↑ Note incorrect time is displayed in left corner

CSCI 3104: Algorithms

Lecture 1: Introduction, Insertion Sort, And Loop Invariants

Rachel Cox

Department of Computer Science

Lecture begins at 9:35 am

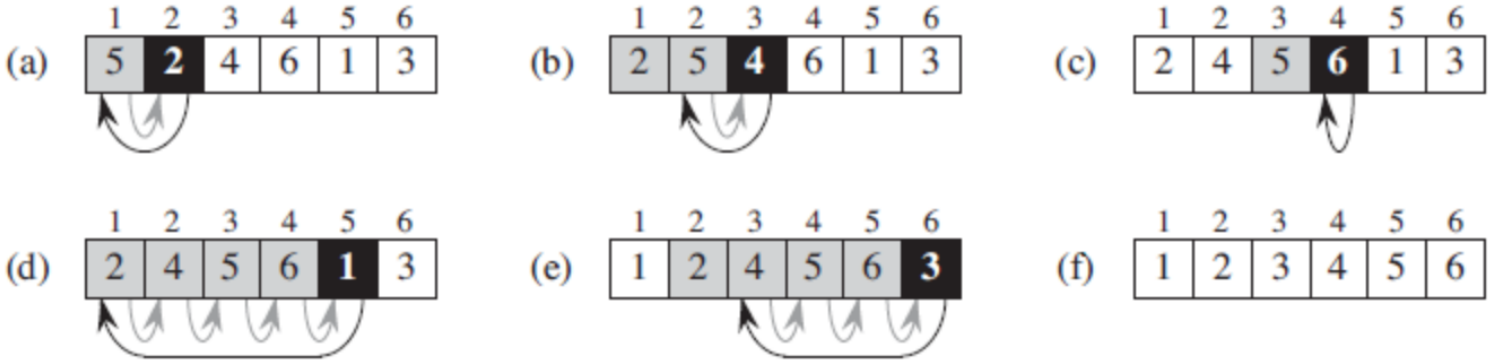


Image source: "Introduction to Algorithms", The MIT Press

Introduction & Logistics

Welcome to CSCI 3104 - Algorithms!

- Course Platforms
- Textbooks/Resources
- Grading
- Assignments & Exams
- Recitations
- Course Schedule
- Course Syllabus
- Brief Intro to Algorithms

Introduction & Logistics

Your Instructor: Rachel Cox

Prior Courses:
CSCI 2824 - Discrete Structures
CSCI 3022 - Intro to Data Science
CSCI 3202 - Intro to Artificial Intelligence
APPM 1235 - Precalculus
APPM 1350 - Calculus 1
APPM 1360 - Calculus 2

- ❑ Graduate degrees from Florida State University and CU Boulder

- ❑ Undergrad at Bucknell University

Introduction & Logistics

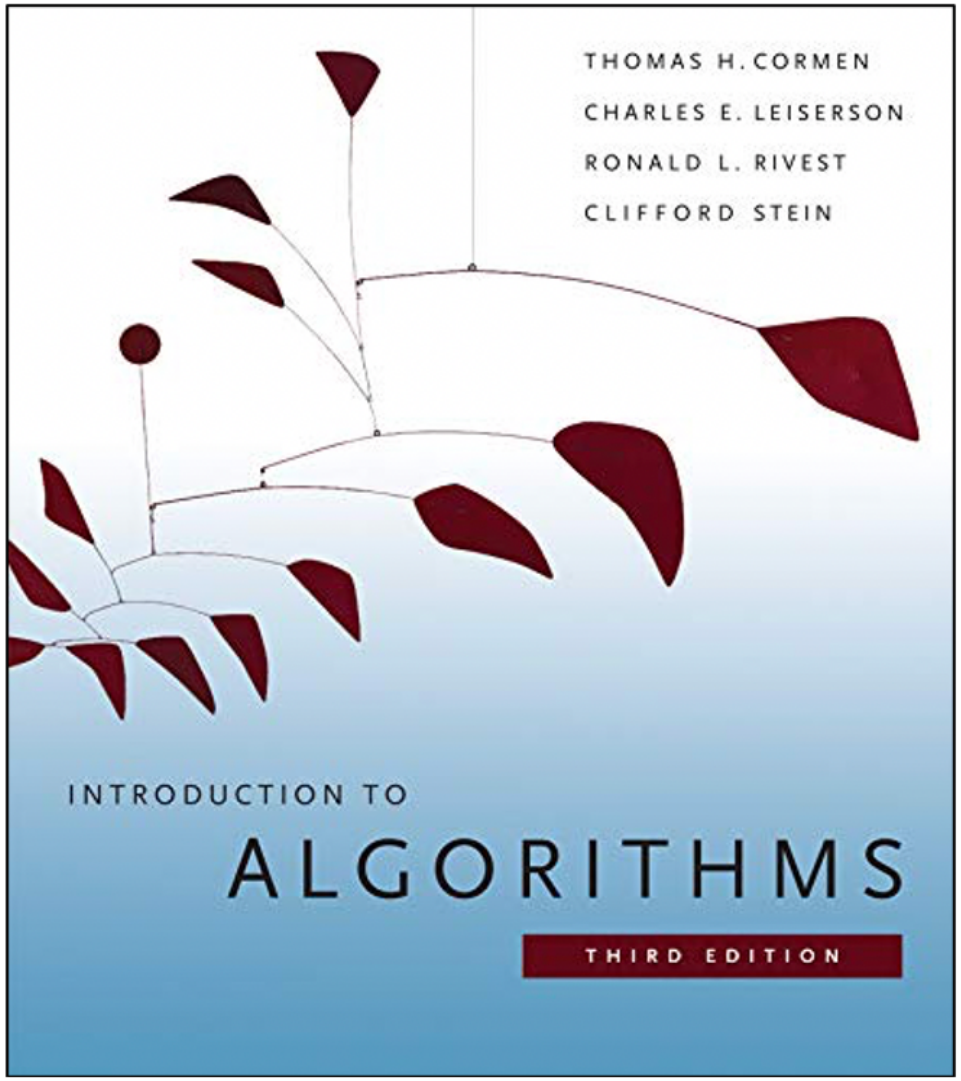
Course Platforms:

Canvas - All links to Zoom Lectures, HW assignments, Quizzes, Grades

Piazza - Q&A from classmates, TAs, CAs, and Instructors

Gradescope - Submission of HW and Exams

Introduction & Logistics



Introduction & Logistics

Lectures will be prerecorded.
- Posted by Monday/Wednesday night

Weekly Homework:
- Due Fridays at 5pm.
- Worth 30% of the total grade.
- You have 2 emergency drops.
- Late policy: Turn in up to 24 hours late for 90% credit, turn in up to 48 hours late for 80% credit.

} 1st HW due Jan 22

Weekly Quizzes:
- Open on Mondays, Due on Wednesdays by 11:59pm
- Worth 5% of your total grade.

1st one due Jan 20

Thursdays at 9:35am - live problem session

Participation:
- Weekly post on Piazza.
- Weekly attendance in recitation.
- Worth 10% of your total grade. We will grade you out of 29 points for this category. (Each of the 14 recitations and each of the 15 posts on Piazza count as a point.)

Midterm Exams:
- 2 Midterms: February 24-25, April 7-8
- Worth 20% each

Final Exam:
- Section 100: May 3
- Worth 15% of your grade.

Recitations
- make sure to attend your scheduled recitation

Introduction & Logistics

Cheating:

- Cheating includes but is not limited to copying from the web, copying from other students (current or former), and using google while taking an exam or quiz.

- If you are caught cheating, you will more likely be automatically dropped from the course and charged with academic dishonesty.

Introduction & Logistics

Recitations:

Attendance will be recorded

- This week's recitation focuses on loop invariants, a review on induction, and good proof writing techniques.

- It is important to attend your recitation because they are not recorded but we expect you to know what is covered in them.

- The goal in each recitation is to see worked examples and practice problems.

What will we learn today?

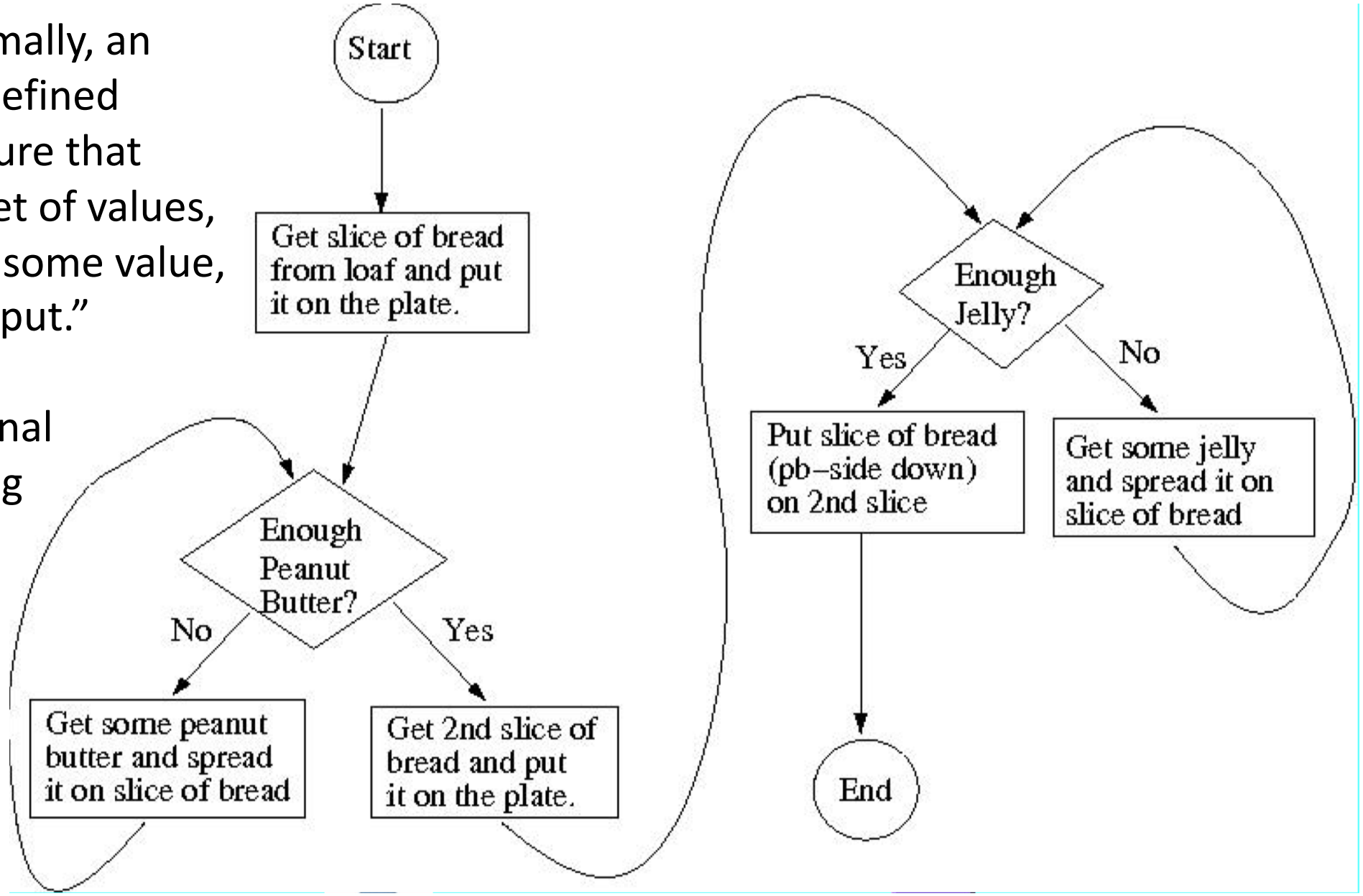
- ❑ Intro to Algorithms
- ❑ Examples of Algorithms
- ❑ Insertion Sort

Textbook CLRS Chapter 1

What is an Algorithm?

From our book: “Informally, an algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.”

“A specific computational procedure for achieving that input/output relationship."



What is an Algorithm?

A few examples:

Source

Babylonian algorithms ~1600 BC
- First known algorithms, written on clay tablets

Euclidean algorithm ~ 300 BC
- Method to find the greatest common divisor

Google's ranking algorithm, PageRank 1996
- Likely the most used algorithm

Source: *PhotoMIX-Company/Pixabay*

Quicksort 1962
- Efficiently sorting lists alphabetically and numerically

JPEG, data compression algorithms 1992
- Make computer systems more efficient

What is an Algorithm?

Key Concepts:

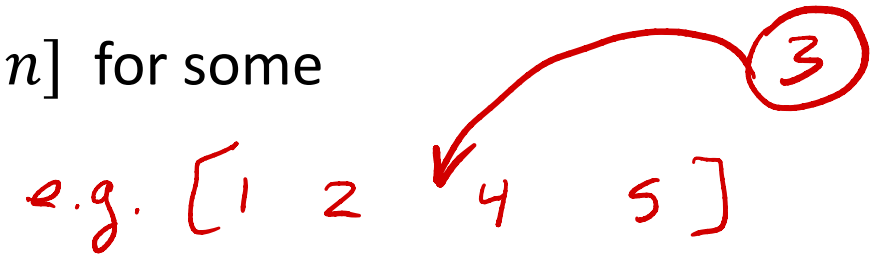
- **Set of Steps/Instructions**: Tells someone or something what to do.
- **Unambiguous/Well-defined**: Can be followed or executed under some set of assumptions such as a collection of predefined primitive steps.
- **Finite/Representable**: Can be conveyed, explained or conceptualized by humans (and/or computers)
- **Solves a problem/Computes somethings**: Has a predictable and desired outcome

Insertion Sort

Goal: Insert number into sorted list.

Input: A number a and list of n sorted numbers $A = [a_1, a_2, \ldots, a_n]$ where $a_1 \leq a_2 \leq \ldots \leq a_n$

Output: List of numbers $A' = A[1 \ldots j-1] \odot [a] \odot A[j \ldots n]$ for some $1 \leq j \leq n$ where $\odot$ denotes concatenation.



- As we study algorithms, two things that we analyze are the correctness and efficiency of algorithms.

Insertion Sort

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that

$$a'_1 \leq a'_2 \leq \ \ldots \leq a'_n$$

assuming that indexing starts at 1

Algorithm 1 Sort an array in ascending order with insertion sort.

```
1: procedure InsertionSort(A)
2:     for j in 2.. len(A) :
3:         k = A[j]
4:         i = j − 1
5:         while i > 0 and A[i] > k :
6:             A[i + 1] = A[i]
7:             i = i − 1
8:         A[i + 1] = k .
```

Assuming that we start with a sorted prefix A[1]

← key

← index

#Insert $A[j]$ into the sorted list $A[1..j-1]$

} shift

← inserting key in proper location

Insertion Sort

Example: Use insertion sort to sort the list below.

Algorithm 1 Sort an array in ascending order with insertion sort.

```
1: procedure InsertionSort(A)
2:     for j in 2.. len(A) :
3:         k = A[j]
4:         i = j − 1
5:         while i > 0 and A[i] > k :        #Insert A[j] into the sorted list A[1..j − 1]
6:             A[i + 1] = A[i]
7:             i = i − 1
8:         A[i + 1] = k
```

sorted prefix

4	7	6	10	12	3	2	5

Assuming A[1] is sorted.

j = 2

K = A[2] = 7

i = 1

while 1 > 0 and A[1] > 7 no

A[2] = 7

A: 4 7 6 10 12 3 2 5

sorted

j = 3

K = A[3] = 6

i = j − 1 = 2

while 2 > 0 and A[2] > 6 yes

A[3] = A[2] 4 7 7 10 12 3 2 5

i = 1

while 1 > 0 and A[1] > 6 no

A[2] = 6 4 6 7 10 12 3 2 5

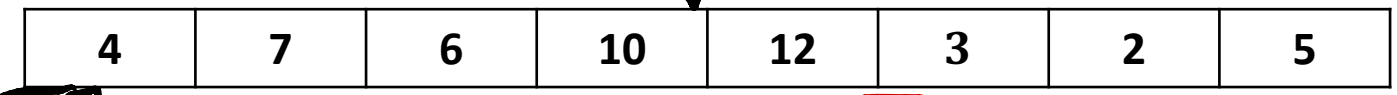
sorted

Insertion Sort

Algorithm 1 Sort an array in ascending order with insertion sort.

```
1: procedure InsertionSort(A)
2:     for j in 2.. len(A) :
3:         k = A[j]
4:         i = j − 1
5:         while i > 0 and A[i] > k :        #Insert A[j] into the sorted list A[1..j − 1]
6:             A[i + 1] = A[i]
7:             i = i − 1
8:     A[i + 1] = k
```

Example: Use insertion sort to sort the list below.



j=4

K = A[4] = 10

i = 3

while 3 > 0 and A[3] > 10

A[4] = 10

A: 4 6 7 10 12 3 2 5

j=5

A: 4 6 7 10 12 3 2 2

j=6

K = A[6] = 3

i = 5

while 5 > 0 and A[5] > 3 yes

A[6] = A[5]

i = 4 4 6 7 10 12 12 2 5

while 4 > 0 and A[4] > 3 yes

A[5] = A[4]

i = 3 4 6 7 10 10 12 2 5

while 3 > 0 and A[3] > 3 yes

A[4] = A[3]

i = 2 4 6 7 7 10 12 2 5

while 2 > 0 and A[2] > 3 yes

A[3] = A[2] 4 6 6 7 10 12 2 5

i = 1

Insertion Sort

Example: Use insertion sort to sort the list below.

Algorithm 1 Sort an array in ascending order with insertion sort.

```
1: procedure InsertionSort(A)
2:     for j in 2.. len(A) :
3:         k = A[j]
4:         i = j − 1
5:         while i > 0 and A[i] > k :        #Insert A[j] into the sorted list A[1..j − 1]
6:             A[i + 1] = A[i]
7:             i = i − 1
8:         A[i + 1] = k
```

insert key

4	7	6	10	12	3	2	5

.... exit while loop A[1] = 3

3	4	6	7	10	12

sorted

2 5

we would keep going until j = 8

Insertion Sort

Correctness & Efficiency

An algorithm is said to be **correct** if, for every input instance, it halts with the correct output.

We say a correct algorithm solves the given computational problem.

Efficiency is typically measured as the speed in which an algorithm comes up with a result.

Insertion Sort

Consider looping over the input array and inserting each number into an already sorted list such that the resulting list is also sorted.

A:	100	40	43	54	26	43	72	16	85	36

Practice a few iterations

Loop Invariants

A **loop invariant** is a property or a set of properties that holds at the beginning of each iteration of a program loop.

❖ Loop invariants can help make it clear whether an algorithm is correct or not.

Method of Loop Invariant:

Initialization - The loop invariant is true prior to the first iteration of the loop

Maintenance - If the loop invariant is true before iteration i of the loop, it remains true before iteration $i + 1$.

Termination - When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

Loop Invariants

Algorithm 1 Sort an array in ascending order with insertion sort.

```
1: procedure InsertionSort(A)
2:     for j in 2.. len(A) :
3:         k = A[j]
4:         i = j − 1
5:         while i > 0 and A[i] > k :          #Insert A[j] into the sorted list A[1..j − 1]
6:             A[i + 1] = A[i]
7:             i = i − 1
8:         A[i + 1] = k
```

Assuming this inner loop is correct.

Claim 1: Assuming $A[1 \ldots j-1]$ is sorted, the inner loop correctly inserts k into the sorted list $A[1 \ldots j-1]$ such that the resulting $A[1 \ldots j]$ is sorted.

↑ Assume this is true

Loop Invariants

Algorithm 1 Sort an array in ascending order with insertion sort.

```
1: procedure InsertionSort(A)
2:     for j in 2.. len(A) :
3:         k = A[j]
4:         i = j − 1
5:         while i > 0 and A[i] > k :        #Insert A[j] into the sorted list A[1..j − 1]
6:             A[i + 1] = A[i]
7:             i = i − 1
8:         A[i + 1] = k
```

claim 1 refers to correctness of inner loop

claim 2 refers to the correctness of the outer loop

I

Claim 2: At the start of each iteration of the for loop, the subarray $A[1 \ldots j-1]$ consists of the elements originally in $A[1 \ldots j-1]$, but in sorted order.

↑ prove this with a "loop invariant proof"

e.g. $j = 4$ assuming

A: [5 4 3 2 1]

A: [3 4 5 | 2 1]

Loop Invariants

Proof Sketch:

Loop Invariant: Prefix is sorted.

Initialization: Every list has an empty or single element prefix that is always sorted.

Maintenance: If I assume a sorted prefix of length q, I can always insert the $q + 1$ number into the prefix such that the resulting list has a sorted prefix of length $q + 1$

Termination: If the above holds for $q = n - 1$, then the maintenance step implies that it holds for n which means the entire list is sorted.

Loop Invariants

indices

Claim 2: At the start of each iteration of the for loop, the subarray $A[1 \ldots j-1]$ consists of the elements originally in $A[1 \ldots j-1]$, but in sorted order.

Proof:

Initialization: The loop initiates at $j=2$.
That tells us that $A[1 \ldots j-1] = A[1]$
$A[1]$ is just a single term and is therefore sorted.

sorted prefix

Maintenance: At the start of the j^{th} iteration, $A[1 \ldots j-1]$ is sorted.
In one run of the loop, the inner while loop loops back to find the place for $A[j]$ in the sorted $A[1 \ldots j-1]$. (we assume the while loop inserts $A[j]$ correctly). The while loop places $A[j]$ via swaps/shifts so that $A[1 \ldots j]$ is sorted. The loop invariant is thus maintained.

Termination: At the start of the last loop, $j=n$
we assume the loop invariant holds before this last iteration. $\Rightarrow A[1 \ldots n-1]$ is sorted.
Using the maintenance step, we get that $A[1 \ldots n]$ is sorted.
j increments to $j=n+1$, we terminate the for loop with with A sorted. ∎

Loop Invariants

Example: Consider the following algorithm. We seek to prove that the algorithm is correct using a loop invariant proof.
First, provide a loop invariant that is useful in proving the algorithm is correct.

```
ProductArray(A[1, ..., n]) : //array A is not empty
    product = 1
    for i = 1 to n {
        product = product * A[i]
    }
    return product
```

Prior to the start of iteration i:

$$product = 1 * A[1] * A[2] * \dots * A[i-2] * A[i-1]$$

(Note: when $i = 0$, the empty product by mathematical convention $= 1$)

Loop Invariants

Example: Using the loop invariant above, provide the initialization component of the loop invariant proof. That is, show that the loop invariant holds before the first iteration of the loop is entered.

```
ProductArray(A[1, ..., n]) : //array A is not empty
    product = 1
    for i = 1 to n {
        product = product * A[i]
    }
    return product
```

Initialization: Prior to the start of the loop, product = 1

This is the correct product of the empty product because $\prod_{i=1}^{0} A[i] = 1$ by mathematical convention.

Loop Invariants

Example: Using the loop invariant above, provide the maintenance component of the loop invariant proof. That is, assume the loop invariant holds just before the i −th iteration of the loop, and use this assumption to show that it still holds just before the $(i+1)$ −st iteration.

```
ProductArray(A[1, ..., n]) : //array A is not empty
    product = 1
    for i = 1 to n {
        product = product * A[i]
    }
    return product
```

Maintenance

Suppose the loop invariant holds prior to the start of iteration i, for $1 \leq i < n$
We must show that the loop invariant holds prior to the start of iteration $i+1$.

Since the loop invariant holds prior to the start of iteration i, we have that
product = $1 \times A[1] \times \cdots \times A[i-2] * A[i-1]$

At the end of iteration i, product = product $* A[i]$
$= (1 * A[1] \times \cdots \times A[i-2] * A[i-1]) * A[i]$

Thus at iteration $i+1$, product = $1 * A[1] * A[2] * \cdots A[i-2] * A[i-1] * A[i]$ and the loop invariant holds.

Loop Invariants

Example: Using the loop invariant above, provide the termination component of the loop invariant proof. That is, assume the loop invariant holds just before the last iteration. Then argue that the loop invariant holds after the loop terminates, based on what happens in the last iteration of the loop. Finally, use this to argue that the algorithm overall is correct.

```
ProductArray(A[1, ..., n]) : //array A is not empty
    product = 1
    for i = 1 to n {
        product = product * A[i]
    }
    return product
```

Termination:

Suppose the loop invariant holds prior to the start of iteration n.

Then product = $1 * A[1] * \dots * A[n-2] * A[n-1]$

At iteration n product = product $* A[n]$

$= (1 * A[1] * \dots * A[n-2] * A[n-1]) * A[n]$

$= 1 * A[1] * \dots * A[n-2] * A[n-1] * A[n]$

So product stores the correct product of the elements of input A.

After the loop terminates, ProductArray returns product. Thus the algorithm correctly returns the product of the elements of array A, as desired ■

Next Time:

- Proof by Induction