Agenda:

- Recursions & Induction
- Recursions to Loops
- Loop invariants (CLRS p.18-20)

Why Prove Correctness

- So far we have discussed how to write down an algorithm, and now comes the next step — how to prove that it does what it is supposed to do. (And *never* errs!)
- Recall: an algorithm is a series of step-by-step instructions that for *any* input must produce the correct output.
- I.e., for all instances, it produces the right output.
- I.e., there's no single instance on which it "badly behaves"
- And since we cannot test our algorithm on any possible input — we must *prove* its correctness.
- This starts by wording correctness formally:

  Claim: For any instance I (satisfying _____),
  `Algorithm-name`(I) returns _____

- E.g., For any two non-negative integers a and b, `Multiply`(a, b) returns the product $a \times b$.

Why should I know how to prove correctness?

- A proof is reasoning made precise.
  - If you can reason about the code, then you can prove its correctness.
  - Conversely, if you can't prove code's correctness, you can't reason about it
- A proof shows you understand what the code does and why.
- A false statement:
  "I can explain why the code works, I just can't prove it..."
  - Take your intuitive explanation, break it into sentences.
  - Turn each sentence into a statement of its own — into a formal claim.
  - Make sure each statement has a justification.
  - Make sure the flow of all statements together actually leads to the right conclusion that "for all I, `Algorithm-name`(I) returns...."
- `procedure Sum`(x, y)
  $a \leftarrow x$
  $a \leftarrow a + y$
  `return` (a)
- "We return $x + y$ because we return a which was x and then we added y to it."
  - Fix any x and y.
  - In the first line we set the variable a to x.
  - Then (in the 2nd line) we add y to a, so now a holds $x + y$.
  - Finally we return a, which holds $x + y$.
  - So for any x and y, `Sum`(x, y) returns $x + y$.
- This is of course a toy example, but the principle remains the same.

Why should I know how to prove correctness?

- A proof is reasoning made precise.
- A proof shows you understand what the code does and why.
- A false statement:
  "Obviously the code does what it is supposed to. Look at this example..."
  - You cannot prove correctness using examples.
  - You have to argue about all instances.
  - It is however extremely useful to our understanding of the code to go over a few examples – as diverse as possible – and see how the code runs on them.
  - However, there's a gap between specific and concrete examples to arguing that on *all* inputs the code does what it is supposed to.
  - A tip: do a few examples, then try to repeat them only when the input is unspecified / in a black-box.
- `procedure Sum(`x, y`)`
  $a \leftarrow x$
  $a \leftarrow a + y$
  `return (`a`)`
  - `Sum(`$5, 14$`)`: first $a = 5$, then $a = 5 + 14 = 19$ we return 19.
  - `Sum(`$-45, 224$`)`: first $a = -45$, then $a = -45 + 224 = 179$ we return 179.
  - ...
  - `Sum(`$\square, \triangle$`)`: first $a = \square$, then $a = \square + \triangle$, we return $(\square + \triangle)$.
  - Hence: `Sum(`x, y`)` returns $x + y$.
- This is of course a toy example, but the principle remains the same.

Basic Proofs

- We will learn many different proofs techniques for a variety of different algorithms.
- But they all begin with basic logic.
  In fact, when an algorithm contains only simple statements, then logic is all we have to use.
- `procedure Swap`(a, b)
  $temp \leftarrow a$
  $a \leftarrow b$
  $b \leftarrow temp$
- Claim: for any two pointers a and b, `Swap`(a, b) indeed assigns a the element that b pointed to originally, and assigns b the element that a pointed to originally.
- Proof: Assume that initially a points to object x and b points to object y.
  The first line creates a new pointer $temp$ that also points to x.
  The second line sets a to point to y (just like b).
  Finally the last line sets b to point to the same object as $temp$, i.e. x.
  So, at the end of the execution, a points to y and b points to x, as required. $\square$

Recursion

- What about code that's written using a recursion?
- The factorial operation: $n! = \prod_{i=1}^{n} i$ (with $0! = 1$)
- Recursive implementation:

```
procedure factorial(n)
if (n = 0) then
    return 1                          ** Base case
else
    return n × factorial(n − 1)       ** Recursive call
```

- How do we prove the correctness of recursive code?
- It is simple to argue that `Factorial(0)` returns the correct answer.
- Based on that, we can argue that `Factorial(1)` returns the right answer.
- Based on that, we can argue that `Factorial(2)` returns the right answer.

  ...
- We need a proof that starts at a simple (base) case, and progresses from there until it covers all integers...
- I.e., we prove correctness of recursions using induction!

Induction

- Claim: For every natural number n, $\texttt{factorial}(n) = n!$.
- Proof: By induction.
  - (Base case) We show the claim holds for some initial value.
    - For $n = 0$ we have that $0! = 1$ and $\texttt{factorial}(0) = 1$.
  - (Induction step) Fix some k. Assuming the claim holds for k, we show the claim also holds for $k + 1$.
    - Assuming $\texttt{factorial}(k) = k!$, we have that

$$
\begin{aligned}
\texttt{factorial}(k+1) &= (k+1) \times \texttt{factorial}(k) && **\text{ since } k+1 > 0 \\
&= (k+1) \times k! && **\text{ using induction assumption} \\
&= (k+1) \times \prod_{i=1}^{k} i \\
&= \prod_{i=1}^{k+1} i = (k+1)! && \blacksquare
\end{aligned}
$$

Induction

- Induction proof structure
  - **Base case:** We show the claim holds for some initial value. (Not necessarily 0)
  - **Induction step:** Fix some natural number k. Assuming the claim holds for f, we show that the claim also holds for $k + 1$.
- Alternative: (Full / Complete induction)
  - **Induction step:** Fix some natural number k. Assuming the claim holds for all natural numbers $0 \leq i \leq k$ we show it also holds for $k + 1$.
- Induction is a powerful and a commonly used tool.
- Must-haves in induction proofs:
  - A *clearly defined* base case
  - A *well-defined* assumption for any instance of a fixed size
  - A correct induction step — that indeed works for any k.
- Without these (and they are sometimes subtle) inductions can go horribly wrong.

Induction Proof, Example #2

- Recall our sorting algorithm

```
procedure InsertionSort(A, n)
if (n > 1) then
    InsertionSort(A, n − 1)
    x ← A[n]
    PutInPlace(A, n − 1, x)
```

```
procedure PutInPlace(A, j, x)
if (j = 0) then
    A[1] ← x
else if (x > A[j]) then
    A[j + 1] ← x
else          ** i.e., x ≤ A[j]
    A[j + 1] ← A[j]
    PutInPlace(A, j − 1, x)
```

- Claim 1: For any array A containing n pair-wise comparable elements, `InsertionSort`(A, n) correctly sorts A.
- Proof: We prove the claim via induction on the number of elements of A.
  - Base case: `InsertionSort`$(A, 1)$ correctly sorts any array of size 1 — as the code does nothing and an array of size 1 is trivially sorted.
  - Induction Step: Fix some natural $n \geq 1$. Assuming `InsertionSort`(A, n) correctly sorts any array of size n, we show `InsertionSort`$(A, n + 1)$ also correctly sorts any array of size $n + 1$.

Induction Proof, Example #2

- Induction Step: Fix some natural $n \geq 1$. Assuming `InsertionSort`(A, n) correctly sorts any array of size n, we show `InsertionSort`$(A, n+1)$ also correctly sorts any array of size $n+1$.
- Since $n+1 > 1$ then we first invoke a recursive call to sort the first n elements of the array. By IH, the recursive call indeed correctly sorts the first n elements.
  We then use the function `PutInPlace` to put the last element (the $n+1$th-element) in its right place.
- "Claim 2:" `PutInPlace()` does what it is supposed to do.
- Assuming Claim 2 is correct, then `PutInPlace`$(A, n, A[n+1])$ places $x = A[n+1]$ in its right place, making the whole $n+1$ elements of the array sorted. $\square$
- What's left is to state Claim 2 formally, and prove it.

Induction Proof, Example #2

- `procedure PutInPlace`(A, j, x)

  `if` $(j = 0)$ `then`
  $A[1] \leftarrow x$
  `else if` $(x > A[j])$ `then`
  $A[j+1] \leftarrow x$
  `else` ** i.e., $x \leq A[j]$
  $A[j+1] \leftarrow A[j]$
  `PutInPlace`$(A, j-1, x)$

- Claim 2: For any array A, and natural number j such that (i) A has (at least) $j+1$ cells, and (ii) the subarray $A[1, ..., j]$ is sorted, when `PutInPlace`(A, j, x) terminates, the first $j+1$ cells of A contain all the elements that were originally in $A[1, .., j]$ plus x in sorted order.
- Proof: By induction. Base case is when $j = 0$, for which the claim is true as `PutInPlace` just puts x in the first cell of A.
  Induction step. Fix $j \geq 1$. Assuming the claim holds for $j-1$ we show it also holds for j.
  Case I: if x is greater than $A[j]$ — then x is greater than all elements in $A[1, ..j]$, so by putting x is the $j+1$-th cell the array $A[1, .., j+1]$ satisfies the required: sorted, and contain all the required elements.

  Case II: If $A[j] \geq x$ then $A[j]$ is the max-element of $A[1, .., j]$ plus x. So we put $A[j]$ in the $j+1$-th cell. Then by invoking `PutInPlace` on $A[1, ..j-1]$, by IH, the result is that $A[1, ..j]$ contains the elements of $A[1, .., j-1]$ and x, sorted. This means that altogether $A[1, ..j+1]$ is sorted. □

Induction Proof, Example #2

- HW: look at the other problems we solved through recursion. Prove each of those algorithm's correctness formally.

A bit of history (optional material)

- Math was a well-established discipline when questions such as "what is a proof?" and "how can we define a proof?" came about (mid 19th century); and the discipling of mathematical logic began to be established (Frege,1879, "Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens [Concept-Script: A Formal Language for Pure Thought Modeled on that of Arithmetic]").
- And the most astounding revelation came with Gödel's incompleteness theorem (1931): in every logic system that is non-trivial (i.e, I can write many claims) and sound (I cannot prove an erroneous claim), there's a claim which cannot be proven or refuted.
  (Gödel shows that any such logic system one can state the claim "I am a false claim".)

- Logic and computation were intertwined from the get-go:
  In mathematical logic, verifying a proof / a refutation of a claim is no more than an algorithmic act. (Each line is an axiom, definition, or a conclusion from previous lines.)
- So in 1934 Gödel asked — can we have an automated verification system?

A bit of history, cont'd (optional material)

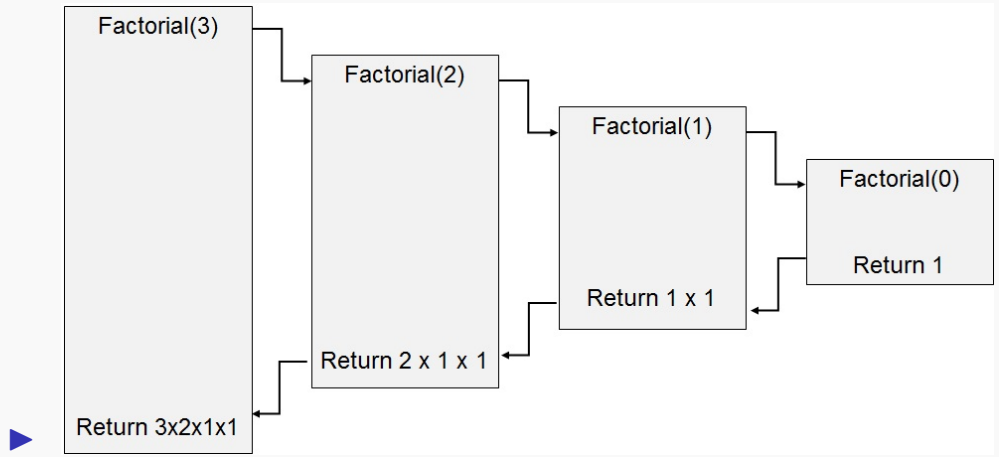
- 1936, Alonzo Church answers Gödel's question on the affirmative, as he defines computation using (Gödel's model of) recursive functions. (λ-calculus).
- Same year, independent work, Alan Turing's paper ("On Computable Numbers, with an Application to the Entscheidungsproblem") models computation using an imaginary machine (known today as "Turing machine" or by its alternative name "computer") that basically has memory cells, loops and if-conditions.
  Using this machine Turing shows an incompleteness result: there does not exist a Turing Machine that solves the Halting problem —
  Input: a code C of some Turing Machine and input I;
  Output: `Yes` if $C(I)$ halts or `No` if $C(I)$ goes into an infinite loop.
- So, which one of the two models for computation is "right?"
- It takes a few years, but eventually equality between the two models is proven. This is known as the Church-Turing thesis.
- Which, in lay-man terms states:
  any problem you can solve using a recursion, can also be solved using loops and vice-versa.
  (Caveat: it might be that one approach is more efficient that the other.)

From Recursions to Loops

- Armed with the Church-Turing thesis, we convert recursions into loops.
- So let's convert this into a loop:

```
procedure factorial(n)
if (n = 0) then
   return 1                          ** Base case
else
   return n × factorial(n − 1)       ** Recursive call
```

- There are more than a single way to convert a recursive code into a loop.
- Here's a basic one: try to see a patten in what the recursive calls return — in the order of conclusion (not the order of calls!)



From Recursions to Loops

- Try to see the patten in what the recursive calls return — in the order of conclusion (not the order of calls!)
- This means we are after a loop that
  - Upon 0 iterations — sets the result to 1
  - Upon 1 iteration — sets the result to $1 \times 1 = 1$
  - Upon 2 iterations — sets the result to $2 \times 1 \times 1 = 2$
  - Upon 3 iterations — sets the result to $3 \times 2 \times 1 \times 1 = 6$
  - Upon 4 iterations — sets the result to $4 \times 3 \times 2 \times 1 \times 1 = 24$
  - ...
- `procedure factorial`(n)
  $res \leftarrow 1$
  `for (`j `from` 1 `to` n`)`
  $\quad res \leftarrow j \times res$
  `return` res
- Note that this is not the only possible code — these are also valid options:

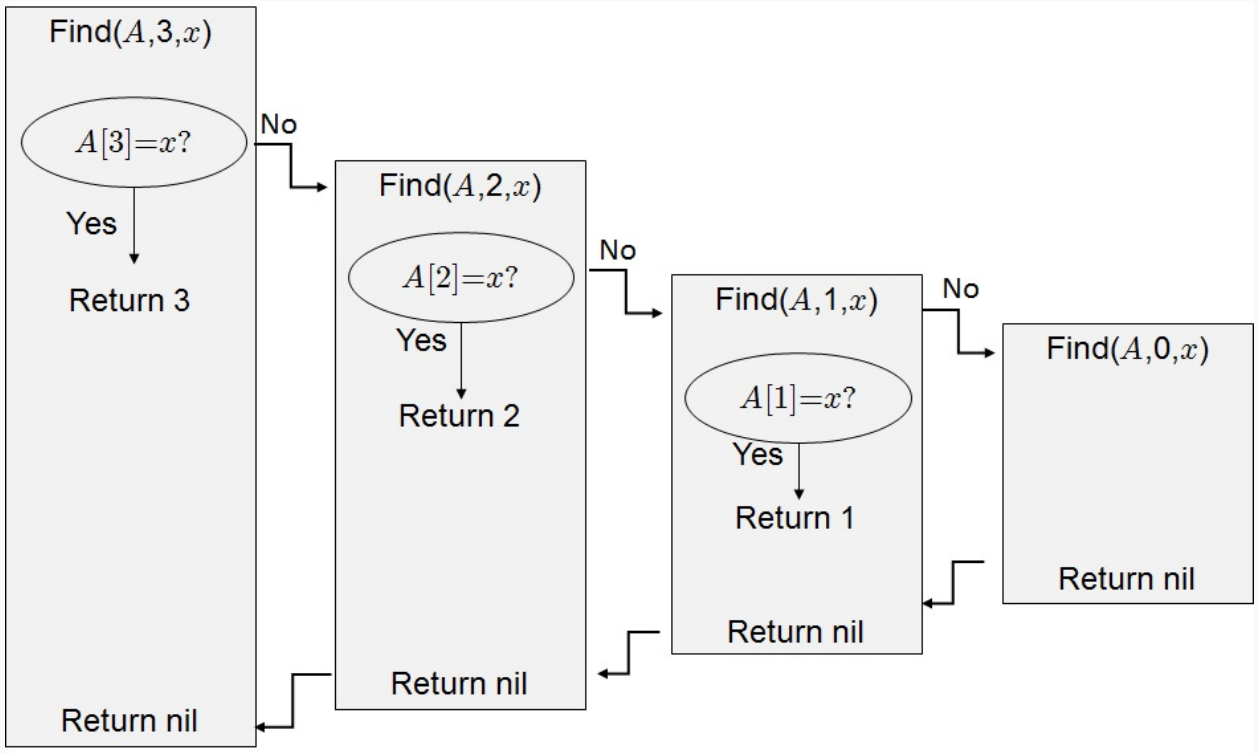
`procedure factorial`(n)
`if (`$n \leq 1$`) then`
$\quad$ `return` 1
$res \leftarrow 1$
`for (`j `from` 2 `to` n`)`
$\quad res \leftarrow j \times res$
`return` res

`procedure factorial`(n)
`Set` A `an array of size` $n+1$
$A[1] \leftarrow 1$
`for (`j `from` 1 `to` n`)`
$\quad A[j+1] \leftarrow j \times A[j]$
`return` $A[n+1]$

From Recursions to Loops

- Example #2:

```
procedure Find(A, n, x)
if (n = 0) then
   return nil
else if (A[n] = x) then
   return n
else
   return Find(A, n − 1, x)
```



From Recursions to Loops

- Example #2:

```
procedure Find(A, n, x)
if (n = 0) then
   return nil
else if (A[n] = x) then
   return n
else
   return Find(A, n − 1, x)
```

- This means we are after a loop that
  - First checks $A[n] = x$, and if so — returns n
  - Then checks $A[n-1] = x$ and if so — returns $n-1$
  - Then checks $A[n-2] = x$ and if so — returns $n-2$
  - ...
  - Then checks $A[1] = x$ and if so — returns 1
  - If all checks fail — returns `nil`
-

```
procedure Find(A, n, x)
for (j from n downto 1)        ** j decreasing
   if (A[j] = x) then
      return j
return nil
```

From Recursions to Loops

- Continuing on the same example. Consider another option:

```
procedure Find(A, n, x)
if (n = 0) then
   return nil
else if (A[n] = x) then
   return n
else
   return Find(A, n − 1, x)
```

```
procedure Find(A, n, x)
for (j from 1 to n)          ** j increasing
   if (A[j] = x) then
      return j
return nil
```

- This loop-code doesn't have the same exact input-output relation as the original recursion
  (E.g., consider executing both codes on $A = [17, 3, 17]$, $n = 3$ and $x = 17$)
- ... But both codes solve the same problem:
  "Given an array A, an index n and an element x, how do we find an index i between 1 and n such that $A[i] = x$ or indicate no such index exists."
- And so — I do not care which one you use
  - Or rather, out of all the possible codes that solve the same problem, I want you to choose the one that uses the least amount of resources.[1]

[1] Resource analysis — next week, once we are done discussing correctness.

From Recursions to Loops

- Sometimes, switching from recursions to loops is a formidable challenge. (E.g., Towers of Hanoi problem.)
- Sometimes, by switching to a loop — the amount of resources you use decrease *drastically*. E.g., the Fibonnaci series:

```
procedure Fibonacci(n)
if (n ≤ 1) then
   return n
else
   return Fibonacci(n − 1) + Fibonacci(n − 2)
```

- ```
  procedure Fibonacci(n)
  Set A an array of n + 1 cells
  A[1] ← 0
  A[2] ← 1
  for (j from 3 to n + 1)
     A[j] ← A[j − 1] + A[j − 2]
  return A[n + 1]
  ```

```
procedure Fibonacci(n)
if (n ≤ 1) then
   return n
x ← 0
y ← 1
for (j from 2 to n)
   temp ← x + y
   x ← y
   y ← temp
return y
```

- Next week we will discuss why the latter two are (far) better than the first recursive code
```

From Recursions to Loops

- What's the Loop-version of this code?

```
procedure InsertionSort(A, n)
if (n > 1) then
    InsertionSort(A, n − 1)
    x ← A[n]
    PutInPlace(A, n − 1, x)
```

```
procedure PutInPlace(A, j, x)
if (j = 0) then
    A[1] ← x
else if (x > A[j]) then
    A[j + 1] ← x
else          ** i.e., x ≤ A[j]
    A[j + 1] ← A[j]
    PutInPlace(A, j − 1, x)
```

- Turns out, it is the following (CLRS) version:

```
procedure InsertionSort(A, n) **sort A[1..n] in place
for (j from 2 to n)
    key ← A[j]          **insert A[j] into sorted sublist A[1..j − 1]
    i ← j − 1
    while (i > 0 and A[i] > key)
        A[i + 1] ← A[i]
        i ← i − 1
    A[i + 1] ← key
```

- How are we to prove that this code is correct?

Proving Correctness using Loop Invariants

- Definition: A loop-invariant is a statement / assertion / predicate about the state of the code that is *always* true at the beginning of each loop-iteration.
- But not any assertion
  - "At the beginning of the each loop iteration the earth is round..."
- An assertion that *accurately* describes the *cumulative effect* of repeatedly iterating through the loop.
  An assertion we can use to prove the correctness of the code.
- Step 1: Identify the loop invariant
  - Q1: Do I understand what the loop does?
  - Q2: Do I understand the cumulative effect of the loop?
  - Q3: Can I word exactly the cumulative effect of the loop?
- Step 2: Prove the loop invariant
  - Initialization
  - Maintenance
  - Termination #1: Does the loop halt eventually?
  - Termination #2: How do I prove correctness from the LI?

Step #1: Identifying and Rigorously Stating the Loop Invariant

- Consider the code

```
procedure FindSum(A, n)   **Returns the sum of all elements in A[1..n]
sum ← A[1]
j ← 2
while (j ≤ n)
    sum ← sum + A[j]
    j ← j + 1
return sum
```

- Intuitively we understand the code returns the sum of all elements in $A[1, ...n]$. But how do we prove this?
- We identify the invariant of the loop — the thing the always holds whenever the loop executes.
- Q1: What does the loop do?
  - Adds $A[j]$ to the value of sum and increments j
- Q2: So what happens overall in iterations (counting by value of j) $2, 3, ...$ up to (but not including) the jth iteration?
  - sum starts at $A[1]$; then it is $A[1] + A[2]$; then it is $A[1] + A[2] + A[3]$; ....;then it is $A[1] + A[2] + ... + A[j-1]$
- Q3: So it always true when the loop iteration begins?
  - sum holds the summation of $A[1] + A[2] + ... + A[j-1]$
- So, the loop-invariant is:

"At the beginning of each loop iteration, $sum = \sum_{i=1}^{j-1} A[i]$"

Step #1: Identifying and Rigorously Stating the Loop Invariant

- Continuing with the `FindSum` example
- So this is the loop-invariant:

  "At the beginning of each loop iteration, $sum = \sum_{i=1}^{j-1} A[i]$"
- The same loop-invariant can be written in many equivalent forms
  - "At the beginning of each loop iteration, $sum = A[1] + A[2] + ... + A[j-1]$"
  - "At the beginning of each loop iteration sum is the summation of the elements in $A[1, ..., j-1]$"
  - "At the beginning of each loop iteration sum is the summation of the first $j-1$ elements in A"
  - or any other equivalent form
- It DOES matter that the loop invariant will be stated correctly and in a way the will give the correctness of the overall algorithm
  - "At the beginning of each loop iteration $sum = A[j]$" — WRONG
  - "At the beginning of each loop iteration sum is the summation of the elements in $A[1, ..., j]$" — WRONG.
  - "At the beginning of each loop iteration $j > 0$" — UNINFORMATIVE '
  - "At the beginning of each loop iteration $sum = sum^{\text{previous\_iteration}} + A[j-1]$" — UNINFORMATIVE
- To make sure you don't mess with the indices — check it!
  Plug-in values of j ($j = 1$, $j = 2$, $j = n$) and check it

Step #1: Identifying and Rigorously Stating the Loop Invariants

- Consider the code

```
procedure FindMax(A, n)   **Returns the largest element in A[1..n]
index ← 1
for (j from 2 to n)
    if (A[j] > A[index])
        index ← j
return A[index]
```

- Intuitively we understand the code returns the largest element in $A[1, ...n]$. But how do we prove this?
- Q1: What does the loop do?
  - We only update $index$ if $A[j]$ is bigger than $A[index]$, in which case we set $index = j$.
  - So $index$ always points to the location of the largest element between $A[index]$ and $A[j]$.
- Q2: What happens overall in iterations $2, 3, ...$ up to (not including) j?
  - Before the loop $A[index] = A[1]$
  - After the first iteration ($j = 2$), $A[index] = \max\{A[1], A[2]\}$.
  - After the 2nd iteration ($j = 3$), $A[index] = \max\{A[index], A[3]\} = \max\{\max\{A[1], A[2]\}, A[3]\} = \max\{A[1], A[2], A[3]\}$
  - $\vdots$
  - After dealing with $j - 1$, $A[index] = \max\{A[1], A[2], ..., A[j-1]\}$
- Q3: What is the loop invariant?
  - At the beginning of each loop-iteration, $A[index] = \max\{A[1], A[2], ..., A[j-1]\}$

Step #1: Identifying and Rigorously Stating the Loop Invariants

- Consider the code

```
procedure Power(b, n)  **Returns b^n
res ← 1
for (j from 1 to n)
    res ← res × b
return res
```

- Q1: What does the loop do?
  - In each iteration we multiply the variable res by b.
- Q2: What happens overall in iterations $1, 2, ..., j-1$ (up to j, not including)
  - First: $res = 1$
  - When iteration $j = 1$ terminates: $res = b$
  - When iteration $j = 2$ terminates: $res = b \times b = b^2$
  - When iteration $j = 3$ terminates: $res = b \times b \times b = b^3$
  - $\vdots$
  - When iteration $j - 1$ terminates: $res = b^{j-1}$
- Q3: What is the loop invariant?
  - "At the beginning of each loop-iteration, $res = b^{j-1}$"

Step #2: Proving Loop Invariants

- Once we have identified and stated the LI, it is time to prove it — and to use it to prove the correctness of the entire code.
- Proving LI means proving the following 4 parts
- Initialization:
  - Does LI hold before the loop starts?
- Maintenance:
  - If LI holds at the beginning of j-th iteration, does it holds also at the beginning of the $j + 1$-iteration?
- Termination #1:
  - Does the loop terminate?
- Termination #2:
  - When the loop terminates, does it prove the correctness of the overall algorithm / the claim we were making?

Step #2: Proving Loop Invariant

- Consider the code

```
procedure FindSum(A, n)    **Returns the sum of all elements in A[1..n]
sum ← A[1]
j ← 2
while (j ≤ n)
    sum ← sum + A[j]
    j ← j + 1
return sum
```

- Our loop-invariant: "At the beginning of each loop iteration, $sum = \sum_{i=1}^{j-1} A[i]$"
- Initially: Before the loop begins $sum = A[1] = A[1, ..., (2-1)]$
- Maintenance: Suppose that at the beginning of the j-th loop iteration, $sum = \sum_{i=1}^{j-1} A[i]$.
  Then, at the beginning of the $j+1$ iteration
  $$sum^{\text{after}} = sum^{\text{before}} + A[j] \overset{\text{LI}}{=} \sum_{i=1}^{j-1} A[i] + A[j] = \sum_{i=1}^{j} A[i] = \sum_{i=1}^{j^{\text{after}}-1} A[i]$$
- Termination: The loop terminates as we only increment j, so eventually we would have $j > n$

Step #2: Proving Loop Invariant

- Consider the code

```
procedure FindSum(A, n)    **Returns the sum of all elements in A[1..n]
sum ← A[1]
j ← 2
while (j ≤ n)
    sum ← sum + A[j]
    j ← j + 1
return sum
```

- Our loop-invariant: "At the beginning of each loop iteration, $sum = \sum_{i=1}^{j-1} A[i]$"
- Remember, our goal is to prove the correctness of `FindSum`.
- Loop Invariant is just the tool that we use in order to argue the overall correctness.
  It is only means to an end.
- That is why we also do another termination reasoning:
- Termination #2: When the while-loop terminates, $j = n + 1$, in which case the LI implies $sum = \sum_{i=1}^{n} A[i]$. We return sum which means we return the sum of all elements in $A[1, ..., n]$.

Step #2: Proving Loop Invariant

- Sometimes, there's more to Termination #2.
- Consider the code

```
procedure FindAvg(A, n)  **Returns the average of all elements in A[1..n]
sum ← A[1]
j ← 2
while (j ≤ n)
    sum ← sum + A[j]
    j ← j + 1
return (sum/n)
```

- Our loop-invariant is the same as before: "At the beginning of each loop iteration, $sum = \sum_{i=1}^{j-1} A[i]$"
- To argue we indeed return the average of all n elements in A our "termination" argument now looks like:
- Termination #2: When the while-loop terminates, $j = n + 1$, in which case the LI implies $sum = \sum_{i=1}^{n} A[i]$. We return $sum/n = \frac{1}{n} \sum_{i=1}^{n} A[i]$ which by definition is the average of all elements in $A[1, ..., n]$.

Step #2: Proving Loop Invariants

- Consider the code

```
procedure Power(b, n)  **Returns b^n
res ← 1
for (j from 1 to n)
    res ← res × b
return res
```

- LI: "At the beginning of each loop-iteration, $res = b^{j-1}$"
- Initially, $res = 1 = b^0$ and the very first value of j is 1.
- Maintenance: We assume the LI holds at the beginning of loop j and show it also holds at the beginning of loop $j + 1$.
  $res^{\text{after}} = res^{\text{before}} \cdot b \overset{\text{LI}}{=} b^{j^{\text{before}}-1} \cdot b = b^{j^{\text{before}}-1+1} = b^{j^{\text{after}}-1}$
- Termination #1: The for-loop only increment j and we do not change its value
- Termination #2: At the end of the for-loop, $j = n + 1$ so the LI gives that $res = b^n$ — Correctness of the overall code proven.

More Loop Invariants Examples

- ```
  procedure InsertionSort(A)
  for (j from 2 to n)
      key ← A[j]          **insert A[j] into sorted sublist A[1..j − 1]
      i ← j − 1
      while (i > 0 and A[i] > key)
          A[i + 1] ← A[i]
          i ← i − 1
      A[i + 1] ← key
  ```
- To prove correctness - use two loop invariants, one *nested* inside another.
- What is the loop invariant of the for-loop?
- LI1: "At the beginning of each for-loop iteration $A[1, ..., j-1]$ contains the same elements that were there initially, only in order."
- Initialization: $j = 2$ and clearly $A[1]$ is a sorted array of size 1.
- Maintenance: TBD
- Termination #1: We don't alter j at the body of the loop + Termination of the while-loop (TBD)
- Termination #2: When the loop terminates, $j = n + 1$ so $A[1, ...n]$ (which is the whole array) is sorted.
- But how to prove maintenance?
  ```

More Loop Invariants Examples

- ```
  procedure InsertionSort(A)
  for (j from 2 to n)
      key ← A[j]            **insert A[j] into sorted sublist A[1..j − 1]
      i ← j − 1
      while (i > 0 and A[i] > key)
          A[i + 1] ← A[i]
          i ← i − 1
      A[i + 1] ← key
  ```
- To prove the maintenance property of the LI for the for-loop we actually use a LI for the while-loop
  - LI2: Let $A^{\text{before}}[1..j]$ denote the array before we started iterating through the `while` loop. Then at the beginning of each iteration of the `while` loop:
    (i) $A[1..i+1] = A^{\text{before}}[1..i+1]$
    (ii)$A[i+2..j] = A^{\text{before}}[i+1..j-1]$
- Initialization / maintenance / termination #1 of LI2:
  - HW
  ```

More Loop Invariants Examples

- `procedure InsertionSort`(A)

  `for (`j `from` 2 `to` n`)`
  $key \leftarrow A[j]$ `**insert` $A[j]$ `into sorted sublist` $A[1..j-1]$
  $i \leftarrow j-1$
  `while (`$i > 0$ `and` $A[i] > key$`)`
  $A[i+1] \leftarrow A[i]$
  $i \leftarrow i-1$
  $A[i+1] \leftarrow key$

- To prove the maintenance property of the LI for the for-loop we actually use a LI for the while-loop
  - LI2: Let $A^{\text{before}}[1..j]$ denote the array before we started iterating through the `while` loop. Then at the beginning of each iteration of the `while` loop:
    (i) $A[1..i+1] = A^{\text{before}}[1..i+1]$
    (ii)$A[i+2..j] = A^{\text{before}}[i+1..j-1]$
- The termination #2 of LI2 is how to derive the maintenance property of LI from the termination of the while-loop.
- Termination #2: At the end of while loop, i is the largest entry in $\{1, 2, 3, ..., j-1\}$ for which $A[i] \leq key$ (or 0, if no such entry exists). So LI2 together with putting key at $A[i+1]$, we have that

$$A[1, ..j] = \left[A^{\text{before}}[1, .., i], key, A^{\text{before}}[i+1, .., j-1]\right]$$

As $A^{\text{before}}[1, ..j-1]$ was sorted & by definition of $i \Rightarrow A[1, ..j]$ is sorted.
□

Loop invariant vs. Mathematical induction

- Arguing correctness
  - When recursion is involved, use induction
  - When loop is involved, use loop invariant (and induction)
- Common points
  - initialization vs. base step
  - maintenance vs. inductive step
- Difference
  - termination vs. infinite

In Practice
(or: Advice for life)

- When writing code it is always a good practice to document it, and add comments explaining its purpose and outline.
- It is quite common for people to write the LI of the code.
- Why?
- Because a (good) LI summarizes exactly what the loop does
  - And implicitly — what it doesn't do...
- Months/years from now, when you try to remember what actually goes on in the code (or when someone else tries to understand what you wrote), the LI makes it so much easier.
- So make it your habit, when you write code, to put next to a loop the loop-invariant
  - And also output the variables in the loop after each iteration, so that you can verify your LI statement is in fact accurate.
- It will only earn you the compliments of your peers...

Summary

- Prove of correctness: instead of applying the code to all possible instances, prove using logic that applying the code for any instance always outputs the correct result.
- A proof: reasoning made precise.
  - If you understand what the code does and why, you should be able to argue it as a formal proof.
    (Just break your explanation into statements, make sure each one is true and that altogether they lead to the right conclusion...)
  - And if you can't argue a formal proof, chances are you didn't really understand what the code does and why...
- Recursions are proved using induction
  - Standard or Full / Complete Induction
  - Make sure you cover the base case
- A recursive code can be turned into a code with only loops.
  - And there could be multiple ways to do it, with various consequences.
- Correctness of a code that uses loops — proved via a loop induction
  - Similar to induction — show that (i) LI holds initially, and (ii) that if it holds at the beginning of iteration then it also holds at the beginning of the next iteration
  - Unlike induction — also show (i) the loop eventually terminates and (ii) that upon termination the LI leads to the correctness of the full code.