Algorithm correctness

William Hendrix

Lecture 2

Today

- Formal definitions
- Algorithm correctness
- Loop invariants

What is an algorithm?

- **algorithm:** a well-defined, step-by-step procedure for solving a *problem*
  - Can be more than just computer programs

- **problem:** general task in which we are given some **input** and need to compute some corresponding **output**
  - **Example**
    - The *minimum problem*: given a set of values that can be compared, output the smallest value in that set

- **instance:** a particular input for a problem
- **solution:** the corresponding output for a problem instance
  - **Examples** *(minimum)*
    - (5, 3, 14) -> 3
    - (-1, -1, -1, -1, -1) -> -1
    - ("Gallup", "Trot", "Canter") -> "Trot"

Let's look at a concrete example

- **Problem:** sorting a list of numbers
- **Algorithm:**

Input: $data$: an array of integers to sort
Input: n: the size of $data$
Output: permutation of $data$ such that
$data[1] \leq data[2] \leq \ldots \leq data[n]$
Algorithm: SelectionSort
for $i = 1$ to n **do**
  Let m be the location of the min value in $data[i..n]$;
  Swap $data[m]$ and $data[i]$;
end
return $data$;

Formal definition of problem

Pseudocode

- Pseudocode
  - Algorithm description between code and English
  - Code-like to *eliminate ambiguity*
  - English-like for *simplicity*
- Convention: most of my pseudocode will number arrays 1 to *n*
  - *data[i..j]* represents subarray of *data* from index *i* to index *j*

How do we determine correctness?

- **Correct**
  - For every input, algorithm must *terminate* with *solution*
- Proof of correctness
  1. Start with *arbitrary* input
  2. Describe what the algorithm does
     - If statements: sometimes proof by cases
     - For/do/while loops: often need *loop invariants*
     - Recursive function calls: induction or strong induction
     - Other function calls: solves associated problem
  3. Prove that output meets problem criteria
     a. And algorithm terminates
- Sometimes you can use *contradiction*
  - Often used for *optimization* algorithms (max/min, best/worst, etc.)
  - Suppose algorithm gets the wrong answer
  - Show this leads to a contradiction

Proofs vs. software testing

- Why don't we just test algorithms?

1. Testing does not guarantee correctness
2. Algorithms ≠ computer programs
3. Coding an algorithm is time-consuming and error-prone

- Why don't we use proofs for all programs?

1. Problem not always easy to define mathematically
2. Code is *very* easy to misinterpret
3. Realistic programs are large and complex

Correctness example

- Prove that the algorithm below computes 2^n when given a nonnegative integer n as an input:

```
Input: n: nonnegative integer
Output: 2^n
Algorithm: TwoToThe
if n = 0 then
|   return 1;
else
|   return 2 · TwoToThe(n − 1);
end
```

Correctness example

- Prove that the algorithm below computes 2^n when given a nonnegative integer n as an input:

Input: n: nonnegative integer
Output: 2^n
Algorithm: TwoToThe
if $n = 0$ **then**
  return 1;
else
  return $2 \cdot \text{TwoToThe}(n-1)$;
end

Proof. We prove the claim by induction on n.

(*Base case*) If $n = 0$, the first if statement is true, so TwoToThe returns 1. $2^0 = 1$, so TwoToThe returns 2^n in this case.

(*Inductive step*) Suppose that TwoToThe returns 2^k when given k as an input, for some $k \geq 0$, and consider $n = k+1$. Since $k \geq 0$, $k+1 \neq 0$, so the first if statement will be false. As a result, TwoToThe will return $2 \cdot \text{TwoToThe}(k)$. By the inductive hypothesis, TwoToThe(k) will return 2^k, so this value will equal $2(2^k) = 2^{k+1}$.

Therefore, TwoToThe will return 2^n for all $n \geq 0$, by induction. $\square$

Loop invariant

- Critical for correctness of algorithms with loops
- Statement that is true for every iteration of a loop
- Right invariant can make your proof much easier

- **To prove a loop invariant:**
  - Prove claim holds after the first iteration
  - Prove claim after iteration k implies claim after iteration k+1
- *Resembles induction!*

- **Alternate proof strategy:**
  - Prove claim holds before first iteration
  - Prove claim before iteration k implies claim after iteration k

Proof of correctness

```
Input: n: nonnegative integer
Output: 2^n
Algorithm: IterTwoToThe
p = 1;
for i = 1 to n do
    p = 2p;
end
return p;
```

Invariant: after iteration i, $p = 2^i$

- Develop loop invariant
  - How does loop get closer to solution?
- Prove loop invariant
- Prove correctness

Proof of correctness

```
Input: n: nonnegative integer
Output: 2^n
Algorithm: IterTwoToThe
p = 1;
for i = 1 to n do
    p = 2p;
end
return p;
```

- Develop loop invariant
  - How does loop get closer to solution?
- Prove loop invariant
- Prove correctness

Invariant: after iteration i, $p = 2^i$

Proof. (1^{st} iter) Before the first iteration, $p = 1$. During the first iteration, p is doubled, so $p = 2$ after the first iteration. Thus, $p = 2^i$ after iteration 1.

(k^{th} iter) Suppose that $p = 2^k$ after the k^{th} iteration of the loop. In the $(k+1)^{\text{st}}$ iteration, p is doubled, so p becomes $2(2^k) = 2^{k+1}$.

Thus, $p = 2^i$ after every iteration i of the for loop. □

Proof. When $n = 0$, $p = 1$ is returned, and $2^n = 1$. Otherwise, by the loop invariant, $p = 2^i$ after every iteration of the for loop, so $p = 2^n$ after iteration n, so TwoToThe returns 2^n when the loop iterates. □

More complex 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
```

Loop invariant solution

- **Invariant:** After iteration *i*, `data[1]` ≤ … ≤ `data[i]` and `data[i]` ≤ all values that follow it

Proof. Note that the second condition follows from the fact that $data[i]$ was selected as the minimum of $data[i..n]$ in lines 2 and 3.
(1^{st} *iteration*) $data[1] \leq data[1]$, trivially.
($(k+1)^{st}$ *iteration*) Suppose that $data[1] \leq \ldots \leq data[k]$ and $data[k]$ is less than or equal to all values that follow it after iteration k. During iteration $k+1$, the min of $data[k+1..n]$ is swapped to position i. Since $data[k]$ is less than or equal to this value, $data[k] \leq data[k+1]$, so $data[1] \leq \ldots \leq data[k] \leq data[k+1]$.
Hence, the loop invariant holds for every iteration of the loop. $\square$

Correctness exercise

- Prove that the following algorithm correctly identifies the location of the minimum value in the array `data`.
  - Can be solved by loop invariant or contradiction

Input:

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

Output: index `min` such that `data[min] ≤ data[j]`, for any `j` between 1 and `n`

Pseudocode:

```
min = 1
for i = 2 to n
  if data[i] < data[min]
    min = i
  end
end
return min
```

Correctness exercise solution

- Prove that the previous algorithm correctly identifies the location of the minimum value in the array `data`.

Proof. First, we show that after iteration i of the loop, $data[min] \leq data[j]$ for all $j \leq i+1$.

(*1st iteration*) Before the loop, $min = 1$, and in the first iteration, min becomes 2 if $data[2] < data[1]$. Thus, if $data[1] \leq data[2]$, $min = 1$, and $min = 2$ otherwise. $data[min] \leq data[1]$ and $data[min] \leq data[2]$, in either case.

($k+1^{st}$ *iteration*) Suppose that $data[min] \leq data[j]$ for all $j \leq k+1$ after iteration k, and consider the next iteration. If $data[min] \leq data[k+2]$, $data[min]$ will be the minimum up to element $k+2$. Otherwise, $data[k+2] < data[min]$, min will set to $k+2$, and $data[min]$ will be the minimum up to element $k+2$. Hence, $data[min] \leq data[j]$ for all $j \leq k+1$ after every iteration k of the `for` loop.

Since $data[min] \leq data[j]$ for all $j \leq k+1$ after each iteration of the loop, $data[min] \leq data[j]$ for all $j \leq n$ after the $n-1^{st}$ iteration. Hence, min will point to the array minimum at the end of the algorithm. $\square$

Correctness exercise solution (2)

- Prove that the previous algorithm correctly identifies the location of the minimum value in the array `data`.

Proof. We prove the claim by contradiction. Suppose that the algorithm does not find the minimum; i.e., suppose that the algorithm returns a value m, but there is some x such that $\mathrm{data}[x] < \mathrm{data}[m]$. Consider the x^{th} iteration of the **`for`** loop in line 2. (Note, at this point, min might not equal m yet.) There are two possibilities at this point.
(*Case 1*: $\mathrm{data}[x] < \mathrm{data}[\mathrm{min}]$) If $\mathrm{data}[x] < \mathrm{data}[\mathrm{min}]$, min will be assigned the value x. However, it is not possible for the algorithm to return the value m now because $\mathrm{data}[m]$ will not be less than $\mathrm{data}[\mathrm{min}]$ on iteration m of the **`for`** loop. This is impossible, as we assumed that the algorithm returned m.
(*Case 2*: $\mathrm{data}[x] \geq \mathrm{data}[\mathrm{min}]$) Since $\mathrm{data}[\mathrm{min}] \leq \mathrm{data}[x] < \mathrm{data}[m]$, it is not possible for the algorithm to return m, as in the previous case.
Thus, in either case, we reach a contradiction, so there must not be any x such that $\mathrm{data}[x] < \mathrm{data}[m]$. Hence, the algorithm is correct. $\square$

Coming up

- Incorrectness example
- Introduction to complexity
  - RAM model
  - Big-Oh notation

- **Homework 2** will be posted on Canvas
  - Due next Wednesday
- **Homework 1** is due Friday
- **Feedback form 1** is due at Exam 1

- **Recommended readings:** Section 2.1
- **Practice problems** (not required): 2.1-3