Correctness practice and algorithm design

William Hendrix

Lecture 2

Today

- SDS announcement
- Review
- Proving incorrectness
- Algorithm design strategies
  - Exhaustive search
  - Greedy algorithms

Terminology

- **algorithm:** decision procedure for solving a problem
- **problem:** task to take **input** and compute its associated **output**
- **instance:** a particular input for a problem
- **solution:** the corresponding output for a problem instance

- **correct:** an algorithm that <u>terminates</u> with the <u>correct output</u>, for <u>every problem instance</u>
- **efficient:** an algorithm that terminates quickly, using as few resources as possible
- **elegant:** an algorithm that is easy to understand

Correctness review

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

```
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
```

- **Proof idea:** After iteration *i*, the first *i* items are less than or equal to everything that follows them. After *n* iterations, each `data[i]` will be less than or equal to `data[i+1]`.

Proof review

- Induction
  - Proof technique for statements of the form "*P(i)*, for all $i \geq b$."
  - Two parts (both required)
    - Base case: prove *P(b)*
    - Inductive step: prove $P(k) \rightarrow P(k+1)$, for any $k \geq b$.

- Contradiction
  - Assume claim is false
  - Show something impossible must be true

Induction exercise

Prove that $n! > 2^{n+1}$ for all $n \geq 5$.

Induction solution

Prove that $n! > 2^{n+1}$ for all $n \geq 5$.

Proof. (*Base case*) $5! = 120$, and $2^{5+1} = 2^6 = 64$. Since $120 > 64$, $n! > 2^{n+1}$ for $n = 5$.
(*Inductive step*) Suppose that $k! > 2^{k+1}$, for some $k \geq 5$, and consider $n = k + 1$. $n! = (k+1)! = (k+1)k!$, while $2^{n+1} = 2^{k+2} = 2(2^{k+1})$.

$$\begin{aligned}
n! &= (k+1)k! && (1)\\
&> (k+1)2^{k+1} && (2)\\
&> 2(2^{k+1}) && (3)\\
&= 2^{n+1} && (4)
\end{aligned}$$

(Note: in line 3, $k+1 > 2$ because $k \geq 5$.) Since $k! > 2^{k+1}$ implies that $(k+1)! > 2^{k+2}$, $n! > 2^{n+2}$ for all $n \geq 5$, by induction.

$\square$

Correctness exercise

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

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. 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 data$[x] <$ 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*: data$[x] <$ data[min]) If data$[x] <$ data[min], min will be assigned the value x. However, it is not possible for the algorithm to return the value m now because data$[m]$ will not be less than data[min] on iteration m of the **`for`** loop. This is impossible, as we assumed that the algorithm returned m.
(*Case 2*: data$[x] \geq$ data[min]) Since data[min] $\leq$ data$[x] <$ 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 data$[x] <$ data$[m]$. Hence, the algorithm is correct. $\square$

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 induction on n.
(*Base case*) If `data` contains one element, `min` is assigned to be 1 at the beginning, and the `for` loop doesn't execute, so the algorithm returns 1. `data[1]` is the min of a one-element array trivially, so the algorithm is correct for arrays of size 1.
(*Inductive step*) Suppose that the algorithm is correct for every input of size k, and suppose that `data` has size $n = k + 1$. Note that the steps the algorithm takes for `data` are the same as those taken to solve `data[1..k]`, with one additional iteration of the `for` loop. So, by the inductive hypothesis, `data[min]` must be the minimum of `data[1..k]` after the first $k - 1$ iterations of the `for` loop. On the k^{th} iteration of the `for` loop, `min` becomes `k+1` if `data[k+1]` $<$ `data[min]`. If so, `data[min]` must be the minimum of the entire array, as `data[k+1]` is less than the minimum of `data[1..k]`. Otherwise, `data[min]` $\leq$ `data[k+1]`, so `data[min]` is the minimum of `data[1..k+1]`. As this conclusion holds in both cases, the algorithm is correct for an array of size $n = k + 1$. Therefore, by induction, the algorithm is correct for arrays of any size ≥ 1. $\square$

Another example

- **Algorithm:** Insertion 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:

```
if n > 1
  Call Insertion Sort on data[1..n-1]
  Let ins = data[n]
  Let j = last index of data[1..n-1] less than or equal to ins
  Shift data[j+1..n-1] to the right one space
  data[j+1] = ins
end
return data
```

[*] Modified to be recursive

Correctness exercise

- Prove that this algorithm correctly sorts its input array.

Proof. We prove the claim by induction on n.
(*Base case*) When $n = 1$, `data` has one element, so it is already sorted, and Insertion Sort returns the array.
(*Inductive step*) Suppose that Insertion Sort correctly sorts every array of size k, and suppose `data` has size $n = k + 1$. Since $k + 1 > 1$, Insertion Sort will pass the *if* condition. The first line of this block calls Insertion Sort on `data[1..k]`, which will be sorted correctly by the Induction Hypothesis, as it is an array of size k.

$\square$

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

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
  for j = 1 to n-i step i
    if data[j] > data[j+i]
      Swap data[j] and data[j+i]
    end
  end
end
```

Incorrectness example solution

Proof. We prove that BadSort is incorrect by counterexample. Consider the input data = (1, 3, 5, 2, 4) and n = 5.
On the first iteration of the outer *for* loop, i = 4, and BadSort will compare 1 to 4 but will not swap them.
On the second iteration, i = 3, and BadSort will compare 1 to 2 but will not swap them.
On the third iteration, i = 2. BadSort will compare 1 to 5 and not swap them, then it will compare 5 to 4 and swap them, leaving (1, 3, 4, 2, 5).
On the fourth and final iteration, i = 1. BadSort will compare 1 to 3 and 3 to 5 but not swap them. It will compare 4 to 2 and swap them, leaving (1, 3, 2, 4, 5). Finally, it will compare 4 to 5 and terminate, returning (1, 3, 2, 4, 5).
However, this result is incorrect, as 3 > 2. Therefore, BadSort is not a correct sorting algorithm. □

Coming up

- **Homework 2** is posted on Canvas
  - Due next Thursday
- **Homework 1** is due Tuesday
- **Feedback form 1** is due at Exam 1

- **Recommended readings:** Chapter 1
- **Practice problems** (not required): solve 1-2 "Interview Problems" from Chapter 1 (p. 30)