CS 5002: Discrete Structures **Fall 2018**

Lecture 11: November 20, 2018 [1]

Instructors: Adrienne Slaughter, Tamara Bonaci

Proving Algorithm Correctness

Readings for this week:
Rosen: Chapter 5: Induction and Recursion

Objective: Analyzing Divide and Conquer Algorithms

1. Review of Mergesort
2. Ways to prove algorithms correct
   - Counterexample
   - Induction
   - Loop Invariant
3. Proving Mergesort correct
4. Other types of proofs

11.1 Introduction

Last week, we focused on computing runtime of algorithms, in particular divide-and-conquer and recursive algorithms.

11.1.1 Merge Sort Algorithm

MERGE-SORT$(A, low, high)$

```
1  if (low < high)
2        mid = ⌊(low + high)/2⌋
3        MERGE-SORT(A, low, mid)
4        MERGE-SORT(A, mid+1, high)
5        MERGE(A, low, mid, high)
```

```
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
 4  j = 1
 5  for k =low  to high
 6  if L[i] < R[j]:
 7        A[k] = L[i]
 8        i = i + 1
 9  else
10        A[k] = R[j]
11        j = j + 1
```

11.2 Mergesort Analysis

Mergesort: Analysis

We do 2 things in our analysis:

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

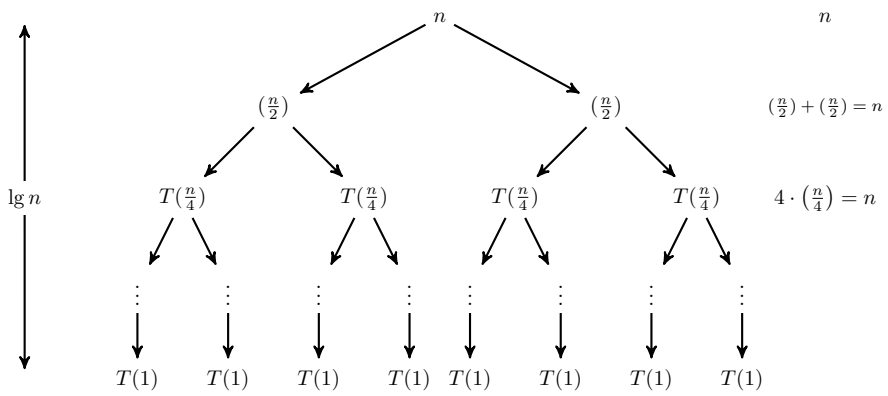
Mergesort: What's the runtime?

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

TODO: put a summary here

11.3 Proof Techniques

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

For any algorithm, we must prove that it always returns the desired output for all legal instances of the problem. For sorting, this means even if the input is already sorted or it contains repeated elements.

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.

11.3.1 Proof by Counterexample

Definition 11.1 (Proof by Counterexample) *Used to prove statements false, or algorithms either incorrect or non-optimal*

Examples: Counterexample

- **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: $x = -1, y = 3; xy = -3; -3 \ngeq -1$

Greedy Algorithms

Definition 11.2 (*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

Interval scheduling is a classic algorithmic problem. In this example, we'll show how we can define a greedy algorithm to solve the problem, and use counterexamples to show a reasonable approach to solving the problem produces a sub-optimal result.

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 the diagram below, we see three sets of requests.

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

For these three sets of requests, S_2 is the set we would consider optimal, because it satisfies the greatest number of requests.

Form of 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 ≤ s_2 ≤ ... ≤ s_n
2  S = ∅
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∪ q_j // Add it to the set of scheduled
7  return S
```

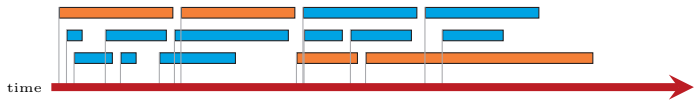
In this algorithm, the in Line 4, "compatible with S" means that q_j does not overlap any of the requests already in S.

Sometimes, this algorithm is written by specifying to put q_j in S, then remove all requests from A that overlap q_j.

Interval Scheduling: Visually

Below is a diagram that represents the a set of requests to be scheduled. This time, ALL requests are in one set, and we are attempting to create an optimal schedule of these requests.

The schedule chosen by our algorithm so far is shown in orange.



The question is: is this an optimal solution? Remember, we said the optimal schedule will satisfy the most requests.

By inspection, I can see that we can choose a different set of requests that do not overlap, and that have more requests satisfied.

When I observe this counterexample, I notice that if I were to order the requests in terms of *finish* time, I will be able to schedule more requests. Upon reflection, this makes sense: By choosing the request that STARTS first, we may choose a very long request that prevents many other requests from being satisfied. But, choosing the request that finishes first leaves more time after that event to schedule more requests.

Interval Scheduling: Correct Greedy Algorithm

Just for reference, here's the correct greedy algorithm. The only difference is that we order the talks by finish time rather than start time.

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 ≤ f_2 ≤ ... ≤ f_n
2  S = ∅
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∪ 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
- Interval scheduling provided an example where it was easy to come up with a simple greedy algorithm.
  - However, we were able to show the algorithm *non-optimal* by using a counterexample.
  - This also depended on our definition of *optimal*. Had we chosen a different definition of optimal (say, utilizing the room the greatest percentage of time), this algorithm may not provide an optimal solution.

Proof by Counterexample

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

Tips for finding counterexamples:

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

11.3.1.1 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

11.3.2 Proof by Induction

This semester, we've used the following identity many times:

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

I waved my hands and said "Right now, we'll just use it as a given. We'll worry about how it's derived later". Well, the time has come.

By now, we know that it certainly works for some numbers, based on our experience:

$$\text{Case } n = 1 : \sum_{i=1}^{1} i = \frac{1(1+1)}{2} \Rightarrow 1 = 1$$

$$\text{Case } n = 5 : \sum_{i=1}^{5} i = \frac{5(5+1)}{2} = 1 + 2 + 3 + 4 + 5 = 15 \Rightarrow 15 \leq 15$$

$$\text{Case } n = 30 : \sum_{i=1}^{30} i = \frac{30(30+1)}{2} \Rightarrow 465 = 465 \text{ Check my math on your own!}$$

But, just because we proved this true for a couple of instances doesn't mean we've proved it is true for all n!

11.3.2.1 Mathematical Induction

Definition 11.3 (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.

Our goal: "reduce" from price $p + 1$ to price p (or, the other way around— basically, show how to add or subtract 1 cent from p).

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 ($\$4.95 + \$3.06 = \$8.01$)

Assume that $p = 5n + 3m$ where $n, m \geq 0$ are integers.

We need to show that $p+1 = 5a+3b$ for integers $a, b \geq 0$.

To do this, we need to consider that we have 3 different cases, and show that it works for each case:

- **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 $p \geq 8$

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

An 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      x = x + 1
4      if n ≥ 1:
5          n := n − 1
6          m := m + 2
7      else
8          n := n + 2
9          m := m − 3
10     return (n, m)
```

Back to our original proof...

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \tag{11.1}$$

Step 1: Proving true for smallest number

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 $n + 1$

Starting with n:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \tag{11.2}$$

$$1 + 2 + 3 + ...(n-1) + n = \frac{n(n+1)}{2} \quad \text{Rewriting the left hand side...} \tag{11.3}$$

$$1 + 2 + 3 + ... + ((n+1) - 1) + (n+1) = \frac{(n+1)[(n+1)+1]}{2} \quad \text{Replace } n \text{ with } n+1 \tag{11.4}$$

$$1 + 2 + 3 + ... + n + (n+1) = \frac{(n+1)(n+2)}{2} \quad \text{Simplifying} \tag{11.5}$$

$$(1 + 2 + 3 + ... + n) + (n+1) = \frac{(n+1)(n+2)}{2} \quad \text{Re-grouping on the left side} \tag{11.6}$$

$$\frac{n(n+1)}{2} + (n+1) = \frac{(n+1)(n+2)}{2} \quad \text{Replace our known (assumed) formula from \#2} \tag{11.7}$$

$$\frac{n(n+1)}{2} + \frac{2(n+1)}{2} = \frac{(n+1)(n+2)}{2} \quad \text{Established a common denominator} \tag{11.8}$$

$$\frac{n(n+1) + 2(n+1)}{2} = \frac{(n+1)(n+2)}{2} \quad \text{Simplify} \tag{11.9}$$

$$\frac{(n+1)(n+2)}{2} = \frac{(n+1)(n+2)}{2} \quad \text{Factor out common factor } n+1 \tag{11.10}$$

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

11.3.2.2 Proof Summary: 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 $n + 1$

11.3.2.3 Building block: The Well-Ordering Property

The Well-Ordering Property

Definition 11.4 *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).

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[2]

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

1. Let S be a subset of the positive integers with no least element.
2. Clearly $1 \notin S$ since it would be the least element if it were.
3. Let T be the complement of S, so $1 \in T$.
4. Now suppose every positive integer $\leq n$ is in T. Then if $n+1 \in S$ it would be the least element of S since every integer smaller than $n+1$ is in the complement of S.
5. This is not possible, so $n+1 \in T$ instead.
6. This implies that every positive integer is in T by strong induction. Therefore, S is the empty set.

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.

1. 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.
2. Let S be the set of integers k such that $P(k)$ is false.
3. Suppose S is nonempty and let k be its least element.
4. Since $P(1)$ is true1 $\notin S$ so $k \neq 1$ so $k-1$ is a positive integer, and by minimality $k-1 \notin S$.
5. So by definition $P(k-1)$ is true, but then by the property of P given above, $P(k-1)$ being true implies that $P(k)$ is true.
6. So $k \notin S$; contradiction.
7. So S is empty; so $P(k)$ is true for all k.

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 $C ::== \{n \in \mathbb{N} |$ NOT $P(n)$ is true$\}$
  - (The notation $\{n|Q(n)\}$ means "the set of all elements n for which $Q(n) istrue$.")

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

- 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.

11.3.2.4 Applying Mathematical Induction to Algorithms

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 $\text{ELEMENTS}(\text{INSERT}(A, e)) = \text{ELEMENTS}(A) \cup \{e\}$

$\text{SORTED}(A)$ indicates the list is sorted.

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

The proof is by induction on length of list A. The proof follows the usual format of a proof by induction: specifying what property we want to prove, what we are inducting on, showing the base case, the inductive step, and clearly specifying when we apply the induction hypothesis (carefully indicating why we can apply it, and showing the lists to which it is applied!).

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, $\textsc{Insert}(A, e)$ evaluates as follows:
  - List $[e]$ has one element, so it is sorted by definition. Hence, $\textsc{Insert}(\emptyset, e)$ is sorted.
  - Furthermore, $\textsc{elements}(\textsc{insert}(e, [])) = \textsc{elements}([e]) = \{e\} = \{e\} \cup A = \{e\} \cup \textsc{elements}([])$. 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, $\textsc{Insert}(A, e)$ is sorted and contains all of the elements of A, plus e. This is the ***induction hypothesis***.

Proving Insertion Sort: Inductive Step

Prove that $P(n+1)$ also holds.

For any e, and any sorted A of length $n+1$, $\textsc{Insert}(A, e)$ is also sorted and contains all elements of A, plus e.

- Symbols we'll use:
  - Let e be an arbitrary element
  - A a sorted list of length $n+1$
  - Let h be the first element of A
  - Let 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.
  - $\textsc{elements}(A) = \textsc{elements}(T) \cup \{h\}$
- The evaluation of $\textsc{Insert}(A, e)$ proceeds as follows:

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, $\textsc{elements}(\textsc{Insert}(\text{A, e})) = \textsc{elements}(e + A) = \textsc{elements}(A) \cup \{e\}$. 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

11.3.3 Proof by Loop Invariant Examples

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 toA.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

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.

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).

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.

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

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

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

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.

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

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

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.

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.

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 $A[j : A.length]$

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 $A[j : A.length]$)

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 $A[j-1]$.

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 $A[i : A.range]$ ($A[j-1 : A.range]$)

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

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?

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

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

11.3.4 Good proofs

[3]

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.

Introduce notation thoughtfully. Sometimes an argument can be greatly simplified by introducing a variable, devising a special notation, or defining a new term. But do this sparingly, since you're requiring the reader to remember all that new stuff. And remember to actually define the meanings of new variables, terms, or notations; don't just start using them!

[3] From *Mathematics for Computer Science*, Lehman, Leighton & Meyer, 2018.

Structure long proofs. Long programs are usually broken into a hierarchy of smaller procedures. Long proofs are much the same. When your proof needed facts that are easily stated, but not readily proved, those fact are best pulled out as preliminary lemmas. Also, if you are repeating essentially the same argument over and over, try to capture that argument in a general lemma, which you can cite repeatedly instead.

Be wary of the "obvious." When familiar or truly obvious facts are needed in a proof, it's OK to label them as such and to not prove them. But remember that what's obvious to you may not be— and typically is not—obvious to your reader.

Most especially, don't use phrases like "clearly" or "obviously" in an attempt to bully the reader into accepting something you're having trouble proving. Also, go on the alert whenever you see one of these phrases in someone else's proof.

Finish. At some point in a proof, you'll have established all the essential facts you need. Resist the temptation to quit and leave the reader to draw the "obvious" conclusion. Instead, tie everything together yourself and explain why the original claim follows.

11.4 Summary

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

Readings for next week:
Rosen, Chapter 10: Graphs. 10.1, 10.2, 10.3, 10.4, 10.6

11.5 Appendix

11.6 Recursion vs Divide and Conquer

What is recursion?

- Define a function in terms of itself
- The call needs to ensure that parameters change/make progress toward the base case.
- Need to ensure the function returns on the base case

Recursive procedure: Example

Factorial: $n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$

```
1
2   int factorial(n){
3     if (n=1):
4       return 1;
5     return n * factorial(n);
6   }
7
8
```

Recursive vs Iterative

Factorial: $n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$

```
1
2   int factorial(n){
3     int out = 1;
4     for (int i=1; i<= n; i++){
5       out *= i;
6     }
7     return out;
8   }
9
10
11
```

Similarities between Recursion and Divide and Conquer