Exam 1 review: Correctness and complexity

William Hendrix

Lecture 8

Exam 1

Potential topics:

- Proof of correctness
- Proof of incorrectness

at least one

- Big-Oh proof
  - Based on definition
  - Based on properties
  - Log properties

at least one

- Algorithm design
  - Exhaustive search
  - Greedy algorithms
- Worst-case analysis

at least one

  - Iterative algorithms
  - Recursive algorithms
    - Modelling runtime as a recurrence
    - Solving recurrences

Back

Summary: proving correctness

- Prove that it produces the correct output for *every* input
  - Trace input to find algorithm's output
  - Prove output is correct
    - I.e., meets output criteria
- Proof technique depends strongly on algorithm
  - Try algorithm on small examples to see if it works
    - Try to figure out the pattern of what it's doing
  - Can use direct proof for simple algorithms
    - Start with arbitrary input, prove output is correct
    - May need to prove a claim about what each iteration of a loop does
  - Recursive algorithms: induction
    - Prove that it works for the base case, then prove inductive step
  - Optimization algorithms: contradiction
    - Suppose that there were some "better" solution, and show that it's impossible the algorithm missed it

Back

Correctness exercise

- **Algorithm:** Bubble Sort

Input:

`data:` `an array of integers to sort`

`n:` `the number of values in data`

Output: permutation of `data` such that `data[1] ≤ ... ≤ data[n]`

Pseudocode:

```
repeat
   for i = 1 to n-1
    if data[i] > data[i+1]
      Swap data[i] and data[i+1]
    end
  end
until for loop makes no swaps
return data
```

Hint: what happens to the largest value in data in the first iteration of the outer loop?

Back

Incomplete solution

- Bubble Sort is correct.

Proof. Note that Bubble Sort will only terminate when the **if** condition in line 3 is false for all i. Thus, data[1] $\leq$ data[2] $\leq \ldots \leq$ data[n] when Bubble Sort terminates. Thus, Bubble Sort must be correct, *as long as this loop eventually terminates.* $\square$

Important observation

Lemma 1. *After iteration k of the outer loop in Bubble Sort (line 1), the last k values will be the largest k values in the array, in sorted order.*

Proof. We prove the claim by induction.
(*Base case*) Consider the first iteration of the loop, and suppose that the largest value in the array is x = data[j]. The body of the **for** loop in lines 2–6 will not move data[j] until i = j - 1. During this iteration, data[j] = x $\geq$ data[j-1], so x will not be swapped into data[j-1]. In iteration j, data[j] = x $\geq$ data[j+1]. If x > data[j+1], x will be swapped into position data[j+1]. Otherwise, x = data[j+1], so data[j+1] = x in either case. On iteration j+1, data[j+1] = x $\geq$ data[j+2], so data[j+2] will become x, and so forth, until data[n] = x.
(*Inductive step*) Suppose the first k iterations of Bubble Sort have moved the k largest values in the array to the last k positions, and let x = data[j] be the $(k+1)^{\text{st}}$ largest value. Similarly to the base case, x will not be moved before iteration j, but afterwards, data[i] $\geq$ data[i+1] until i = n-k, so data[i+1] will become x. In this way, data[n-k] will become x, so the last $k+1$ values in the array will be the largest $k+1$ values. $\square$

Back

Correctness solution

- Bubble Sort is correct.

Proof. Note that Bubble Sort will only terminate when the **if** condition in line 3 is false for all i. Thus, data[1] $\leq$ data[2] $\leq \ldots \leq$ data[n] when Bubble Sort terminates. Thus, Bubble Sort must be correct, as long as this loop eventually terminates.
By the lemma, Bubble Sort moves the k largest values in the array to the end after k iterations of the outer loop. So, after (at most) n iterations of this loop, all n values will in sorted order. At this point, the inner **for** loop won't swap anything, and the outer loop will terminate. Since Bubble Sort always terminates with the correct output, Bubble Sort is correct. $\square$

Back

Summary: proving incorrectness

- Proof by counterexample
  - Find *one* instance with an incorrect solution
  - Typically easier than induction
  - Counterexample may be tricky to find

- Counterexample strategies
  - Start small
  - Think about how the algorithm deals with extremes
    - Large and small
    - Near and far
    - Large range vs. all identical values
  - Look at the algorithm for a hint about its weaknesses
    - Step through with one example
    - Check if a modification to the input could break the algorithm

Back

Incorrectness exercise

- **Problem:** swap
  - **Input:** pointers to two integers, *a* and *b*
  - **Output:** none, but the values of *a* and *b* should be swapped

```
void swap(int* a, int* b)
{
  *a = *a - *b;
  *b = *a + *b;
  *a = *a - *b;
}
```

  - **Example:**

```
*a = *a - *b;
*b = *a + *b;
*a = *b - *a;
```

***a**	***b**
100	73
27	73
27	100
73	100

Back

Incorrectness sample solution

- Prove that swap (below) is incorrect:

```
void swap(int* a, int* b)
{
1   *a = *a - *b;
2   *b = *a + *b;
3   *a = *b - *a;
}
```

```
If a ≠ b:
b = (a - b) + b = a
a = a - (a - b) = b
```

Proof. The swap algorithm is provably correct if $a \neq b$. However, if $a = b$, both will equal 0 after line 1, and continue to (both) equal 0 thereafter, whereas a correct algorithm would not modify the value of `*a` or `*b`. $\square$

Back

Summary: complexity and Big-Oh

- RAM model of computation
- Useful approximation of real-world behavior:
  - Basic instructions take same amount of time
  - Memory access is instantaneous
- Key aspect of complexity: asymptotic growth
  - How fast does the function grow?
  - Constant, logarithmic, linear, etc.?
- Big-Oh: classify functions according to growth rate

$f(n) = O(g(n))$ if and only if there exist positive constants c and n_0 such that $f(n) \le cg(n)$ for all $n \ge n_0$.

$f(n) = \Omega(g(n))$ if and only if there exist positive constants c and n_0 such that $f(n) \ge cg(n)$ for all $n \ge n_0$.

$f(n) = \Theta(g(n))$ if and only if there exist positive constants c_1, c_2, and n_0 such that $c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0$.

- *Analogy:* O, Ω, and Θ "act like" ≤, ≥, and =

[Back](#)

Big-Oh properties

- **Interrelationships:** O and Ω are "*opposite*", Θ is "*composite*"

  $$f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$$
  $$f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n)) \Leftrightarrow f(n) = \Theta(f(n))$$

- **Reflexive**

  $$f(n) = O(f(n)), \text{ for any function } f$$

- *Θ only*: **Symmetric**

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

- **Transitive**

  $$f(n) = O(g(n)) \text{ and } g(n) = O(h(n)) \rightarrow f(n) = O(h(n))$$

- **Ignore constant coefficients**

  $$\forall x > 0, xf(n) = O(f(n))$$

- **Ignore small terms**

  $$f(n) = O(g(n)) \rightarrow \Theta(f(n) + g(n)) = \Theta(g(n))$$

- **Envelopment** (+ and *)

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

Back

Big-Oh exercise 1

- Prove the multiplicative envelopment property of Big-Omega:

  $j(n)k(n) = \Omega(f(n)g(n))$,
  where $j(n) = \Omega(f(n)$ and $k(n) = \Omega(g(n))$

Back

Big-Oh sample solution 1

- Prove the multiplicative envelopment property of Big-Omega:

  $j(n)k(n) = \Omega(f(n)g(n))$,
  where $j(n) = \Omega(f(n)$ and $k(n) = \Omega(g(n))$

Proof. By the definition of Big-Omega, there exist some positive constants c_1, n_1, c_2, and n_2 such that $j(n) \geq c_1 f(n)$ for all $n \geq n_1$ and $k(n) \geq c_2 g(n)$ for all $n \geq n_2$.
Let $n_3 = \max\{n_1, n_2\}$. Since $n_3 \geq n_1$ and $n_3 \geq n_2$, $j(n) \geq c_1 f(n)$ and $k(n) \geq c_2 g(n)$ for all $n \geq n_3$. Multiplying both sides of these inequalities, we see that $j(n)k(n) \geq c_1 f(n)(c_2 g(n)) = (c_1 c_2) f(n) g(n)$ for all $n \geq n_3$.
Thus, there exist constants $c_3 = c_1 c_2$ and $n_3 = \max\{n_1, n_2\}$ such that $j(n)k(n) \geq c_3 f(n) g(n)$ for all $n \geq n_3$, so $j(n)k(n) = \Omega(f(n)g(n))$, by the definition of Big-Omega. $\square$

Big-Oh exercise 2

Prove that if $g(n) \neq O(1)$ and $f(n)$ is positive everywhere, $f(n)g(n) \neq O(f(n))$.

Back

Big-Oh exercise 2

Prove that if $g(n) \neq O(1)$ and $f(n)$ is positive everywhere, $f(n)g(n) \neq O(f(n))$.

Proof. By the negation of the definition of Big-Oh, for any positive constants c and n_0, there must exist some $n \geq n_0$ such that $g(n) > c$.

Let $f(n)$ be a positive function, and consider $f(n)g(n)$. If c and n_0 are constants, there is some $n \geq n_0$ such that $g(n) > c$. Since $f(n) > 0$, we can multiply both sides by $f(n)$ and see that $f(n)g(n) > cf(n)$, so $f(n)g(n) \neq O(f(n))$. $\square$

Back

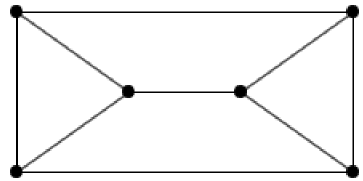
Summary: algorithm design

- Try to understand the problem
  - Solve small examples
- Apply algorithm design strategies
- **Strategy:** exhaustive search
  - Test all possibilities for the solution
  - Report the correct/best solution
  - Always works, often unacceptably slow
- **Strategy:** greedy algorithms
  - Applies to optimization (maximize, minimize, etc.)
  - Select the "best" possible element to add to the solution
  - Repeat until there are no more possible elements to add
  - Generally efficient, may not find optimum (incorrect)

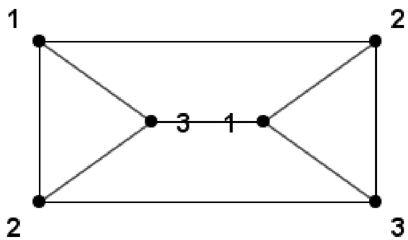
[Back](#)

Algorithm design exercise

- **Problem:** Graph coloring
  - **Input:** a graph network *G* and a coloring number *n*
  - **Output:** an assignment of colors (1..*n*) to the nodes of G such that no nodes connected by an edge are the same color, or "no such coloring" if none exists
  - **Example:** $n = 3$, *G*:



    - **Possible solution:**



1. Design a greedy algorithm that finds a graph coloring.
   - **Hint:** *N(v)* is the set of *neighbors* of *v*; i.e., the vertices joined to *v* by an edge

Back

Algorithm design sample solution

Input: $G = (V, E)$: a graph with vertices V and edges E
Input: n: the proposed coloring number of G
Output: A valid coloring of G that uses no more than n colors, or "No such coloring"
Algorithm: GreedyColoring

```
while some vertex of G is not yet colored do
    Let v be an uncolored vertex of G;
    for c = 1 to n do
        if no vertex of N(v) has color c then
            Assign color c to v;
            break;
        end
    end
    if v was not colored then
        return "No such coloring";
    end
end
return G;
```

Back

Summary: algorithm analysis

- Identify loops and function calls
  - Everything else is *O(1)*
- *For loops*:
  - Estimate number of iterations
    - Incrementing by *c*: divide range by *c* to get iterations
    - Multiplying by *c:* take $\log_c$ of the end/start ratio
  - Estimate loop body running time
    - Might depend on iteration #
  - If iterations don't depend on i: # iterations * time per iteration
  - Otherwise: add up all iteration times (summation)
- *For functions:*
  - Analyze other functions separately
  - Recursive functions: set up a recurrence and solve
- Overall complexity: largest loop or function call complexity

Back

Analysis exercise

- Find the worst-case complexity for Bubble Sort:

Input:

```
data:  an array of integers to sort
n:     the number of values in data
```

Output: permutation of `data` such that `data[1] ≤ ... ≤ data[n]`

Pseudocode:

```
1 repeat
2    for i = 1 to n-1
3     if data[i] > data[i+1]
4       Swap data[i] and data[i+1]
5     end
6   end
7 until for loop makes no swaps
8 return data
```

Hint: After iteration k of the outer loop in Bubble Sort (line 1), the last k values will be the largest k values in the array, in sorted order.

Back

Analysis exercise sample solution

- Bubble Sort is $O(n^2)$.

Proof. The inner loop (lines 2–6) iterates n times, and all operations inside the loop are $O(1)$, for a total of $O(n)$ time total.
Since the last k elements of the array are the k largest values, in order, after iteration k of the outer loop, the entire array must be sorted after at most n iterations. Since each iteration of the outer loop takes $O(n)$ time, this is a total of $O(n^2)$. However, the array could be sorted in as few as one iteration of the loop, so this estimate could potentially be an overestimate.
We show that $O(n^2)$ is a tight upper bound by proving that Bubble Sort is $\Omega(n^2)$. Consider an array where the minimum value is the last value. Note that the min value is not moved until i is one less than its location in the inner **for** loop, and after it swaps the min with the previous value in the array, i is larger than its position, so it isn't swapped again. As such, the min element will only move left one position each iteration of the outer loop, so it will require a total of $n - 1 = \Omega(n)$ iterations until this element is in its correct position. Since each iteration takes $\Omega(n)$ time, Bubble Sort will take $\Omega(n^2)$ time total on this type of input, so its worst-case complexity is $\Theta(n^2)$. $\square$

Back

Summary: solving recurrences

Linear nonhomogeneous recurrences with constant coefficients

$$T(n) = c_1 T(n-1) + c_2 T(n-2) + \ldots + c_k T(n-k) + f(n)$$

1. Write down the characteristic polynomial
   $$c(x) = x^k - c_1 x^{k-1} - c_2 x^{k-2} - \ldots - c_{k-1} x - c_k$$
   - Degree = k (# of previous terms)
   - First coefficient = 1
   - Other coefficients are negation of coeff. from eqn, in decreasing order
   - Don't forget 0 coefficients!
2. Find the roots of the characteristic polynomial
3. Write the general form of solution
   $$T(n) = O(n^{m_1 - 1}) {r_1}^n + \ldots + O(n^{m-1-1}) {r_k}^n + O(n^m f(n))$$
   - $r_1, \ldots, r_k$ are roots of char. poly.
   - Add multiple of n for each additional root (multiple roots)
   - Add $O(n^m f(n))$ to the end, where m is multiplicity of 1
4. Simplify

Back

Recurrence exercise

- Identify the worst-case complexity of the PolyEval algorithm (below):

- **Algorithm:** PolyEval

Input:

```
d:     the degree of the polynomial to evaluate
coeff: the coefficients of the polynomial (largest to smallest)
x:     the point at which to evaluate the polynomial
```

Output: the value of *f(x)*

```
if d = 1
  return coeff[1]
else
  Let temp = PolyEval(d-1, coeff[1..d-1], x)
  return x * temp + coeff[d]
end
```

Back

Recurrence exercise sample solution

$$T(d) = \begin{cases} O(1), & \text{if } d = 0 \\ T(d-1) + O(1), & \text{if } d > 0 \text{ and we use pointers} \\ T(d-1) + O(n), & \text{if } d > 0 \text{ and we copy coeff[1..d-1]} \end{cases}$$

If $d = 0$, we will only execute the **if** condition and the **return** statement in line 2, for a total of $O(1)$ time.
If $d > 1$, we will execute the **if** condition ($O(1)$ time), calculate $d-1$ ($O(1)$), calculate coeff[1..d-1] (depends), call PolyEval on coeff[1..d-1] ($T(d-1)$), then multiply by x, add coeff[d], and return (all $O(1)$). If we assume that we make a copy of coeff[1..d-1], this will take $O(d)$ time, but if we just use pointers, this will just take $O(1)$ time.
In the case we make a copy, $T(d) = T(d-1) + O(d)$, as the cost to make the copy will dominate all of the constant-time instructions. However, if we just use pointers, all of the instructions other than the recursive call will be constant time, for a total of $T(d) = T(d-1) + O(1)$.

Recurrence exercise sample solution

If we make a copy of the array, $T(n) = O(n^2)$. If we just use pointers, $T(n) = O(n)$.

In the case we make a copy, $T(d) = T(d-1) + O(d)$. The characteristic equation will be $c(x) = x-1$, which has a zero at $r = 1$. So, the general form of the solution will be $T(n) = O(1^n) + O(n^m f(n)) = O(1) + O(n^m f(n))$. Since 1 is a single root of the characteristic equation, $m = 1$, so $n^m f(n) = n^1 O(n) = O(n^2)$. Thus, $T(n) = O(1) + O(n^2) = O(n^2)$.
In the case we use pointers, $T(d) = T(d-1) + O(1)$. The characteristic equation (and its zero) will be the same, so the general form of the solution will be $T(n) = O(1^n) + O(n^m f(n)) = O(1) + O(n^m f(n))$. Since 1 is a single root of the characteristic equation, $m = 1$, so $n^m f(n) = n^1 O(1) = O(n)$. Thus, $T(n) = O(1) + O(n) = O(n)$.

Coming up

- **Exam 1** will be Tuesday
  - Single-sided sheet of notes if you bring **Feedback Form 1**
  - Practice exam and sample solution posted
- Data structures after Exam 1