Proof Techniques

- Why we need proof methods
- Proof by contraposition
- Proof by contradiction
- Recursion: review
- Mathematical induction
- Fibonacci analysis

Some slides adapted from Fall 2010 CS251: Prof Luo Si

Why we need proof methods

Mathematical proofs all have their basis in formal logic.

- There are some basic techniques for proving things, on the basis of different logical truths.

Why are proofs really necessary?

- Because intuition and guessing don't always work.

It is often hard to exhaustively validate claims with many cases (e.g., infinite)

- Example: $1 + 2 \ldots + n\text{-}1 + n = n(n+1)/2$ for all n as an integer.

Intuition may not be correct.

- Example: $n! < n^2$ is easy to show for $n=1,2,3$, but what about for larger n?

Proof by direct derivation

Use existing knowledge as derivative rules to generate new knowledge

- A and (A⇒B) ⇒ B

Example:

- Assume we know the sum of angles in a triangle is 180.
- What about the sum of angles in a rectangle?

However, proof by direct derivation can be difficult

Proof by reasoning tricks

Problem:

- In a tennis tournament, players are eliminated in each round as only the winner of a game goes to the next round until the final round and only one winner gets the trophy.

Claim:

- If there are n players, prove that there are exactly n – 1 games played.

Proof:

- Everyone except the champion loses exactly 1 game.
- There are n – 1 non-champions.
- Thus, there are n – 1 games.

Proof by contraposition

Instead of proving a difficult original claim as itself, we can prove the contrapositive:

- Original: $p \Rightarrow q$ Contrapositive: $\neg q \Rightarrow \neg p$

Example:

- Prove for all possible integers, if n^2 is odd, then n is odd.
- Contrapositive: Prove that if n is even, then n^2 is even.
  - Assume that n is even.
  - $n = 2k$
  - Then $n^2 = 4k^2 = 2(2k^2)$ which, by definition, is even.

Proof by contradiction

Show that if we assume a claim is wrong, then we will find contradiction with what we know.

- Original: $p \Rightarrow q$
- Contradiction:
  - Use $p \wedge \neg q \Rightarrow F$
  - To show $p \Rightarrow q$

Example:

- Claim: If $3x = x$, then $x = 0$.
- If the claim is wrong, then $3x = x$ is true for some $x \neq 0$

- Divide both sides by x (since $x \neq 0$)
- This implies $3=1$, which contradicts existing knowledge
- Therefore, if $3x=x$ then $x=0$.

Proof by contradiction

Example 2:

- Claim: There is no positive integer solution to the equation $x^2-y^2=1$
- If the claim is wrong, there is a solution solution (x_0, y_0) such that x_0 and y_0 are both positive integers
- Therefore $(x_0-y_0)(x_0+y_0)=1$
- ……

Example 3:

- Claim: For any integers a and b, $a+b \geq 15$ implies that $a \geq 8$ or $b \geq 8$
- If the claim is wrong, there are solutions a and b such that both are integers and both are smaller than 8
- ……

- Why we need proof methods
- Proof by contraposition
- Proof by contradiction
- *Recursion: review*
- Mathematical induction
- Fibonacci analysis

Recursion

What is recursion?

- When one function calls itself directly or indirectly.

Why learn recursion?

- New mode of thinking.
- Powerful programming paradigm.

Many computations are naturally self-referential.

- Mergesort, FFT, gcd.
- Linked data structures.
- A folder contains files and other folders.

Recursive definitions

A recursive definition is a definition with two parts:

- A set of base cases - simple cases that are defined explicitly
- A recursive definition – where additional cases are defined in terms of previous cases.

Example recursive function S that defines the set {2, 4, 8, 16, 32, …}:

- $S(1) = 2$ base case
- $S(n) = 2S(n - 1)$ for $n \geq 2$

It is often convenient to define objects recursively because it is then easier to develop algorithms for such objects by using **recursion**.

- Recursive definitions lend themselves to proof by mathematical induction

How to write simple recursive programs?

Base case. Reduction step.

Trace the execution of a recursive program.

- Every possible chain of recursive calls must eventually reach a base case, and the handling of each base case should not use recursion.

Classic example--the factorial function:

- $n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot (n-1) \cdot n$

Recursive definition:

- $f(0) = 1$ base case
- $f(n) = n \cdot f(n-1)$ for $n > 0$

As a Java method:

```
public static int recursiveFactorial(int n) {
    if (n == 0) return 1;                          // base case
    else return n * recursiveFactorial(n- 1);      // recursive case
```

Linear recursion

Recur once per method

- Perform a single recursive call
- This step may have a test that decides which of several possible recursive calls to make, but it should ultimately make just one of these calls
- Define each possible recursive call so that it makes progress towards a base case.

Example:

- Reversing an array
  - Input: An array A and nonnegative integer indices i and j
  - Output: Reversal of the elements in A starting at index i and ending at j

```
public static void ReverseArray(A, i, j):
 if i < j then
   Swap A[i] and A[ j]
   ReverseArray(A, i + 1, j - 1)
 return
```

Defining arguments for recursion

In creating recursive methods, it is important to define the methods in ways that facilitate recursion.

This sometimes requires we define additional parameters that are passed to the method.

- For example, we defined the array reversal method as ReverseArray(A, i, j), not ReverseArray(A).

Tail recursion

Tail recursion occurs when a linearly recursive method makes its recursive call as its last step.

- The array reversal method is an example.
- Such methods can be easily converted to non-recursive methods (which saves on some resources).

Example:

- Input: An array A and nonnegative integer indices i and j
- Output: Reversal of the elements in A starting at index i and ending at j

```
public static void IterativeReverseArray(A, i, j ):
    while i <  j do
        Swap A[i ] and A[ j ]
        i  = i + 1
        j  = j - 1
    return
```

Binary recursion

Binary recursion occurs whenever there are two recursive calls for each non-base case.

Example

- Problem: add all the numbers in an integer array A:
  - Input: An array A and integers i and n
  - Output: The sum of the n integers in A starting at index i

```
public static int BinarySum(A, i, n):
    if n = 1 then
        return A[i ]
    return BinarySum(A, i, n/2) + BinarySum(A, i + n/2, n/2)
```

- Why we need proof methods
- Proof by contraposition
- Proof by contradiction
- Recursion: review
- **Mathematical induction**
- Fibonacci analysis

Mathematical induction

Used to establish that a claim is true for all natural numbers based on creating a one to one correspondence with the integers.

- Show that the first among an infinite sequence of claims is true.
- Then prove that if any one claim in the sequence is true, the subsequent must also be true.

Steps:

- **Basis:** Claim is true for simple cases (e.g., $n = 1$)
- **Assumption**: Claim is true for some integer $n \geq 1$.
- **Induction**: Based on the assumption, show the claim must be true for $n + 1$.
- **Conclusion**: Since claim is true for the base case, must be true for all subsequent cases.

Mathematical induction

Can be thought of as proof by the domino effect.

For any domino presented in a long row of dominoes, it will fall if we assume:

- The first domino falls
- When a domino falls, the next one also falls

Mathematical induction example

Claim:

- For all $n > 0$, the sequence $1 + 2 \ldots + n\text{-}1 + n = n(n+1)/2$.

Proof:

- Basis ($n=1$): $1 = 1(1+1)/2$

- Induction:
  - Assume claim is true for any n', i.e.:
    $1+..+n' = n'(n'+1)/2$
  - Show that it must be true for n'+1 as well, i.e.:
    $n'(n'+1)/2 + (n'+1) = (n'+1)(n'+2)/2$

- Conclusion:
  - Thus it must be true for all $n > 0$

Mathematical induction example

Claim:

- 1 / (1*3) + 1 / (3*5) … + 1 / [(2n-1)(2n+1)] = n/(2n+1)

Proof:

- Basis (n=1): 1 / (1*3) = 1 / (2*1+1)

- Induction:
  - Assume this is correct for n'…
  - Show for the case of n'+1…

- Conclusion:
  - Thus it must be true for all n > 0

Mathematical induction example

Claim:

- Consider the Fibonacci function F(n), where:
  - $F(n) = F(n-1) + F(n-2)$ for $n \geq 2$, and $F(0)=0$, $F(1)=1$
- Then $F(n) < 2^n$

Proof:

- Basis (n=1): $F(0) < 2^0$
- Basis (n=2): $F(1) < 2^1$

- Induction:
  - Assume this is correct for all values $\leq n'$…
  - Show for the case of $n'+1$…

- Conclusion:
  - Thus it must be true for all $n > 0$

Note: when claim is assumed true for all values $\leq n'$ it is called weak induction;
when assumed true for only n' it is called strong induction

- Why we need proof methods
- Proof by contraposition
- Proof by contradiction
- Recursion: review
- Mathematical induction
- Fibonacci analysis

Computing Fibonacci numbers

Recursive algorithm (first attempt):

- Input: Nonnegative integer k
- Output: The kth Fibonacci number F_k

```
public static int BinaryFib(k):
  if k ≤ 1 then
    return k
  else
    return BinaryFib(k - 1) + BinaryFib(k - 2)
```

Analysis

Let n_k be the number of recursive calls by BinaryFib(k)

- $n_0 = 1$
- $n_1 = 1$
- $n_2 = n1 + n0 + 1 = 1 + 1 + 1 = 3$
- $n_3 = n2 + n1 + 1 = 3 + 1 + 1 = 5$
- $n_4 = n3 + n2 + 1 = 5 + 3 + 1 = 9$
- $n_5 = n4 + n3 + 1 = 9 + 5 + 1 = 15$
- $n_6 = n5 + n4 + 1 = 15 + 9 + 1 = 25$
- $n_7 = n6 + n5 + 1 = 25 + 15 + 1 = 41$
- $n_8 = n7 + n6 + 1 = 41 + 25 + 1 = 67$

Note that n_k at least doubles every other time

- That is, $n_k > 2^{k/2}$.
- This is **exponential** growth, which is even larger than quadratic

A better Fibonacci algorithm

Use linear recursion instead

- Input: A nonnegative integer k
- Output: Pair of Fibonacci numbers (F_k, F_{k-1})

```
public static int[] LinearFibonacci(k):
if k = 1 then
    return [k, 0]
else
    [i, j] = LinearFibonacci(k − 1)
return [i +j, i]
```

LinearFibonacci makes k−1 recursive calls