in θ(n lg n)]
O(n^2)	n-squared; quadratic	Insertion / selection / bubble sort; several graph algos
O(n^3)	n-cubed; cubic	*My favorite algorithm*! (a graph algo)
O(2^n)	Exponential	Number of *subsets* of a set of size n
O(n!)	Factorial	Number of orderings / permutations of elements of a list of length n

Constant Time

- O(1) usually refers to primitive operations
  - Printing a finite string (print "Hello, World!")
  - Accessing an array
  - Copying a reference or value into a variable of fixed (constant) size

I might not usually consider these to be "algorithms"…

- … or to a constant number of them to complete a task (compare and swap two elements)—which can be very useful!

Constant Time

- O(1) usually refers to primitive operations
  - Printing a finite string (print "Hello, World!")
  - Accessing an array
  - Copying a reference or value into a variable of fixed (constant) size

I might not usually consider these to be "algorithms"…

- … or to a constant number of them to complete a task (compare and swap two elements)—which can be very useful!
- ***Pro Tip / Warning:*** Technically, by definition of asymptotic complexity, the time complexity of the following are all upper bounded by a constant:
  - Evaluating all possible moves for a turn in chess
  - Solving a standard 9x9 Sudoku
  - Reading our entire CLRS textbook into a new text file

Why are these technically O(1)?

Constant Time

- O(1) usually refers to primitive operations
  - Printing a finite string (print "Hello, World!")
  - Accessing an array
  - Copying a reference or value into a variable of fixed (constant) size

I might not usually consider these to be "algorithms"…

- … or to a constant number of them to complete a task (compare and swap two elements)—which can be very useful!
- ***Pro Tip / Warning:*** Technically, by definition of asymptotic complexity, the time complexity of the following are all upper bounded by a constant:
  - Evaluating all possible moves for a turn in chess
  - Solving a standard 9x9 Sudoku
  - Reading our entire CLRS textbook into a new text file
- But calling these O(1) is not helpful
  - It does nothing to illuminate how algorithms to carry out such tasks vary with input sizes
  - E.g., how would the time complexity for a sudoku-solving algorithm scale up as the grid size got bigger?

Why are these technically O(1)?
- Because they're all constant size, as specified.

But that's not a helpful observation for *asymptotic complexity!*