Lecture 11: Proving Correctness

CS 5002: Discrete Math

Adrienne Slaughter, Tamara Bonaci

Northeastern University

December 9, 2018

1 Mergesort Analysis

2 Proof Techniques
- Proof by Counterexample
- Proof by Induction
  - Mathematical Induction
  - Building block: The Well-Ordering Property
  - Applying Mathematical Induction to Algorithms
- Proof by Loop Invariant Examples

3 Summary

Mergesort: Analysis

We do 2 things in our analysis:

- What's the runtime?
- Is it correct?

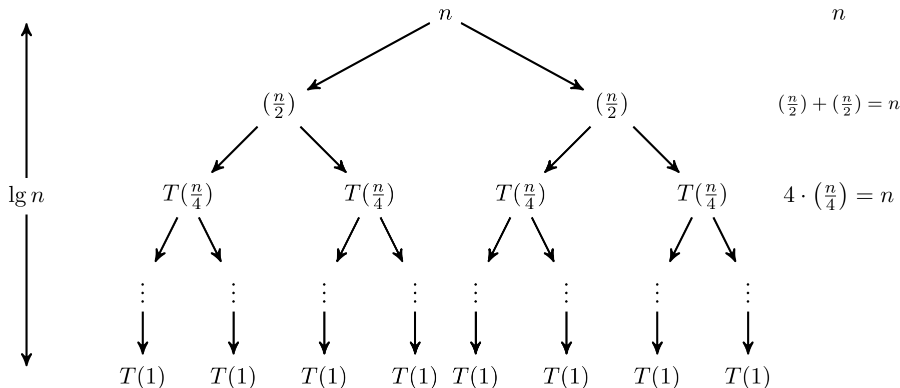
Mergesort: What's the runtime?

Recall the runtime of Mergesort:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 2T(\frac{n}{2}) + \Theta(n) & \text{otherwise} \end{cases}$$

Example: Recursion Tree

$$T(n) = 2T(n/2) + n$$



Mergesort: Runtime Summary

Mergesort: Correctness Proof

Agenda

- Review of Mergesort
- Ways to prove algorithms correct
  - Counterexample
  - Induction
  - Loop Invariant
- Proving Mergesort correct
- Other types of proofs
  - Contradiction
  - Cases
  - Contrapositive
  - Chain of iffs

Section 2

Proof Techniques

Proving Correctness

How to prove that an algorithm is correct?

Proving Correctness

How to prove that an algorithm is correct?

Proof by:

- Counterexample (*indirect proof*)
- Induction (*direct proof*)
- Loop Invariant

Other approaches: proof by cases/enumeration, proof by chain of iffs, proof by contradiction, proof by contrapositive

Proof by Counterexample

Searching for counterexamples is the best way to disprove the correctness of some things.

- Identify a case for which something is NOT true
- If the proof seems hard or tricky, sometimes a counterexample works
- Sometimes a counterexample is just easy to see, and can shortcut a proof
- If a counterexample is hard to find, a proof might be easier

Proof by Induction

Failure to find a counterexample to a given algorithm does not mean "it is obvious" that the algorithm is correct.

Mathematical induction is a very useful method for proving the correctness of recursive algorithms.

1. Prove base case
2. Assume true for arbitrary value n
3. Prove true for case $n + 1$

Proof by Loop Invariant

- Built off proof by induction.
- Useful for algorithms that loop.

Formally: find loop invariant, then prove:

1. Define a Loop Invariant
2. Initialization
3. Maintenance
4. Termination

Informally:

1. Find p, a loop invariant
2. Show the base case for p
3. Use induction to show the rest.

1 Mergesort Analysis

2 Proof Techniques
- Proof by Counterexample
- Proof by Induction
  - Mathematical Induction
  - Building block: The Well-Ordering Property
  - Applying Mathematical Induction to Algorithms
- Proof by Loop Invariant Examples

3 Summary

Proof by Counterexample

Used to prove statements false, or algorithms either incorrect or non-optimal

Counterexample: Examples

- **Prove or disprove:** $\lceil x + y \rceil = \lceil x \rceil + \lceil y \rceil$.

Counterexample: Examples

- **Prove or disprove:** $\lceil x + y \rceil = \lceil x \rceil + \lceil y \rceil$.
  - Proof by counterexample: $x = \frac{1}{2}$ and $y = \frac{1}{2}$

Counterexample: Examples

- **Prove or disprove:** $\lceil x + y \rceil = \lceil x \rceil + \lceil y \rceil$.
  - Proof by counterexample: $x = \frac{1}{2}$ and $y = \frac{1}{2}$
- **Prove or disprove:** "Every positive integer is the sum of two squares of integers"

Counterexample: Examples

- **Prove or disprove:** $\lceil x + y \rceil = \lceil x \rceil + \lceil y \rceil$.
  - Proof by counterexample: $x = \frac{1}{2}$ and $y = \frac{1}{2}$
- **Prove or disprove:** "Every positive integer is the sum of two squares of integers"
  - Proof by counterexample: 3

Counterexample: Examples

- **Prove or disprove:** $\lceil x + y \rceil = \lceil x \rceil + \lceil y \rceil$.
  - Proof by counterexample: $x = \frac{1}{2}$ and $y = \frac{1}{2}$
- **Prove or disprove:** "Every positive integer is the sum of two squares of integers"
  - Proof by counterexample: 3
- **Prove or disprove:** $\forall x \forall y (xy \geq x)$ (over all integers)

Counterexample: Examples

- **Prove or disprove:** $\lceil x + y \rceil = \lceil x \rceil + \lceil y \rceil$.
  - Proof by counterexample: $x = \frac{1}{2}$ and $y = \frac{1}{2}$
- **Prove or disprove:** "Every positive integer is the sum of two squares of integers"
  - Proof by counterexample: 3
- **Prove or disprove:** $\forall x \forall y (xy \geq x)$ (over all integers)
  - Proof by counterexample: $x = -1, y = 3; xy = -3; -3 \not\geq -1$

Example: Greedy Algorithms

Greedy Algorithm: An algorithm that selects the best choice at each step, instead of considering all sequences of steps that may lead to an optimal solution.

- It's usually straight-forward to find a greedy algorithm that is ***feasible***, but hard to find a greedy algorithm that is ***optimal***
- Either prove the solution optimal, or find a counterexample such that the algorithm yields a non-optimal solution
- An algorithm can be greedy even if it doesn't produce an optimal solution

Example: Interval Scheduling

The Problem: We have a resource r, such as a classroom, and a bunch of requests $q : \{start, finish\}$. How can we schedule the requests to use the resource?

Example: Interval Scheduling

The Problem: We have a resource r, such as a classroom, and a bunch of requests $q : \{start, finish\}$. How can we schedule the requests to use the resource?

- We want to identify a set S of requests such that no requests overlap.

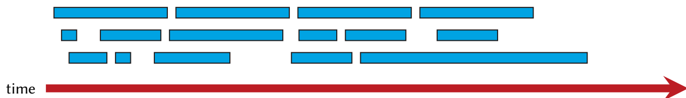
Example: Interval Scheduling

The Problem: We have a resource r, such as a classroom, and a bunch of requests $q : \{start, finish\}$. How can we schedule the requests to use the resource?

- We want to identify a set S of requests such that no requests overlap.
- Ideally, the S that we find contains the maximum number of requests.

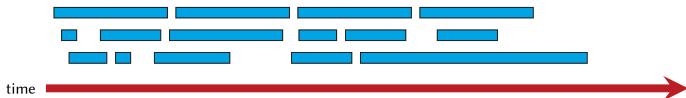
In this diagram, we see three sets of requests.

Which set of requests is the preferred choice for the interval scheduling problem as defined?



In this diagram, we see three sets of requests.

Which set of requests is the preferred choice for the interval scheduling problem as defined?



Solution: A simple heuristic that is an example of a ***greedy algorithm***.

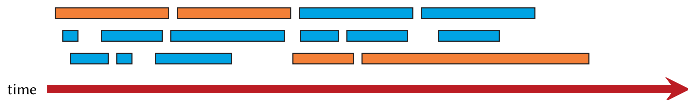
Interval Scheduling: Simple Greedy Algorithm

Input: Array A of requests $q : \{start, finish\}$ such that
$(q_1 = \{s_1, f_1\}, q_2 = \{s_2, f_2\}, \ldots q_n = \{s_n, f_n\})$
Output: S is the set of talks scheduled

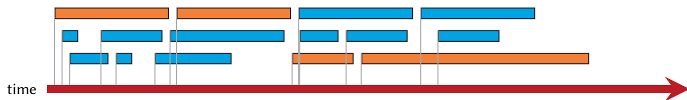
Schedule(A):

1 Sort talks by start time; reorder so that $s_1 \leq s_2 \leq \ldots \leq s_n$
2 $S = \emptyset$
3 **for** $j = 1$ to n:
4   **if** q_j is compatible with S:
5 **return** S

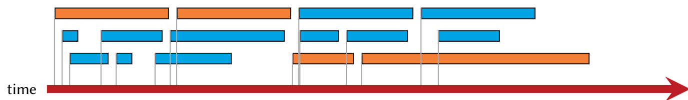
Interval Scheduling: Visually

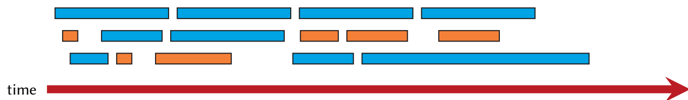


Interval Scheduling: Visually

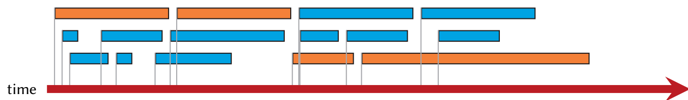


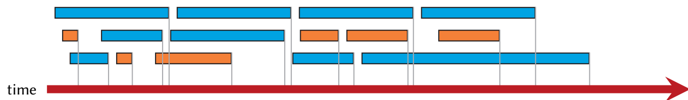
Interval Scheduling: Visually





Interval Scheduling: Visually





Interval Scheduling: Correct Greedy Algorithm

Input: Array A of requests $q : \{start, finish\}$ such that
$(q_1 = \{s_1, f_1\}, q_2 = \{s_2, f_2\}, \ldots q_n = \{s_n, f_n\})$
Output: S is the set of talks scheduled

Schedule(A):

1 Sort talks by finish time; reorder so that $f_1 \leq f_2 \leq \ldots \leq f_n$
2 $S = \emptyset$
3 **for** $j = 1$ to n:
4 **if** q_j is compatible with S:
5 ▷ The current request doesn't conflict with any others we've chosen
6 $S = S \cup q_j$ ▷ Add it to the set of scheduled
7 **return** S

Interval Scheduling: Proving the simple wrong

- Greedy algorithms are easy to design, but hard to prove correct
- Usually, a counterexample is the best way to do this

Proof by Counterexample

Searching for counterexamples is the best way to disprove the correctness of some things.

- Think about small examples
- Think about examples on or around your decision points
- Think about extreme examples (big or small)

Summary: Counterexamples

- Sometimes it's easy to provide a counterexample
- It's usually enough to provide a counterexample to prove something wrong or False
- In algorithms, particularly useful for proving heuristics or greedy algorithms wrong or non-optimal

1 Mergesort Analysis

2 Proof Techniques
- Proof by Counterexample
- Proof by Induction
  - Mathematical Induction
  - Building block: The Well-Ordering Property
  - Applying Mathematical Induction to Algorithms
- Proof by Loop Invariant Examples

3 Summary

Consider the Equation

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Consider the Equation

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

How do we prove this true?

Consider the Equation: It's true for some numbers...

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Case $n = 1 : \sum_{i=1}^{1} i =$

Consider the Equation: It's true for some numbers...

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\text{Case } n = 1 : \sum_{i=1}^{1} i =$$

$$\text{Case } n = 5 : \sum_{i=1}^{5} i =$$

How do we prove this true?

Consider the Equation: It's true for some numbers...

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\text{Case } n = 1 : \sum_{i=1}^{1} i =$$

$$\text{Case } n = 5 : \sum_{i=1}^{5} i =$$

$$\text{Case } n = 30 : \sum_{i=1}^{30} i =$$

How do we prove this true?

Consider the Equation: It's true for some numbers...

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\text{Case } n = 1 : \sum_{i=1}^{1} i =$$

$$\text{Case } n = 5 : \sum_{i=1}^{5} i =$$

$$\text{Case } n = 30 : \sum_{i=1}^{30} i =$$

How do we prove this true?

Just because we proved this true for a couple of instances doesn't mean we've proved it!

Table of Contents

1 Mergesort Analysis

2 Proof Techniques
- Proof by Counterexample
- Proof by Induction
  - Mathematical Induction
  - Building block: The Well-Ordering Property
  - Applying Mathematical Induction to Algorithms
- Proof by Loop Invariant Examples

3 Summary

Mathematical Induction

1. Prove the formula for the smallest number that can be used in the given statement.
2. Assume it's true for an arbitrary number n.
3. Use the previous steps to prove that it's true for the next number $n + 1$.

Example

A simple example:

Theorem: For all prices $p \geq 8$ cents, the price p can be paid using only 5-cent and 3-cent coins

Step 1: Proving true for smallest number

Show the theorem holds for price $p = 8$ cents.

Step 2: Assume true for arbitrary n

Assume that theorem is true for some $p \geq 8$.

Step 3: Show true for $n+1$

Show the theorem is true for price $p+1$
Inductive step:

- Assume price $p \geq 8$ can be paid using only 3-cent and 5-cent coins.
- Need to prove that price $p+1$ can be paid using only 3-cent and 5-cent coins.

Main idea: “reduce” from price $p+1$ to price p.

Step 3: Show true for $n + 1$

If we have 100 5-cent coins, and 100 3-cent coins (for a total of $p = \$8.00$), how can we modify the number of 5-cent and 3-cent coins so that we can make the $p + 1$ price $(p + 1 = \$8.01)$?

- 40 5-cent coins + 200 3-cent coins ($2.00 + $6.00 = $8.00)
- 39 5-cent coins + 202 3-cent coins ($1.95 + $6.06 = $8.01)
- 99 5-cent coins + 102 3-cent coins ($4.95 + $3.06 = $8.01)

Step 3: Show true for $n + 1$

Assume that $p = 5n + 3m$ where $n, m \geq 0$ are integers.
We need to show that $p + 1 = 5a + 3b$ for integers $a, b \geq 0$. Partition to cases:

- **Case 1:** $n \geq 1$. We have more than 1 5-cent piece.

Step 3: Show true for $n + 1$

Assume that $p = 5n + 3m$ where $n, m \geq 0$ are integers.
We need to show that $p + 1 = 5a + 3b$ for integers $a, b \geq 0$. Partition to cases:

- **Case 1:** $n \geq 1$. We have more than 1 5-cent piece.
  - In this case, $p + 1 = 5 \cdot (n - 1) + 3 \cdot (m + 2)$.

Step 3: Show true for $n+1$

Assume that $p = 5n + 3m$ where $n, m \geq 0$ are integers.
We need to show that $p + 1 = 5a + 3b$ for integers $a, b \geq 0$. Partition to cases:

- **Case 1:** $n \geq 1$. We have more than 1 5-cent piece.
  - In this case, $p + 1 = 5 \cdot (n-1) + 3 \cdot (m+2)$.
  - Remove one 5-cent piece, add 2 3-cent pieces.

Step 3: Show true for $n + 1$

Assume that $p = 5n + 3m$ where $n, m \geq 0$ are integers.
We need to show that $p + 1 = 5a + 3b$ for integers $a, b \geq 0$. Partition to cases:

- **Case 1:** $n \geq 1$. We have more than 1 5-cent piece.
  - In this case, $p + 1 = 5 \cdot (n - 1) + 3 \cdot (m + 2)$.
  - Remove one 5-cent piece, add 2 3-cent pieces.
- **Case 2:** $m \geq 3$. We have more than 3 3-cent pieces.

Step 3: Show true for $n + 1$

Assume that $p = 5n + 3m$ where $n, m \geq 0$ are integers.
We need to show that $p + 1 = 5a + 3b$ for integers $a, b \geq 0$. Partition to cases:

- **Case 1:** $n \geq 1$. We have more than 1 5-cent piece.
  - In this case, $p + 1 = 5 \cdot (n - 1) + 3 \cdot (m + 2)$.
  - Remove one 5-cent piece, add 2 3-cent pieces.
- **Case 2:** $m \geq 3$. We have more than 3 3-cent pieces.
  - $p + 1 = 5 \cdot (n + 2) + 3 \cdot (m - 3)$.

Step 3: Show true for $n+1$

Assume that $p = 5n + 3m$ where $n, m \geq 0$ are integers.
We need to show that $p + 1 = 5a + 3b$ for integers $a, b \geq 0$. Partition to cases:

- **Case 1:** $n \geq 1$. We have more than 1 5-cent piece.
  - In this case, $p + 1 = 5 \cdot (n - 1) + 3 \cdot (m + 2)$.
  - Remove one 5-cent piece, add 2 3-cent pieces.
- **Case 2:** $m \geq 3$. We have more than 3 3-cent pieces.
  - $p + 1 = 5 \cdot (n + 2) + 3 \cdot (m - 3)$.
  - Add 2 5-cent pieces, remove 3 3-cent pieces

Step 3: Show true for $n + 1$

Assume that $p = 5n + 3m$ where $n, m \geq 0$ are integers.
We need to show that $p + 1 = 5a + 3b$ for integers $a, b \geq 0$. Partition to cases:

- **Case 1:** $n \geq 1$. We have more than 1 5-cent piece.
  - In this case, $p + 1 = 5 \cdot (n - 1) + 3 \cdot (m + 2)$.
  - Remove one 5-cent piece, add 2 3-cent pieces.
- **Case 2:** $m \geq 3$. We have more than 3 3-cent pieces.
  - $p + 1 = 5 \cdot (n + 2) + 3 \cdot (m - 3)$.
  - Add 2 5-cent pieces, remove 3 3-cent pieces
- **Case 3:** $n = 0, m \leq 2$. We have no 5-cent pieces, and 2 or fewer 3-cent pieces.

Step 3: Show true for $n + 1$

Assume that $p = 5n + 3m$ where $n, m \geq 0$ are integers.
We need to show that $p + 1 = 5a + 3b$ for integers $a, b \geq 0$. Partition to cases:

- **Case 1:** $n \geq 1$. We have more than 1 5-cent piece.
  - In this case, $p + 1 = 5 \cdot (n - 1) + 3 \cdot (m + 2)$.
  - Remove one 5-cent piece, add 2 3-cent pieces.
- **Case 2:** $m \geq 3$. We have more than 3 3-cent pieces.
  - $p + 1 = 5 \cdot (n + 2) + 3 \cdot (m - 3)$.
  - Add 2 5-cent pieces, remove 3 3-cent pieces
- **Case 3:** $n = 0, m \leq 2$. We have no 5-cent pieces, and 2 or fewer 3-cent pieces.
  - $p = 5n + 3m \leq 6$, which is a contradiction to $p \geq 8$

Just a little thought...

Now that we've proven the theorem, we can use it to derive an algorithm:

The Algorithm

Input: price $p \geq 8$.
Output: integers $n, m \geq 0$ so that $p = 5n + 3m$

PayWithThreeCentsAndFiveCents(p):

1. Let $x = 8, n = 1, m = 1$ (so that $x = 5n + 3m$).
2. **while** $x < p$:
3. $\quad x = x + 1$
4. $\quad$ **if** $n \geq 1$:
5. $\quad\quad n := n - 1$
6. $\quad\quad m := m + 2$
7. $\quad$ **else**
8. $\quad\quad n := n + 2$
9. $\quad\quad m := m - 3$
10. $\quad$ **return** (n, m)

Back to our original proof...

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Step 1: Proving true for smallest number

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\text{Case } n = 1 : \sum_{i=1}^{1} i =$$

Step 2: Assume true for arbitrary n

Assumed.

Proof: Summing n integers

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Proof:

- Does it hold true for $n = 1$?
  $1 = \frac{1(1+1)}{2}$ ✓
- Assume it works for n ✓
- Prove that it's true when n is replaced by $n + 1$

Proof Step 3: Summing n integers

Starting with n:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \quad (1)$$

(6)

Proof Step 3: Summing n integers

Rewriting the left hand side...

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \tag{1}$$

$$1 + 2 + 3 + ...(n-1) + n = \frac{n(n+1)}{2} \tag{2}$$

(6)

Proof Step 3: Summing n integers

Replace n with $n+1$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \tag{1}$$

$$1+2+3+...(n-1)+n = \frac{n(n+1)}{2} \tag{2}$$

$$1+2+3+...+((n+1)-1)+(n+1) = \frac{(n+1)[(n+1)+1]}{2} \tag{3}$$

(6)

Proof Step 3: Summing n integers

Simplifying

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \tag{1}$$

$$1 + 2 + 3 + \ldots (n-1) + n = \frac{n(n+1)}{2} \tag{2}$$

$$1 + 2 + 3 + \ldots + ((n+1) - 1) + (n+1) = \frac{(n+1)[(n+1)+1]}{2} \tag{3}$$

$$1 + 2 + 3 + \ldots + n + (n+1) = \frac{(n+1)(n+2)}{2} \tag{4}$$

(6)

Proof Step 3: Summing n integers

Re-grouping on the left side

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \tag{1}$$

$$1 + 2 + 3 + \ldots (n-1) + n = \frac{n(n+1)}{2} \tag{2}$$

$$1 + 2 + 3 + \ldots + ((n+1) - 1) + (n+1) = \frac{(n+1)[(n+1)+1]}{2} \tag{3}$$

$$1 + 2 + 3 + \ldots + n + (n+1) = \frac{(n+1)(n+2)}{2} \tag{4}$$

$$(1 + 2 + 3 + \ldots + n) + (n+1) = \frac{(n+1)(n+2)}{2} \tag{5}$$

$$\tag{6}$$

Proof Step 3: Summing n integers

Replace our known (assumed) formula from #2

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \tag{1}$$

$$1 + 2 + 3 + ...(n-1) + n = \frac{n(n+1)}{2} \tag{2}$$

$$1 + 2 + 3 + ... + ((n+1) - 1) + (n+1) = \frac{(n+1)[(n+1)+1]}{2} \tag{3}$$

$$1 + 2 + 3 + ... + n + (n+1) = \frac{(n+1)(n+2)}{2} \tag{4}$$

$$(1 + 2 + 3 + ... + n) + (n+1) = \frac{(n+1)(n+2)}{2} \tag{5}$$

$$\frac{n(n+1)}{2} + (n+1) = \frac{(n+1)(n+2)}{2} \tag{6}$$

Proof Step 3: Summing n integers (pt 2)

Established a common denominator

$$\frac{n(n+1)}{2} + \frac{2(n+1)}{2} = \frac{(n+1)(n+2)}{2} \tag{7}$$

(9)

Proof Step 3: Summing n integers (pt 2)

Simplify

$$\frac{n(n+1)}{2} + \frac{2(n+1)}{2} = \frac{(n+1)(n+2)}{2} \quad (7)$$

$$\frac{n(n+1) + 2(n+1)}{2} = \frac{(n+1)(n+2)}{2} \quad (8)$$

$$(9)$$

Proof Step 3: Summing n integers (pt 2)

Factor out common factor $n+1$

$$\frac{n(n+1)}{2} + \frac{2(n+1)}{2} = \frac{(n+1)(n+2)}{2} \tag{7}$$

$$\frac{n(n+1) + 2(n+1)}{2} = \frac{(n+1)(n+2)}{2} \tag{8}$$

$$\frac{(n+1)(n+2)}{2} = \frac{(n+1)(n+2)}{2} \tag{9}$$

Proof Step 3: Summing n integers (pt 2)

$$\frac{n(n+1)}{2} + \frac{2(n+1)}{2} = \frac{(n+1)(n+2)}{2} \tag{7}$$

$$\frac{n(n+1) + 2(n+1)}{2} = \frac{(n+1)(n+2)}{2} \tag{8}$$

$$\frac{(n+1)(n+2)}{2} = \frac{(n+1)(n+2)}{2} \quad \checkmark \tag{9}$$

Proof Step 3: Summing n integers (pt 2)

$$\frac{n(n+1)}{2} + \frac{2(n+1)}{2} = \frac{(n+1)(n+2)}{2} \tag{7}$$

$$\frac{n(n+1) + 2(n+1)}{2} = \frac{(n+1)(n+2)}{2} \tag{8}$$

$$\frac{(n+1)(n+2)}{2} = \frac{(n+1)(n+2)}{2} \quad \checkmark \tag{9}$$

We've proved that the formula holds for $n+1$.

Proof: Summing n integers

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Proof:

- Does it hold true for $n = 1$?
  $1 = \frac{1(1+1)}{2}$ ✓
- Assume it works for n ✓
- Prove that it's true when n is replaced by $n + 1$ ✓

Mathematical Induction

- Prove the formula for a base case
- Assume it's true for an arbitrary number n
- Use the previous steps to prove that it's true for the next number $n + 1$

Table of Content

1. Mergesort Analysis
2. Proof Techniques
   - Proof by Counterexample
   - Proof by Induction
     - Mathematical Induction
     - Building block: The Well-Ordering Property
     - Applying Mathematical Induction to Algorithms
   - Proof by Loop Invariant Examples
3. Summary

The Well-Ordering Property

The Well-Ordering property

The positive integers are *well-ordered*. An ordered set is ***well-ordered*** if each and every nonempty subset has a smallest or least element.
Every nonempty subset of the positive integers has a least element.

Note: this property is not true for the set of integers (in which there are arbitrarily small negative numbers) or subsets of, e.g., the positive real numbers (in which there are elements arbitrarily close to zero).

©Northeastern University

The Well-Ordering Principle

An equivalent statement to the well-ordering principle is as follows:
The set of positive integers does not contain any infinite strictly decreasing sequences.

Proving Well-Ordered Principle with Induction[1]

Let S be a subset of the positive integers with no least element.

[1]adapted from: https://brilliant.org/wiki/the-well-ordering-principle/

Proving Induction with the Well-Ordered Principle

Suppose P is a property of an integer such that $P(1)$ is true, and $P(n)$ being true implies that $P(n+1)$ is true.

Template for proofs based on Well-Ordering

The Well-Ordering Principle can be used for proofs. A template:

To prove that "$P(n)$ is true for all $n \in \mathbb{N}$" using the Well Ordering Principle:

- Define the set C of counterexamples to P being true. Specifically, define $C ::== \{n \in \mathbb{N} | \text{ NOT } P(n) \text{ is true}\}$
  - (The notation $\{n|Q(n)\}$ means "the set of all elements n for which $Q(n) is true.$")
- Assume for proof by contradiction that C is nonempty.
- By WOP, there will be a smallest element n in C.
- Reach a contradiction somehow-often by showing that $P(n)$ is actually true or by showing that there is another member of C that is smaller than n. This is the open-ended part of the proof task.
- Conclude that C must be empty, that is, no counterexamples exist.

Summary: Well-Ordering Property

- Let's us order things
- Basis for proving that induction works
- Good to know about it; delve into more details on your own.

1 Mergesort Analysis

2 Proof Techniques
- Proof by Counterexample
- Proof by Induction
  - Mathematical Induction
  - Building block: The Well-Ordering Property
  - Applying Mathematical Induction to Algorithms
- Proof by Loop Invariant Examples

3 Summary

Mathematical Induction to Algorithmic Induction

- We've seen an example of mathematical induction
- We generated an algorithm from our proof
- We saw another example of mathematical induction
- Time to see another algorithm proof

Insert Algorithm

Insert(A, e)

```
1   Add(A, e)                          ▷ Add e at the end of A
2   for i = A.length − 1 to 1:
3       while A[i + 1] < A[i]:
4           A[i + 1] = A[i]            ▷ Move the larger one to the end
```

We want to prove that for any element e and any list A:

1. The resulting list after $\text{INSERT}(A, e)$ is sorted
2. The resulting list after $\text{INSERT}(A, e)$ contains all of the elements of A, plus element e.

Proving: Let's say $P(n)$ is defined for any list A and element e as:

- If $\text{SORTED}(A)$ and $\text{LENGTH}(A) = n$ then $\text{SORTED}(\text{INSERT}(A, l))$ and $\text{ELEMENTS}(\text{INSERT}(A, e)) = \text{ELEMENTS}(A) \cup \{e\}$

We want to prove that $P(n)$ holds for all $n \geq 0$.

Proving Insertion Sort: Base Case

$n = 0$: Prove that $P(0)$ holds.

- Let a list A such that $\text{SORTED}(A) = True$ and $\text{LENGTH}(A) = 0$.
- The only list with length zero is the empty list, so $A = \emptyset$. Therefore, $\text{INSERT}(A, e)$ evaluates as follows:
  - List $[e]$ has one element, so it is sorted by definition. Hence, $\text{INSERT}(\emptyset, e)$ is sorted.
  - Furthermore, $\text{ELEMENTS}(\text{INSERT}(e, [])) = \text{ELEMENTS}([e]) = \{e\} = \{e\} \cup A = \{e\} \cup \text{ELEMENTS}([])$. Therefore, the base case holds.

Proving Insertion Sort: Assume true for n

a.k.a "The Invisible Step"

- Assume that $P(n)$ holds. That is, for any element e and any sorted list of length n, Insert(A, e) is sorted and contains all of the elements of A, plus e. This is the ***induction hypothesis***.

Proving Insertion Sort: Inductive Step

We want to prove that $P(n+1)$ also holds.

- For any e, and any sorted A of length $n+1$, INSERT(A, e) is also sorted and contains all elements of A, plus e.
  - Let e be an arbitrary element, and A a sorted list of length $n+1$. Let h be the first element of A, and T be the rest of the elements of A, such that h is less than all elements in T, and T is a sorted list of length n. Also, ELEMENTS$(A) =$ ELEMENTS$\{T\} \cup \{h\}$
  - The evaluation of INSERT(A, e) proceeds as follows:

Thanks to the evaluation model by substitution, we have a formal way of describing the execution of insert. According to that model, the evaluation of INSERT(A, e) proceeds as follows:
INSERT(A, e)
-¿ (function evaluation and replacing l with h::t) match h::t with [] -¿ [e] — x::xs -¿ if e ¡ x then n::l else x::(insert(e,xs))
-¿ (pattern matching) if e ¡ h then e::l else h::(insert(e,t))

Case 1: If $e < h$ is true, then $\textsc{Insert}(A, e) = e + A$.

We have the following:

- Since h is less than all elements in T, and $e < h]$, it means that e is less than all elements in $h + T = A$.
- We also know that A is sorted.

Together, the above imply that $e + A$ is sorted. Therefore, $\textsc{Insert}(A, e)$ is sorted.

Also, $\textsc{elements}(\textsc{Insert}(\mathrm{A}, \mathrm{e})) = \textsc{elements}(e + A) = \textsc{elements}(A) \cup \{e\}$. So $P(n+1)$ holds in this case.

Case 2. If $h \leq e$, then $\textsc{Insert}(A, e)$ = $h + \textsc{Insert}(T, e)$
Let $A' = \textsc{Insert}(T, e)$.

- Because T is a sorted list of length n, it means that we can apply the induction hypothesis. By the IH for element e and list T, the list $A' = \textsc{Insert}(T, e)$ is sorted, and $\textsc{elements}(A') = \textsc{elements}(T) \cup e$.
- Since $h + T$ is sorted, h is less than any element in $\textsc{elements}(T)$.
- Further, $h <= e$. Therefore h is less than all elements in A'. Since A' is sorted, $\textsc{Insert}(A, e) = h + A'$ is sorted.
- Finally:

$$
\begin{aligned}
\textsc{elements}(\textsc{insert}(A, e)) &= \textsc{elements}(h + \textsc{Insert}(T, e)) \\
&= \textsc{elements}(h + A') \\
&= \{h\} \cup \textsc{elements}(A') \\
&= \{h\} \cup \{e\} \cup \textsc{elements}(t) \\
&= e \cup h \cup \textsc{elements}(t) \\
&= \{e\} \cup \textsc{elements}(A)
\end{aligned}
$$

- Therefore, $P(n+1)$ holds in this case.

Since the conclusion of $P(n+1)$ holds for all branches of evaluation, we have proved the inductive step.

We can therefore conclude that $P(n)$ holds for all $n \geq 0$.

Summay: Proof by Induction

Induction has three steps:

1. Base case
2. Assume true for n
3. Show true for $n+1$

We:

- Defined proof by induction
- Defined Well-Ordering Property
- Example of mathematical induction
- Example of induction applied to Insertion Sort

1 Mergesort Analysis

2 Proof Techniques
- Proof by Counterexample
- Proof by Induction
  - Mathematical Induction
  - Building block: The Well-Ordering Property
  - Applying Mathematical Induction to Algorithms
- Proof by Loop Invariant Examples

3 Summary

Proof by Loop Invariant Is...

Invariant: something that is always true

After finding a candidate loop invariant, we prove:

1. ***Initialization***: How does the invariant get initialized?

Proof by Loop Invariant Is...

Invariant: something that is always true

After finding a candidate loop invariant, we prove:

1. ***Initialization***: How does the invariant get initialized?
2. ***Loop Maintenance***: How does the invariant change at each pass through the loop?

Proof by Loop Invariant Is...

Invariant: something that is always true

After finding a candidate loop invariant, we prove:

1. ***Initialization***: How does the invariant get initialized?
2. ***Loop Maintenance***: How does the invariant change at each pass through the loop?
3. ***Termination***: Does the loop stop? When?

Loop Invariant Proof Examples

We have a few examples:

- Linear Search
- Insertion Sort
- Bubble Sort
- Merge Sort

LinearSearch(A, v)

```
1   for j = 1 to A.length:
2       if A[j] == v:
3           return j
4       return NIL
```

LinearSearch(A, v)

```
1   for j = 1 to A.length:
2       if A[j] == v:
3           return j
4       return NIL
```

Loop Invariant . At the start of each iteration of the for loop on line 1, the subarray $A[1 : j - 1]$ does not contain the value v

LinearSearch(A, v)

```
1  for j = 1 toA.length:
2      if A[j] == v:
3          return j
4      return NIL
```

Initialization Prior to the first iteration, the array $A[1 : j-1]$ is empty ($j == 1$). That (empty) subarray does not contain the value v.

LinearSearch(A, v)

```
1   for j = 1 to A.length:
2       if A[j] == v:
3           return j
4       return NIL
```

Maintenance Line 2 checks whether $A[j]$ is the desired value (v). If it is, the algorithm will return j, thereby terminating and producing the correct behavior (the index of value v is returned, if v is present). If $A[j] \neq v$, then the loop invariant holds at the end of the loop (the subarray $A[1 : j]$ does not contain the value v).

LinearSearch(A, v)

```
1   for j = 1 to A.length:
2       if A[j] == v:
3           return j
4       return NIL
```

Termination The for loop on line 1 terminates when $j > A.length$ (that is, n). Because each iteration of a for loop increments j by 1, then $j = n + 1$. The loop invariant states that the value is not present in the subarray of $A[1 : j - 1]$. Substituting $n + 1$ for j, we have $A[1 : n]$. Therefore, the value is not present in the original array A and the algorithm returns NIL.

New example: INSERTION SORT

Insertion Sort

InsertionSort(A)

1 **for** $i = 1$ **to** $A.length$
2 $\quad j = i$
3 $\quad$ **while** $j > 0$ and $A[j-1] > A[j]$
4 $\quad\quad$ SWAP$(A[j], A[j-1])$
5 $\quad\quad j = j - 1$

Insertion Sort

InsertionSort(A)

```
1  for i = 1 to A.length
2      j = i
3      while j > 0 and A[j − 1] > A[j]
4          Swap(A[j], A[j − 1])
5          j = j − 1
```

Invariant $A[0 : i - 1]$ are sorted

InsertionSort(A)

```
1  for i = 1 to A.length
2      j = i
3      while j > 0 and A[j - 1] > A[j]
4          Swap(A[j], A[j - 1])
5          j = j - 1
```

Initialization At the top of the first loop, this is $A[0:0]$, which is vacuously true.

Insertion Sort

InsertionSort(A)

```
1  for i = 1 to A.length
2      j = i
3      while j > 0 and A[j - 1] > A[j]
4          Swap(A[j], A[j - 1])
5          j = j - 1
```

Maintenance An inner loop where we start from i and work our way down, swapping values until we find the location for $a[i]$ in the sorted section of the data

Insertion Sort

InsertionSort(A)

1 **for** $i = 1$ **to** $A.length$
2 $\quad j = i$
3 $\quad$ **while** $j > 0$ and $A[j-1] > A[j]$
4 $\quad\quad$ SWAP$(A[j], A[j-1])$
5 $\quad\quad j = j - 1$

Termination: And the end of the **for** loop, $i = len(A)$. That means that the array $A[0 : A.length - 1]$ is now sorted, which is the entire array.

New example: BubbleSort

Bubble Sort: Outer Loop

BubbleSort(A)

```
1   for i = 1 to A.length − 1
2       for j = A.length to i + 1
3           if A[j] < A[j − 1]
4               SWAP(A[j], A[j − 1])
```

Bubble Sort: Outer Loop

BubbleSort(A)

1 **for** $i = 1$ to $A.length - 1$
2   **for** $j = A.length$ **to** $i + 1$
3   **if** $A[j] < A[j-1]$
4   SWAP($A[j]$, $A[j-1]$)

Invariant At the start of each iteration of the **for** loop on line 1, the subarray $A[1 : i-1]$ is sorted

Bubble Sort: Outer Loop

BubbleSort(A)

```
1  for i = 1 to A.length − 1
2      for j = A.length to i + 1
3          if A[j] < A[j − 1]
4              Swap(A[j], A[j − 1])
```

Initialization Prior to the first iteration, the array $A[1 : i-1]$ is empty ($i = 1$). That (empty) subarray is sorted by definition.

Bubble Sort: Outer Loop

BubbleSort(A)

```
1  for i = 1 to A.length − 1
2      for j = A.length to i + 1
3          if A[j] < A[j − 1]
4              SWAP(A[j], A[j − 1]
```

Maintenance Given the guarantees of the inner loop, at the end of each iteration of the **for** loop at line 1, the value at $A[i]$ is the smallest value in the range $A[i : A.range]$. Since the values in $A[i : i - 1]$ were sorted and were less than the value in $A[i]$, the values in the range $A[1 : i]$ are sorted.

Bubble Sort: Outer Loop

BubbleSort(A)

1 **for** $i = 1$ to $A.length - 1$
2   **for** $j = A.length$ **to** $i + 1$
3   **if** $A[j] < A[j-1]$
4   SWAP($A[j], A[j-1]$)

Termination The **for** loop at line 1 ends when i equals $A.length - 1$. Based on the maintenance proof, this means that all values in $A[1 : A.length - 1]$ are sorted and less than the value at $A[length]$. So, by definition, the values in $A[1 : A.length]$ are sorted.

Now we need to do the inner loop.

BubbleSort(A)

```
1  for i = 1 to A.length − 1
2      for j = A.length to i + 1
3          if A[j] < A[j − 1]
4              SWAP(A[j], A[j − 1])
```

Bubble Sort: Inner Loop

BubbleSort(A)

1 **for** $i = 1$ to $A.length - 1$
2   **for** $j = A.length$ **to** $i + 1$
3   **if** $A[j] < A[j - 1]$
4   Swap($A[j], A[j - 1]$)

Invariant At the start of each iteration of the **for** loop on line 2, the value at location $A[j]$ is the smallest value in the subrange from $A[j : A.length]$

Bubble Sort: Inner Loop

BubbleSort(A)

1 **for** $i = 1$ to $A.length - 1$
2 **for** $j = A.length$ **to** $i + 1$
3 **if** $A[j] < A[j-1]$
4 SWAP($A[j], A[j-1]$)

Initialization Prior to the first iteration, $j = A.length$. The subarray $A[j : A.length]$ contains a single value ($A[j]$) and the value at $A[j]$ is (trivially) the smallest value in the range from $A[j : A.length]$)

Bubble Sort: Inner Loop

BubbleSort(A)

```
1  for i = 1 to A.length − 1
2      for j = A.length to i + 1
3          if A[j] < A[j − 1]
4              SWAP(A[j], A[j − 1])
```

Maintenance The **if** statement on line 3 compares the elements at $A[j]$ and $A[j-1]$, swapping $A[j]$ into $A[j-1]$ if it is the lower value and leaving them in place, if not. Given the initial condition that the value in $A[j]$ was the smallest value in the range $A[j : A.length]$, this means the value in $A[j-1]$ is now the smallest value in the range $A[j-1 : A.length]$. This also means that every value in the subarray $A[j : A.length]$ is greater than the value at $A[j-1]$.

Bubble Sort: Inner Loop

BubbleSort(A)

```
1  for i = 1 to A.length − 1
2      for j = A.length to i + 1
3          if A[j] < A[j − 1]
4              SWAP(A[j], A[j − 1])
```

Termination 2 The **for** loop on line 2 terminates when $j = i + 1$ and given the Maintenance property, this means that the value at $A[i]$ (which is $A[j - 1]$) will be the smallest value in the range $A[i : A.range]$ ($A[j - 1 : A.range]$)

Back to proving Mergesort correct

The Merge Algorithm

```
MERGE(A, low, mid, high)
 1  L = A[low:mid] ▷ (L is a new array copied from A[low:mid])
 2  R = A[mid+1, high] ▷ (R is a new array copied from A[mid+1, high])
 3  i = 1, j = 1
 4  for k =low  to high:
 5      if L[i] < R[j]:
 6          A[k] = L[i]
 7          i = i + 1
 8      else
 9          A[k] = R[j]
10          j = j + 1
```

Merge Sort Invariant

Invariant At the start of each **for** loop iteration, the array starting at $A[k]$ with length $k - low$ contains the $k - low$ smallest elements, in increasing sorted order

Merge Sort Invariant

Invariant At the start of each **for** loop iteration, the array starting at $A[k]$ with length $k - low$ contains the $k - low$ smallest elements, in increasing sorted order

Initialization Prior to the first iteration, the array starting at $A[k]$ with length $k - low$ is empty because $k - low = 0$. L and R are assumed sorted.

Merge Sort Invariant

Invariant At the start of each **for** loop iteration, the array starting at $A[k]$ with length $k - low$ contains the $k - low$ smallest elements, in increasing sorted order

Initialization Prior to the first iteration, the array starting at $A[k]$ with length $k - low$ is empty because $k - low = 0$. L and R are assumed sorted.

Maintenance Since L and R are sorted, the value at $L[i]$ is the smallest in L and the value at $R[j]$ is the smallest in R. The smallest of these is the smallest in the union of L and R, which is $A[low : high]$. Copy that into $A[k]$.

Merge Sort Invariant

Invariant At the start of each **for** loop iteration, the array starting at $A[k]$ with length $k - low$ contains the $k - low$ smallest elements, in increasing sorted order

Initialization Prior to the first iteration, the array starting at $A[k]$ with length $k - low$ is empty because $k - low = 0$. L and R are assumed sorted.

Maintenance Since L and R are sorted, the value at $L[i]$ is the smallest in L and the value at $R[j]$ is the smallest in R. The smallest of these is the smallest in the union of L and R, which is $A[low : high]$. Copy that into $A[k]$.

Termination On the last iteration, $k = high + 1$. This means that the array at $A[low]$ with length $k - low(low \cdot high + 1)$ is sorted, which is the array $A[low : high]$. $A[low \ldots high]$ is sorted. $k - low = (high + 1) - low = high - low + 1$

Steps to Loop Invariant Proof

After finding your loop invariant:

- Initialization
  - Prior to the loop initiating, does the property hold?
- Maintenance
  - After each loop iteration, does the property still hold, given the initialization properties?
- Termination
  - After the loop terminates, does the property still hold? And for what data?

©Northeastern University

Things to remember

- ***Algorithm termination*** is necessary for proving correctness of the entire algorithm.
- ***Loop invariant termination*** is necessary for proving the behavior of the given loop.

Summary: Proof by Loop Invariant

Proof by Loop Invariant is based on induction and has 4 steps:

1. Define loop invariant
2. Show initialization
3. Show maintenance
4. Show termination

We:

- Defined proof by loop invariant
- Examples:
  - Linear Search
  - Insertion Sort
  - Bubble Sort
  - Merge Sort

1 Mergesort Analysis

2 Proof Techniques
- Proof by Counterexample
- Proof by Induction
  - Mathematical Induction
  - Building block: The Well-Ordering Property
  - Applying Mathematical Induction to Algorithms
- Proof by Loop Invariant Examples

3 Summary

Why Mergesort matters

- Merge sort used to be king due to media
  - tape drives
- Resurfaced a few years ago
  - magnetic hard drives
- Systems work is about the ratio of available resources
  - memory vs IO
  - CPU vs memory
  - local IO vs network

Good proofs

State your plan A good proof begins by explaining the general line of reasoning, for example, "We use case analysis" or "We argue by contradiction."

Keep a flow Sometimes proofs are written like mathematical mosaics, with juicy tidbits of independent reasoning sprinkled throughout. This is not good. The steps of an argument should follow one another in an intelligible order.

A proof is an essay, not a calculation. Many students initially write proofs the way they compute integrals. The result is a long sequence of expressions without explanation, making it very hard to follow. This is bad. A good proof usually looks like an essay with some equations thrown in. Use complete sentences.

Avoid excessive symbolism. Your reader is probably good at understanding words, but much less skilled at reading arcane mathematical symbols. Use words where you reasonably can.

Revise and simplify. Your readers will be grateful.

Summary

- Approaches to proving algorithms correct
- Counterexamples
  - Helpful for greedy algorithms, heuristics
- Induction
  - Based on mathematical induction
  - Once we prove a theorem, can use it to build an algorithm
- Loop Invariant
  - Based on induction
  - Key is finding an invariant
- Lots of examples

1 Mergesort Analysis

2 Proof Techniques
- Proof by Counterexample
- Proof by Induction
  - Mathematical Induction
  - Building block: The Well-Ordering Property
  - Applying Mathematical Induction to Algorithms
- Proof by Loop Invariant Examples

3 Summary