Induction

Lecture 22
Spring 2011

1

Goals for Today

- Be able to state the principle of induction
  - Identify its relationship to recursion
  - State how it is different from recursion
- Be able to understand inductive proofs
  - Identify the reason why induction is necessary
  - Follow most important steps of the proof
- Be able to construct simple inductive proofs
  - More of this to come next lecture, discussion

2

Overview

- **Recursion**
  - A **programming**/**algorithm strategy**
  - Solves a problem by reducing it to simpler or smaller instance(s) of the same problem
- **Induction**
  - A **mathematical proof technique**
  - Proves statements about natural numbers 0,1,2,...
  - (or more generally, inductively defined objects)
- Closely related, but different

3

Merge Sort

How do we know this is true?

Or that this is true?

```
/**
 * Sorts the Comparable array x between lo
 * (inclusive) and hi (exclusive), recursively and
 * in O(n log n) time
 */
private void mergeSort(T[] x, int lo, int hi, T[] y) {
  // base case
  if (hi <= lo + 1) return; // nothing to do

  // at least 2 elements
  // split and recursively sort
  int mid = (lo + hi)/2;
  mergeSort(x, lo, mid, y);
  mergeSort(x, mid, hi, y);
  // merge sorted sublists
  merge(x, lo, mid, hi, y);
}
```

```
/**
 * Merge 2 subarrays of x, using y as temp
 */
private void merge(T[] x, int lo, int mid, int hi,
                   T[] y) {
  int i = lo; // subarray pointers
  int j = mid;
  int k = lo; // destination pointer

  while (i < mid && j < hi) {
    y[k++] = (x[i].compareTo(x[j]) > 0)? x[j++] :
                                          x[i++];
  }

  // one of the subarrays is empty
  // copy remaining elements from the other
  System.arraycopy(x, i, y, k, mid - i);
  System.arraycopy(x, j, y, k, hi - j);
  // now copy everything back to original array
  System.arraycopy(y, lo, x, lo, hi - lo);
}
```

4

Merge Sort

Is this still true?

How about this?

```
/**
 * Sorts the Comparable array x between lo
 * (inclusive) and hi (exclusive), recursively and
 * in O(n log n) time
 */
private void mergeSort(T[] x, int lo, int hi, T[] y) {
  // base case
  if (hi <= lo + 1) return; // nothing to do

  // at least 2 elements
  // split and recursively sort
  int mid = lo+1;
  mergeSort(x, lo, mid, y);
  mergeSort(x, mid, hi, y);
  // merge sorted sublists
  merge(x, lo, mid, hi, y);
}
```

```
/**
 * Merge 2 subarrays of x, using y as temp
 */
private void merge(T[] x, int lo, int mid, int hi,
                   T[] y) {
  int i = lo; // subarray pointers
  int j = mid;
  int k = lo; // destination pointer

  while (i < mid && j < hi) {
    y[k++] = (x[i].compareTo(x[j]) > 0)? x[j++] :
                                          x[i++];
  }

  // one of the subarrays is empty
  // copy remaining elements from the other
  System.arraycopy(x, i, y, k, mid - i);
  System.arraycopy(x, j, y, k, hi - j);
  // now copy everything back to original array
  System.arraycopy(y, lo, x, lo, hi - lo);
}
```

5

Merge Sort

```
/**
 * Sorts the Comparable array x between lo
 * (inclusive) and hi (exclusive), recursively and
 * in O(n log n) time
 */
private void mergeSort(T[] x, int lo, int hi, T[] y) {
  // base case
  if (hi <= lo + 1) return; // nothing to do

  // at least 2 elements
  // split and recursively sort
  int mid = (lo + hi)/2;
  mergeSort(x, lo, mid, y);
  mergeSort(x, mid, hi, y);
  // merge sorted sublists
  merge(x, lo, mid, hi, y);
}
```

Recursion:
- The strategy you used to perform the sorting
- Result is algorithm/program

Induction:
- How you show that the program actually sorts
- Also, how you show it has O(n log n) performance
- Result is a proof/argument

Guides the Process

6

Simpler Example: Sum of Integers

- We can describe a function in different ways
- S(n) = "the **sum of the integers** from 0 to n"
  S(0) = 0, …, S(3) = 0+1+2+3 = 6, …
- **Iterative Definition**
  $S(n) = 0+1+ \ldots + n = \Sigma_{i=0}^{n} i$
- **Closed form characterization**
  $S_C(n) = n(n+1)/2$
- Are S(n) and $S_C(n)$ the same function?

7

What are We Proving?

- Our claim must be a **property** of the natural numbers
  - is a statement with variable n
  - Write as P(n)
  - allows (numeric) values to be substituted for n

    P(0), P(1), P(2), …
- For each number n, P(n) is either true or false

Examples

- P(n): The number n is even
- P(n): Number n is even or odd
- P(n): $S(n) = S_C(n)$
- ~~P(n): MergeSo[illegible] given array~~
- P(n): MergeSort sorts any given array of length n
- P(n): On any given array of length n, MergeSort finishes in less than c (n log n) steps

8

Are These Functions the Same?

- Are the same if same inputs give same outputs
- **Property** P(n): $S(n) = S_C(n)$
- Test some values and see if work
  - S(0) = 0, $S_C(0) = 0(1/2) = 0$ ✓
  - S(1) = 0+1 = 1, $S_C(1) = 1(2/2) = 1$ ✓
  - S(2) = 0+1+2 = 3, $S_C(2) = 2(3/2) = 3$ ✓
  - S(3) = 0+1+2+3 = 6, $S_C(3) = 3(4/2) = 6$ ✓
- This approach will never be complete, as there are infinitely many n to check

9

Recursive Definition

- Let's formulate S(n) in yet another way:
  $S(n) = \underbrace{0 + 1 + 2 + \ldots + n\text{-}1}_{\text{this is } S(n-1)} + n$
- This gives us a recursive definition:
  - $S_R(0) = 0$ ← Base Case
  - $S_R(n) = S_R(n\text{-}1) + n, n > 0$ ← Recursive Case
- Example:
  - $S_R(4) = S_R(3) + 4 = S_R(2) + 3 + 4$
    $= S_R(1) + 2 + 3 + 4 = S_R(0) + 1 + 2 + 3 + 4$
    $= 0 + 1 + 2 + 3 + 4$

10

An Intermediate Problem

- Are these functions the same?
  - **Recursive definition**:
    - $S_R(0) = 0$
    - $S_R(n) = S_R(n\text{-}1) + n, n > 0$
  - **Closed form characterization:**
    - $S_C(n) = n(n+1)/2$
- **Property** P(n): $S_R(n) = S_C(n)$

11

Induction over Natural Numbers

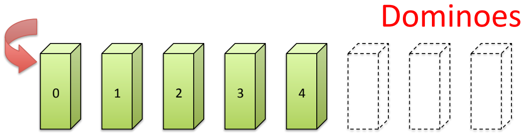
Goal: Prove property P(n) holds for n ≥ 0

1. Base Step:
   - Show that P(0) is true
2. Inductive Step:
   - Assume P(k) true for an unspecified integer k ← Inductive Hypothesis
   - Use assumption to show that P(k+1) is true

Conclusion: Because we could have picked *any* k, we conclude P(n) holds for all integers n ≥ 0

12

Dominoes



- Assume equally spaced dominos, where spacing between dominos is less than domino length.
- Want to argue that all dominoes fall:
  - Domino 0 falls because we push it over
  - Domino 0 hits domino 1, therefore domino 1 falls
  - Domino 1 hits domino 2, therefore domino 2 falls
  - Domino 2 hits domino 3, therefore domino 3 falls
  - …

  } Repetitive argument. Requires one sentence per domino.
- What is a better way to make this argument?

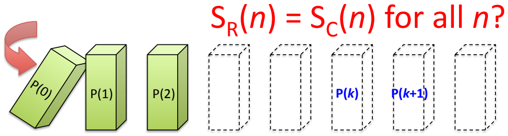
13

A Better Argument

- **Argument**:
  - (Base Step) Domino 0 falls because we push it over
  - (Inductive Hypothesis) Assume domino *k* falls over
  - (Inductive Step) Because domino *k*'s length is larger than the spacing, it will knock over domino *k*+1
  - (Conclusion) Because we could have picked any domino to be the *k*th one, the dominoes will fall over
- This is an inductive argument
  - Much more compact than example from last slide
  - Works for an arbitrary number of dominoes!

14

$S_R(n) = S_C(n)$ for all n?



- Property P(n): $S_R(n) = S_C(n)$
- Base Step:
  - Prove P(0) using the definition
- Inductive Hypothesis (IH):
  - Assume that P(k) holds for unspecified k
- Inductive Step:
  - Prove that P(k+1) is true using IH and the definition

15

Proof (by Induction)

- Recall:
  - $S_R(0) = 0$, $S_R(n) = S_R(n\text{-}1) + n$, $n > 0$
  - $S_C(n) = n(n+1)/2$
- **Property** P(n): $S_R(n) = S_C(n)$
- Base Step: $S_R(0) = 0$ and $S_C(0) = 0$, both by definition
- Inductive Hypothesis: Assume $S_R(k) = S_C(k)$
- Inductive Step:

$S_R(k+1) = S_R(k) + (k+1)$	Definition of $S_R(k+1)$
$= S_C(k) + (k+1)$	Inductive Hypothesis
$= k(k+1)/2 + (k+1)$	Definition of $S_C(k)$
$= [k(k+1)+2(k+1)]/2 = (k+1)(k+2)/2$	Algebra
$= S_C(k+1)$	Definition of $S_C(k+1)$

- Conclusion: $S_R(n) = S_C(n)$ for all $n \geq 0$

16

Our Original Problem

- S(n) = "the **sum of the integers** from 0 to n"

  S(0) = 0, …, S(3) = 0+1+2+3 = 6, …
- **Iterative Definition**

  $S(n) = 0+1+ \ldots + n = \Sigma_{i=0}^{n}\, i$
- **Closed form characterization**

  $S_C(n) = n(n+1)/2$
- **Property** P(n): $S(n) = S_C(n)$ ⟵ Did we show this?

17

Finishing the Proof

- Can just show that $S(n) = S_R(n)$
  - For some, this is a convincing argument:

    $S(n) = \boxed{0 + 1 + 2 + \ldots + n\text{-}1} + n$

    (this is S(n-1))
  - Can also do another inductive proof
- Or could have worked it into our original proof
  - **Old** P(n): $S(n) = S_C(n)$
  - **New** P(n): $S(n) = S_R(n) = S_C(n)$

    Implies

    "Recursive Go-Between"

18

A Complete Argument

- Recall:
  - $S(n) = 0 + 1 + \ldots + n$
  - $S_R(0) = 0$, $S_R(n) = S_R(n\text{-}1) + n$, $n > 0$
  - $S_C(0) = n(n+1)/2$
- **Property** $P(n)$: $S(n) = S_R(n) = S_C(n)$
- Base Step: $S(0) = 0$ and $S_R(0) = 0$ and $S_R(0) = 0$, all by definition
- Inductive Hypothesis: Assume $S(k) = S_R(k) = S_C(k)$
- Inductive Step: First prove $S(k+1) = S_R(k+1)$

$S(k+1) = 0 + 1 + \ldots + k + (k+1)$	Definition of $S(k+1)$
$= S(k) + (k+1)$	Definition of $S(k)$
$= S_R(k) + (k+1)$	Inductive Hypothesis
$= S_R(k+1)$	Definition of $S_R(k+1)$

19

A Complete Argument

- Recall:
  - $S(n) = 0 + 1 + \ldots + n$
  - $S_R(0) = 0$, $S_R(n) = S_R(n\text{-}1) + n$, $n > 0$
  - $S_C(0) = n(n+1)/2$
- **Property** $P(n)$: $S(n) = S_R(n) = S_C(n)$
- Inductive Step (Continued): Now prove $S_R(k+1) = S_C(k+1)$

$S_R(k+1) = S_R(k) + (k+1)$	Definition of $S_R(k+1)$
$= S_C(k) + (k+1)$	Inductive Hypothesis
$= k(k+1)/2 + (k+1)$	Definition of $S_C(k)$
$= [k(k+1)+2(k+1)]/2 = (k+1)(k+2)/2$	Algebra
$= S_C(k+1)$	Definition of $S_C(k+1)$

- Conclusion: $S(n) = S_R(n) = S_C(n)$ for all $n \geq 0$

20

Induction Requires Recursion

- Either a recursive algorithm is provided
  - Induction used to prove property of algorithm
  - **Example**: Correctness of MergeSort
- Or you must construct a recursive algorithm
  - May not be an actual program; could be a recursive function, or abstract process
  - **Example**: Our "recursive go-between" for $S(n)$, $S_C(n)$
  - Often call this the "inductive" strategy
- Remember
  - Algorithm or strategy: recursion
  - Proof argument: induction

Recursion to be used in a proof only

21

Example With No (Initial) Recursion

- **Claim**: Can make any amount of postage above 8¢ with some combination of 3¢ and 5¢ stamps

- **Property** P(n): You can make n¢ of postage from some combination of 3¢ and 5¢ stamps
- Induction: Prove that it can be done
- Recursion: A strategy that computes the number of 3¢, 5¢ stamps needed

22

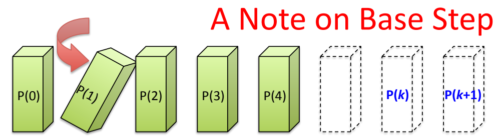
Recursive Strategy

- Given: n¢ of postage
- Returns: amount of 3¢ and amount of 5¢ stamps

```
if (n == 8) {
    return one 3¢, one 5¢
} else {
    Compute answer for (n-1)¢
    Result is p 3¢ stamps, q 5¢ stamps
    if (q > 0) {  // If there is a 5¢ stamp, replace with two 3¢ ones
        return p+2 3¢ stamps, q-1 5¢ stamps
    } else {      // If no 5¢ stamp, must be at least three 3¢ ones
        return p-3 3¢ stamps, q+2 5¢ stamps
    }
}
```

23

A Note on Base Step



- Sometimes want to show a property is true for integers ≥ *b*
- **Intuition**:
  - Knock over domino *b*, and dominoes in front get knocked over
  - Not interested in 0, 1, ..., (*b*-1)
- In general, the base step in induction does not have to be 0
- If base step is some integer *b*
  - Induction proves the proposition for *n* = *b*, *b*+1, *b*+2, ...
  - Does not say anything about *n* = 0, 1, ..., b-1

24

Induction: Base Step

- Given: n¢ of postage
- Returns: amount of 3¢ and amount of 5¢ stamps

n = 8

Base Step: 3¢+5¢ = 8¢

```
if (n == 8) {
    return one 3¢, one 5¢
} else {
    Compute answer for (n-1)¢
    Result is p 3¢ stamps, q 5¢ stamps
    if (q > 0) {  // If there is a 5¢ stamp, replace with two 3¢ ones
        return p+2 3¢ stamps, q-1 5¢ stamps
    } else {      // If no 5¢ stamp, must be at least three 3¢ ones
        return p-3 3¢ stamps, q+2 5¢ stamps
    }
}
```

25

Induction: Inductive Step

- Given: n¢ of postage
- Returns: amount of 3¢ and amount of 5¢ stamps

n = (k+1)

IH: (3p)¢+(5q)¢ = k¢

Inductive Step: (3p+6)¢+(5q-5)¢ = (3p)¢+(5q)¢+1¢ = (k+1)¢

Inductive Step: (3p-9)¢+(5q+10)¢ = (3p)¢+(5q)¢+1¢ = (k+1)¢

```
if (n == 8) {
    return one 3¢, one 5¢
} else {
    Compute answer for (n-1)¢
    Result is p 3¢ stamps, q 5¢ stamps
    if (q > 0) {  // If there is a 5¢ stamp, r...
        return p+2 3¢ stamps, q-1 5¢ stamps
    } else {      // If no 5¢ stamp, must be at least three 3¢ ones
        return p-3 3¢ stamps, q+2 5¢ stamps
    }
}
```

26

Cleaning it Up: Inductive Proof

- **Claim**: You can make any amount of postage above 8¢ with some combination of 3¢ and 5¢ stamps
- Base Step: It is true for 8¢, because 8 = 3 + 5
- Inductive Hypothesis: Suppose true for some k ≥ 8
- Inductive Step:
  - If we used a 5¢ stamp to make *k*, we replace it by two 3¢ stamps. This gives *k*+1
  - If did not use a 5¢ stamp to make *k*, we must have used at least three 3¢ stamps. Replace three 3¢ stamps by two 5¢ stamps. This gives *k*+1.
- Conclusion: Any amount of postage above 8¢ can be made with some combination of 3¢ and 5¢ stamps

27

Alternate Recursive Strategy

- Given: n¢ of postage
- Returns: amount of 3¢ and amount of 5¢ stamps

```
if (n == 8) {
    return one 3¢, one 5¢ stamp
} else if (n == 9) {
    return three 3¢ stamps
} else if (n == 10) {
    return two 5¢ stamps
} else {
    Compute answer for (n-3)¢
    Result is p 3¢ stamps, q 5¢ stamps
    return p+1 3¢ stamps, q 5¢ stamps
}
```

28

Strong Induction

- **Weak induction**
  - P(0): Show that property P is true for 0
  - P(*k*) => P(*k*+1): Show that if property P is true for *k*, it is true for *k*+1
  - Conclude that P(*n*) holds for all *n*
- **Strong induction**
  - P(0), …, P(*m*): Show property P is true for 0 to *m*
  - P(0) and P(1) and … and P(*k*) => P(*k*+1): Show that if P is true for numbers less than or equal to *k*, then it is true for *k*+1
  - Conclude that P(*n*) holds for all *n*
- Both proof techniques are equally powerful

29

Strong Induction: Base Step

- Given: n¢ of postage
- Returns: amount of 3¢ and amount of 5¢ stamps

n = 8, 9, 10

Base Step (part 1): 3¢+5¢ = 8¢

Base Step (part 2): 3¢+3¢+3¢ = 9¢

Base Step (part 3): 5¢+5¢ = 10¢

```
if (n == 8) {
    return one 3¢, one 5¢ stamp
} else if (n == 9) {
    return three 3¢ stamps
} else if (n == 10) {
    return two 5¢ stamps
} else {
    Compute answer for (n-3)¢
    Result is p 3¢ stamps, q 5¢ stamps
    return p+1 3¢ stamps, q 5¢ stamps
}
```

30

Strong Induction: Inductive Step

- Given: n¢ of postage
- [n = k+1] ...s: amount of 3¢ and amount of 5¢ stamps

```
if (n == 8) {
    return one 3¢, one 5¢ stamp
} else if (n == 9) {
    return three 3¢ stamps
} else if (n == 10) {
    return two 5¢ stamps
} else {
    Compute answer for (n-3)¢
    Result is p 3¢ stamps, q 5¢ stamps
    return p+1 3¢ stamps, q 5¢ stamps
}
```

Strong Induction Hypothesis: Strategy works for any amount of postage *m*, where $8 \le m \le k$

SIH: (3p)¢+(5q)¢ = (k-2)¢

Inductive Step: (3p+3)¢+(5q)¢ = (k+1)¢

31

Clean Up: Strong Inductive Proof

- **Claim**: You can make any amount of postage above 8¢ with some combination of 3¢ and 5¢ stamps
- Base Step: We consider three base cases: 8¢, 9¢, and 10¢
  - It is true for 8¢, since 3+5 = 8
  - It is true for 9¢, since 3+3+3 = 9
  - It is true for 10¢, since 5+5 = 10
- (Strong) Inductive Hypothesis: Suppose there is some *k* such that claim is true for all numbers *m*, where $8 \le m \le k$
- Inductive Step: As $8 \le k-2 \le k$, make postage for (k-2)¢ and add a 3¢ stamp. This gives answer for (k+1)¢.
- Conclusion: Any amount of postage above 8¢ can be made with some combination of 3¢ and 5¢ stamps

32

Merge Sort: Correctness Idea

Property P(*n*): For any array of length *n* as input, **mergeSort**() sorts the contents in place

```
private void mergeSort(T[] x, int lo, int hi, T[] y) {
   // base case
   if (hi <= lo + 1) return; // nothing to do

   // at least 2 elements
   // split and recursively sort
   int mid = (lo + hi)/2;
   mergeSort(x, lo, mid, y);
   mergeSort(x, mid, hi, y);
   // merge sorted sublists
   merge(x, lo, mid, hi, y);
}
```

```
/**
 * Merge 2 subarrays of x, using y as temp
 */
...e(T[] x, int lo, int mid, int hi,
        T[] y) {
   ... // subarray pointers
   int j = mid;
   int k = lo; // destination pointer

   ... hi) {
      ...areTo(x[j]) > 0)? x[j++] :
                          x[i++];

   // one of the subarrays is empty
   // copy remaining elements from the other
   System.arraycopy(x, i, y, k, mid - i);
   System.arraycopy(x, j, y, k, hi - j);
   // now copy everything back to original array
   System.arraycopy(y, lo, x, lo, hi - lo);
}
```

Any array of length 0 or 1

Base Step: Array of 1 or 0 elements is already sorted

33

Merge Sort: Correctness Idea

Property P(*n*): For any array of length *n* as input, **mergeSort**() sorts the contents in place

```
private void mergeSort(T[] x, int lo, int hi, T[] y) {
   // base case
   if (hi <= lo + 1) return; // nothing to do

   // at least 2 elements
   // split and recursively sort
   int mid = (lo + hi)/2;
   mergeSort(x, lo, mid, y);
   mergeSort(x, mid, hi, y);
   // merge sorted sublists
   merge(x, lo, mid, hi, y);
}
```

```
/**
 * Merge 2 subarrays of x, using y as temp
 */
...e(T[] x, int lo, int mid, int hi,
        T[] y) {
   ... // subarray pointers
   int j = mid;
   int k = lo; // destination pointer

   while (i < mid && j < hi) {
      y[k++] = (x[i].compareTo(x[j]) > 0)? x[j++] :
                                           x[i++];
   ...
   ...s empty
   ...ents from the other
   ..., k, mid - i);
   System.arraycopy(x, j, y, k, hi - j);
   ...back to original array
   ... lo, hi - lo);
```

Any array of length k+1

Strong Inductive Hypothesis: For any array of length $m \le k$, **mergeSort**() sorts the array

Inductive Step: Show that **merge**() takes two sorted halves and produces a single sorted array of length *k*+1

34

Induction and Recursion

Recursion:
- A strategy used to solve a problem or compute result
- Result is algorithm/program

Provides the "Inductive" Strategy

Guides the Algorithm Development

Induction:
- Prove statements about the natural numbers
- Prove properties of recursive algorithms
- Result is a proof/argument

35

Summary of Today

- Induction is a technique to prove statements
  - Recursion is a strategy to construct algorithms
  - Useful for program correctness and complexity
- But all induction requires a recursive strategy
  - Hard part is finding the strategy
  - Afterwards, induction is often straightforward
  - Different variations of induction exist to tailor to your recursive strategy

36

To Think About for Next Time

```
/**
 * Sorts the Comparable array x between lo
 * (inclusive) and hi (exclusive), recursively and
 * in O(n log n) time
 */
private void mergeSort(T[] x, int lo, int hi, T[] y) {
   // base case
   if (hi <= lo + 1) return; // nothing to do

   // at least 2 elements
   // split and recursively sort
   int mid = lo+1;
   mergeSort(x, lo, mid, y);
   mergeSort(x, mid, hi, y);
   // merge sorted sublists
   merge(x, lo, mid, hi, y);
}
```

What does this do to complexity?

```
/**
  * Merge 2 subarrays of x, using y as temp
  */
private void merge(T[] x, int lo, int mid, int hi,
                   T[] y) {
   int i = lo; // subarray pointers
   int j = mid;
   int k = lo; // destination pointer

   while (i < mid && j < hi) {
      y[k++] = (x[i].compareTo(x[j]) > 0)? x[j++] :
                                           x[i++];
   }

   one of the subarrays is empty
   copy remaining elements from the other
   System.arraycopy(x, i, y, k, mid - i);
   System.arraycopy(x, j, y, k, hi - j);
   // now copy everything back to original array
   System.arraycopy(y, lo, x, lo, hi - lo);
}
```

37