Introduction to complexity classes

William Hendrix

Lecture 3

Today

- Review
  - Loop invariant and incorrectness examples

- Introduction to complexity classes
  - Big-Oh
  - Big-Omega
  - Big-Theta

Correctness and incorrectness

- To prove an algorithm is correct:
  - Prove that it produces the correct output for *every* input
    - Trace input to find algorithm's output
    - Prove output is correct
  - Loop invariants for loops
  - Cases for if statements
  - Recursion: generally proof by induction
  - May also use proof by contradiction

- To prove an algorithm is incorrect:
  - Find a *counterexample*
    - Instance where the algorithm computes an incorrect solution

Loop invariant example

- **Algorithm:** SelectionSort

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:

```
for i = 1 to n
  Let m be the location of the min value in the array data[i..n]
  Swap data[i] and data[m]
end
return data
```

- **Invariant:** After iteration *i*, `data[x]` ≤ all values that follow it, for all *x* from 1 up to *i*

Variant invariant solution

- **Invariant:** After iteration *i*, `data[x]` $\leq$ all values that follow it, for all *x* from 1 up to *i*

  Proof. (1^{st} *iteration*) During the first iteration, the min of *data* is swapped with $data[1]$, so $data[1]$ is less than or equal to all of the other values by the definition of the minimum.
  ($(k+1)^{st}$ *iteration*) Suppose that each $data[x]$ is less than or equal to everything that follows it up to $data[k]$ after iteration k. During iteration $k+1$, the min of $data[k+1..n]$ is swapped to position i, so it will be less than or equal to everything that follows it. Since no previous elements in the array were moved, all of the previous inequalities still hold, so now all $data[x]$ up to $k+1$ are less than or equal to all elements that follow them.
  Hence, the loop invariant holds for every iteration of the loop. □

  Proof of correctness. By the loop invariant, after the $i = n$ (last) iteration of the for loop, all n elements of *data* will be less than or equal to everything that follow them. In particular, they will all (other than $data[n]$) be less than or equal to the subsequent element, so $data[1] \leq data[2] \leq \ldots \leq data[n]$. Thus, the output of SelectionSort is sorted. □

Loop invariant exercise

- **Algorithm:** InsertionSort

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:

```
for i = 2 to n
  Let ins = data[i]
  Let j = last index of data[1..i-1] <= ins (or 0 if all > ins)
  Shift data[j+1..i-1] one space to the right
  data[j+1] = ins
end
return data
```

- **Invariant:** after iteration *i*, `data[1..i]` is sorted.

Exercise solution

- **Invariant:** after iteration *i*, `data[1..i]` is sorted.

Proof. (1^{st} iteration) When $i = 2$, ins becomes $data[2]$. We have two cases for the value of j.
(*Case 1*) If $data[1] \leq data[2]$, $j = 1$, nothing is shifted right, and $data[2] = ins = data[2]$. Since $data[1] \leq data[2]$, $data[1..2]$ is sorted.
(*Case 2*) If $data[1] > data[2]$, $j = 0$, $data[1]$ is shifted into $data[2]$, and $data[1] = ins$, the original value of $data[2]$. Since this value is smaller than the original value of $data[1]$, $data[1] < data[2]$, and $data[1..2]$ is sorted.
Hence, in both cases, $data[1..i]$ is sorted.
($(k+1)^{\text{st}}$ iteration) Suppose that $data[1] \leq data[2] \leq \ldots \leq data[k]$ after iteration k. In iteration $k+1$, $i = k+1$, so ins becomes $data[k+1]$. Then, j becomes the last value of $data[1..k]$ such that $data[j] \leq ins$ (or 0), and $data[j+1..i-1]$ are all shifted right. After this shift, $data[1] \leq data[2] \leq \ldots \leq data[j]$ and $data[j+2] \leq \ldots \leq data[k+1]$ because of the previous iteration. When $data[j+1]$ becomes ins, $data[j+2]$ must be greater than ins, and (if $j > 0$) $data[j] \leq ins$, both by the definition of j. Hence, $data[1] \leq \ldots \leq data[j] \leq data[j+1] < data[j+2] \leq \ldots \leq data[k+1]$, so $data[1..k+1]$ is sorted.
Therefore, by induction, $data[1..i]$ is sorted after every iteration of the for loop. $\square$

The importance of pseudocode

- Prove that BadSort (below) is not a correct sorting algorithm.

Input:

```
data:  an array of integers to sort
n:     the number of values in data
for i = n-1 to 1 step -1
  for j = 1 to n-i step i
    if data[j] > data[j+i]
      Swap data[j] and data[j+i]
    end
  end
end
```

- What does this even mean?

Incorrectness example

- Prove that BadSort (below) is not a correct sorting algorithm.

Input:

```
data:  an array of integers to sort
n:     the number of values in data
for i = n-1 to 1 step -1
  Consider every ith value of data, starting at data[1]
  if two of these values are out of order,
    Swap them
end
```

Incorrectness example

- Prove that BadSort (below) is not a correct sorting algorithm.

Input:

```
data:  an array of integers to sort
n:     the number of values in data
for i = n-1 to 1 step -1
  Consider every ith value of data, starting at data[1]
  if two of these values are out of order,
    Swap them
end
```

$i = 4$

- Input: 3 2 5 4 1

Incorrectness example

- Prove that BadSort (below) is not a correct sorting algorithm.

Input:

```
data:  an array of integers to sort
n:     the number of values in data
for i = n-1 to 1 step -1
  Consider every ith value of data, starting at data[1]
  if two of these values are out of order,
    Swap them
end
```

$i = 3$

- Input: 1 2 5 4 3

Incorrectness example

- Prove that BadSort (below) is not a correct sorting algorithm.

Input:

```
data:  an array of integers to sort
n:     the number of values in data
for i = n-1 to 1 step -1
   Consider every ith value of data, starting at data[1]
   if two of these values are out of order,
      Swap them
end
```

$i = 2$

- Input: 1 2 5 4 3

Incorrectness example

- Prove that BadSort (below) is not a correct sorting algorithm.

Input:

```
data:  an array of integers to sort
n:     the number of values in data
for i = n-1 to 1 step -1
  Consider every ith value of data, starting at data[1]
  if two of these values are out of order,
    Swap them
end
```

$i = 1$

- Input: 1 2 3 4 5

Incorrectness example

- Prove that BadSort (below) is not a correct sorting algorithm.

Input:

```
data:  an array of integers to sort
n:     the number of values in data
for i = n-1 to 1 step -1
  Consider every ith value of data, starting at data[1]
  if two of these values are out of order,
    Swap them
end
```

- Counterexample: [1, 3, 5, 2, 4], which outputs [1, 3, 2, 4, 5]

Complexity analysis

- What are the factors that contribute to the running time of an algorithm?
  - Processor speed
  - Number of instructions executed
  - Cache coherency
  - Resource conflicts (network, hard disk, etc.)

- Which of these are important when comparing algorithms?
  - Processor speed affects fast and slow algorithms equally
    - Not an important factor

- What can we *most reliably* control when designing an algorithm?
  - Number of instructions executed

RAM model of computation

- Set of assumptions that make analysis more reasonable

Assumptions

1. All "basic" operations (assignment, arithmetic, branching, etc.) take 1 operation
   - Loops and functions do not qualify
2. Memory access is instantaneous
   - All variables are in registers
3. We have "infinite" memory

Cons

- Different operations take different number of clock cycles
- Cache locality has significant impact on performance
- Virtual memory can slow performance

Pros

- Can actually analyze algorithms

RAM model example

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

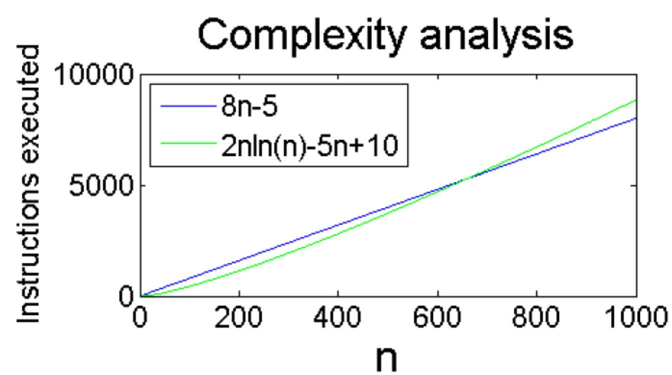
Min algorithm:

	Ops per line	Times executed
`min = 1`	1	1
`for i = 2 to n`	2	n-1 (or n)
`  if data[i] < data[min]`	4	n-1
`    min = i`	1	??? (≤ n-1)
`  end`	0	n/a
`end`	1	n-1
`return min`	1	1

Total ops: $\le 8(n-1) + 3 = 8n - 5$

Question: is this better or worse than an algorithm that takes at most $2n \ln n - 5n + 10$ ops?

Better unless n < 658



Big-Oh notation

- Technique for *abstracting away details* of complexity
  - Can be used for time complexity, space complexity, etc.

- **Main idea:** most important aspect of complexity is *how fast it grows* relative to input size
  - Focus on asymptotic (eventual) growth rate
  - "Fast" functions will eventually pass "slow" functions for large *n*
  - Coefficients only matter if growth rate is similar
  - Predicting behavior for small *n* is difficult and often pointless

- Big-Oh notation
  - Organizes growth rates into classes
  - Three main symbols: $O(f(n)), \Omega(f(n)), \Theta(f(n))$
    - Analogous to "at least", "at most", and "similar to" *f(n)*

Big-Oh

- Upper bound *("at most")*

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

- We say "*g(n)* dominates *f(n)*" when *f(n) = O(g(n))*
- Notation weirdness:
  - *O*, *Ω*, and *Θ* are classes (sets) of functions
  - BUT: we use = to assign class, not ∈
- **Example**
  - Prove that $7n^2 + 19n - 4444 = O(n^2)$.

Proof. If $n \geq 19$,

$$\begin{aligned} 7n^2 + 19n - 4444 &\leq 7n^2 + 19n \\ &\leq 7n^2 + n^2 \\ &= 8n^2 \end{aligned}$$

Therefore, there exist positive constants $c = 8$ and $n_0 = 19$ such that $7n^2 + 19n - 4444 \leq cn^2$ for all $n \geq n_0$. □

Coming up

- Big-Omega and Big-Theta
- Big-Oh notation
- Big-Oh properties
- Logarithms review

- **Homework 3** is due Friday
- **Homework 2** is due tomorrow

- **Recommended readings:** Sections 1.2, 2.2, and 3.1
- **Practice problems:** 2.2-, 2.2-3, 3.1-1