Recursion

CSCI 170 Spring 2021

Sandra Batista

1.1-1.2

Induction

- **Induction and Recursive Functions**
- Loop Invariants
- Algorithmic Correctness
- Well-Ordering Principle

Recursion

Defining an object, mathematical function, or computer function in terms of *itself*

Problem in which the solution can be expressed in terms of itself (usually a smaller instance/input of the same problem) ***and a base/terminating case***

Input to the problem must be categorized as a:

- Base case: Solution known beforehand or easily computable (no recursion needed)
- Recursive case: Solution can be described using solutions to smaller problems of the same type

Keeping putting in terms of something smaller until we reach the base case

Factorial: n! = n * (n-1) * (n-2) * … * 2 * 1

- n! = n * (n-1)!
- Base case: n = 1, n= 0 $1! = 1$ $0! = 1$
- Recursive case: n > 1 => n*(n-1)!
- Assume n is a non-negative integer

n = Non-Negative Integers and is defined as:

- The number 0 [Base]
- n + 1 where n is some non-negative integer [Recursive]

$2 = [[0]_0 + 1]_1 + 1]_2 = 2$

A string is

- Empty string, ε [Base] $\lambda = ""$
- String concatenated with a character (e.g. 'a'-'z') [Recursive]

"cat" $= [[[\lambda]_{""} + 'c']_{"c"} + 'a']_{"ca"} + 't']_{"cat"}$

Palindrome (string that reads the same forward as backwards)

- Example: dad, peep, level
- Defined as:

  Empty string [Base]

  Single character [Base]

  xPx where x is a character and P is a Palindrome [Recursive]

peep

λ
↓
ee
↓
peep

Base case is empty string

Base case is single character

v
↓
eve
↓
level

Can you write a function to check if a string is a palindrome?

Steps to Formulating Recursive Solutions

1. Write out some solutions for a few input cases to discover how the problem can be decomposed into smaller problems of the same form
   - Does solving the problem on an input of smaller value or size help formulate the solution to the larger
2. Identify the base case(s)
3. Identify how to combine the small solution(s) to solve the larger problem

Base cases for palindrome:
if even length base case is empty string
if odd length base case is a single character

Recursive case: check if 1st and last character are the same. if not → not palindrome
if so we need to check if substring w/ 1st and

Induction and Recursive Function

Similarities between proof by induction and recursive functions:

- Base cases in proofs by induction often correspond to base cases in recursive functions.
- Bases cases are cases for which no inductive hypotheses may be used and no recursive calls may be made.
- The inductive step corresponds to a recursive function call.
- In the inductive step, the solution from a smaller case of the problem assumed to be true by the inductive hypothesis is used to solve the problem for the inductive step.
- In a recursive function call, a solution of the same problem on a smaller input size is used to solve the recursive call

Recursive Factorial Code

Factorial of n, n!=n x (n-1) x (n-2) x …x 2 x 1
Note 0! = 1.

```
int factorial(int n) {
    if (n < 0) return -1; /* error checking */
    if (n == 0 || n == 1) return 1;
    return n * factorial(n-1);
}
```

$\forall n \geq 0 \; P(n)$:

Claim: For any n greater than or equal to 0, factorial(int n) code computes n!
Proof by induction:

Base cases: $P(0)$: returns $1 = 0!$ ✓ ($n=0$)

$P(1)$: returns $1 = 1!$ ✓ ($n=1$)

} by inspection

Recursive Factorial Code

Factorial of n, n!=n x (n-1) x (n-2) x …x 2 x 1
Note 0! = 1.

```
int factorial(int n) {
    if (n < 0) return -1; /* error checking */
    if (n == 0 || n == 1) return 1;
    return n * factorial(n-1);
}
```

Proof by induction (continued):

$\forall_{1 \le k \le n} P(k)$

Strong IH: Assume $P(0), P(1), \ldots, P(n)$ specific $n \ge 1$

Inductive step: $P(n+1)$ holds

factorial(n+1) returns (n+1) times result of call to factorial(n) by inspection. Since P(n) holds by IH, factorial(n) returns n!. That means factorial(n+1) returns $(n+1) * n! = (n+1)!$ $\Rightarrow P(n+1)$ holds

Recursive Fibonacci Code

Fibonacci sequence: $f_0=0$, $f_1=1$ and $f_n= f_{n-1} + f_{n-2}$

```
int fibonacci(int n) {
    if (n < 0) return -1; /* error checking */
    if (n == 0 || n == 1) return n;   // base case
    return fibonacci(n-1)+ fibonacci(n-2);   // recursive
}
```

$\forall n \geq 0 \; P(n)$

Claim: For any n greater than or equal to 0, P(n) = fibonacci(int n) code returns the n-th Fibonacci number.

Proof by induction:

Base cases: $P(0)$: $n=0$ return $0 = f_0$ ✓

$P(1)$: $n=1$ return $1 = f_1$ ✓ > by inspection

Assume $P(0) \wedge P(1) \wedge \ldots \wedge P(n)$ for some specific $n \geq 1$. (strong IH)

Recursive Fibonacci Code

Fibonacci sequence: $f_0=0$, $f_1=1$ and $f_n = f_{n-1} - f_{n-2}$

```
int fibonacci(int n) {
    if (n < 0) return -1; /* error checking */
    if (n == 0 || n == 1) return ;
    return fibonacci(n-1)+ fibonacci(n-2);
}
```

(abbreviation: fib(n) = fibonacci(n))

Proof by induction (continued): Inductive step: Show P(n+1) holds

fib(n+1) returns fib(n) and fib(n-1) by inspection.

By IH P(n) and P(n-1) holds ⇒ fib(n) returns f_n and fib(n-1) returns f_{n-1}.

⇒ fib(n+1) returns $f_n + f_{n-1} = f_{n+1}$ by def of Fibonacci sequence ⇒ P(n+1) holds.

Induction

- Induction and Recursive Functions
- **Loop Invariants**
- Algorithmic Correctness
- Well-Ordering Principle

Recursive Functions and Iteration

All recursive code examples we have given can be implemented again using iteration

How does have a recursive definition help us with iterative implementations?

Let's consider the number of times the inductive cases are being applied with an iterative example.

Iterative Factorial Code

Factorial of n, n!=n x (n-1) x (n-2) x …x 2 x 1
Note 0! = 1.

```
int iterative_factorial(int n){
    int x = 1, i =1;
   if (n == 0) return x;   /* base case*/
   while (i <= n) {                /*recursive cases or inductive steps*/
     x *= i;
     i++;
   }
   return x;
}
```

$P(0) \rightarrow P(1) \rightarrow P(2) \rightarrow P(3)$

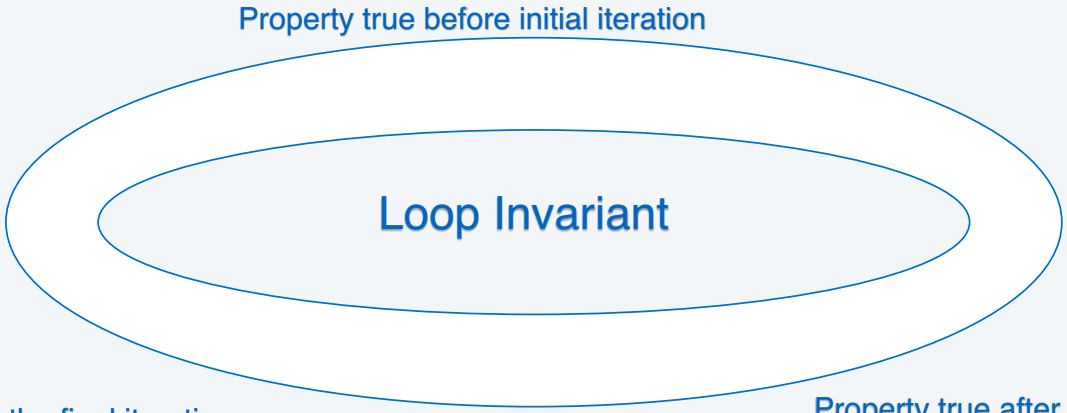
$\forall\ n \geq 0$

Claim: For any n greater than or equal to 0, iterative_factorial(int n) code computes n!

Loop Invariant

A **loop invariant** is a predicate, P(i), that must be true before a loop is first executed, after each iteration, and after the final iteration. The variable i is used to count the number of iterations.

Specify the loop invariant by stepping through the code or description of an algorithm to see what property must be true before the first iteration, after each iteration, and after the final iteration.



Iterative Factorial Code

iterative_factorial(4)

n = 4

Factorial of n, n!=n x (n-1) x (n-2) x …x 2 x 1
Note 0! = 1.

```
int iterative_factorial(int n){
    int x = 1, i =1;
   if (n == 0) return x;
   while (i <= n) {
      x *= i;
      i++;
   }
   return x;
}
```

Before iteration →

i	x	?!
1	1	0!
2	1	1!
3	2	2!
4	6	3!
5	24	4!

What is the loop invariant?

Iterative Factorial Code

Factorial of n, n!=n x (n-1) x (n-2) x …x 2 x 1
Note 0! = 1.

```
int iterative_factorial(int n){
     int x = 1, i =1;
    if (n == 0) return x;
    while (i <= n) {
       x *= i;
       i++;
    }
    return x;
}
```

$\forall\ i \geq 1 \quad P(i): x = (i-1)!$

$(1 \leq i \leq n+1)$

Loop invariant: For any i greater than or equal to 1, P(i): The variable x holds the value (i-1)!

Iterative Fibonacci Code

Fibonacci sequence: $f_0=0$, $f_1=1$ and $f_n= f_{n-1} + f_{n-2}$

```
int iterative_fibonacci(int n){
   If (n =0 or n = 1) return n;   /* base cases*/
   x = 0; y = 1; i =1;
   while (i < n) {                    /*recursive cases*/
      i++;
      z = x + y;
      x = y;
      y = z;
   }
   return y;
}
```

$i = 1 \quad x = 0 = f_0 \quad y = 1 = f_1$

$i = 2$

$z = x + y = f_0 + f_1 = f_2$

$x = y = f_1$

$y = z = f_2$

What is the loop invariant?

Iterative Fibonacci Code

Fibonacci sequence: $f_0=0$, $f_1=1$ and $f_n= f_{n-1} – f_{n-2}$

```
int iterative_fibonacci(int n){
    If (n =0 or n = 1) return n;   /* base cases*/
    x = 0; y = 1; i =1;
   while (i < n) {                    /*recursive cases*/
       i++;
       z = x + y;
       x = y;
       y = z;
   }
   return y;
}
```

$\forall i \geq 1 \quad y = f_i \wedge x = f_{i-1} \wedge (1 \leq i \leq n)$

Loop Invariant: For any i greater than or equal to 1, P(i): y equals the i-th Fibonacci number, f_i, and x equals the (i-1)-th Fibonacci number, f_{i-1}.

Iterative Linear Search

Given an array of numbers A[1,…,n], search for if it contains value x
Using 1 based indexing (not 0). If the array contains x, return its position. Otherwise return 0 for not found.

```
int Search(A,x){
    int i = 1;
    for (i=1; i<= n; i++){
        if (A[i] == x) return i;
    }
    return 0
}
```

A = [1, 5, 9, 7]

X = 6

X = 5

break out of for loop

What is the loop invariant?

.

Iterative Linear Search

Given an array of numbers A[1,…,n], search for if it contains value x
Using 1 based indexing (not 0). If the array contains x, return its position. Otherwise return 0 for not found.

```
int Search(A,x){
    int i = 1;
    for (i=1; i<= n; i++){
        if (A[i] == x) return i;
    }
    return 0
}
```

$\forall i \geq 1$

Loop Invariant: For any i greater than or equal to 1, P(i): $1 \leq i \leq n+1$ and x is not in A[1,…, i-1]

Induction

- Induction and Recursive Functions
- Loop Invariants
- **Algorithmic Correctness**
- Well-Ordering Principle

Showing Algorithmic Correctness Using Loop Invariants

1. State the loop invariant. The loop invariant is a property that must be true before a loop is first executed, after each iteration, and after the final iteration. The variable i is used to count the number of iterations.

2. Prove the loop invariant is true by induction over the number of iterations.

3. If an algorithm requires a loop to execute k times and if upon exiting the loop the value of the iteration variable is (k+1), show that P(k+1) ensures correctness for the entire algorithm.

Iterative Factorial Code Correctness

Factorial of n, n!=n x (n-1) x (n-2) x …x 2 x 1
Note 0! = 1.

```
int iterative_factorial(int n){
     int x = 1, i =1;
    if (n == 0) return x;   /* base case*/
    while (i <= n) {                    /*recursive cases or inductive steps*/
       x *= i;
       i++;
    }
    return x;
}
```

$\forall n \geq 0$

Claim: For any n greater than or equal to 0, iterative_factorial(int n) code computes n!

$\forall i \geq 1$

1. State the Loop invariant: For any i greater than or equal to 1, P(i): The variable x holds the value (i-1)!

$(1 \leq i \leq n+1)$

2. Prove Loop Invariant by induction over i.

Iterative Factorial Code Correctness

```
Factorial of n, n!=n x (n-1) x (n-2) x …x 2 x 1
Note 0! = 1.

int iterative_factorial(int n){
   int x = 1, i =1;
   if (n == 0) return x;   /* base case*/
   while (i <= n) {                /*recursive cases or inductive steps*/
      x *= i;
      i++;
   }
   return x;
}
```

$\forall\, i \geq 1 \qquad x = (i-1)! \qquad (\wedge\ 1 \leq i \leq n+1)$

Loop invariant: For any i greater than or equal to 1, P(i): The variable x holds the value (i-1)!

Proof by induction over i:

Base case: $i = 1$ $P(1)$ $i = 1$ $x = 1$

$= 0!$

$= (1-1)!$ ✓ by inspection

Assume $\forall\, 1 \leq i \leq k$ $P(i)$ for some specific $k \geq 1$ (Strong IH)

$k < n+1$

$P(1) \wedge P(2) \wedge \ldots \wedge P(k)$

Iterative Factorial Code Correctness

Factorial of n, n!=n x (n-1) x (n-2) x …x 2 x 1
Note 0! = 1.

```
int iterative_factorial(int n){
    int x = 1, i =1;
   if (n == 0) return x;   /* base case*/
   while (i <= n) {                /*recursive cases or inductive steps*/
     x *= i;
     i++;
   }
   return x;
}
```

x = x * i ;

Loop invariant: For any i greater than or equal to 1, P(i): The variable x holds the value (i-1)!
Proof by induction over i:

Inductive step: P(k+1) holds
Since P(k) holds by IH ⇒ i = k and x = (k-1)!
By inspection x = i · x = k · (k-1)! = k! ✓
and i is incremented so i = k+1 ✓
⇒ P(k+1) holds

Iterative Factorial Code Correctness

Factorial of n, n!=n x (n-1) x (n-2) x …x 2 x 1
Note 0! = 1.

```
int iterative_factorial(int n){
    int x = 1, i =1;
   if (n == 0) return x;   /* base case*/
   while (i <= n) {                  /*recursive cases or inductive steps*/
      x *= i;
      i++;
   }
   return x;
}
```

3. The code must run the loop n times and ends the loop when i = n+1. Show P(n+1) ensures the correctness of the entire function.

If P(n+1) holds ⇒ $i = n+1 \wedge x = n!$

code returns x (by inspection)

and since $x = n!$ code returns $n!$

Iterative Fibonacci Code Correctness

Fibonacci sequence: $f_0=0$, $f_1=1$ and $f_n= f_{n-1} - f_{n-2}$

```
int iterative_fibonacci(int n){
   If (n =0 or n = 1) return n;   /* base cases*/
   x = 0; y = 1; i =1;
  while (i < n) {                    /*recursive cases*/
     i++;
     z = x + y;
     x = y;
     y = z;
  }
  return y;
}
```

$\forall n \geq 0$

Claim: For any n greater than or equal to 0, iterative_fibonacci returns the n-th Fibonacci number.

1. State the Loop Invariant: For any i greater than or equal to 1, P(i): y equals the i-th Fibonacci number, f_i, and x equals the (i-1)-th Fibonacci number, f_{i-1}.

$\forall i \geq 1$

2. Prove the loop invariant using induction over i.

Iterative Fibonacci Code Correctness

Fibonacci sequence: $f_0=0$, $f_1=1$ and $f_n = f_{n-1} + f_{n-2}$

```
int iterative_fibonacci(int n){
   If (n =0 or n = 1) return n;   /* base cases*/
   x = 0; y = 1; i =1;
   while (i < n) {                    /*recursive cases*/
      i++;
      z = x + y;
      x = y;
      y = z;
   }
   return y;
}
```

$\forall\ i \geq 1 \quad y = f_i \ \wedge\ x = f_{i-1}\ (\wedge\ 1 \leq i \leq n)$

Loop Invariant: For any i greater than or equal to 1, P(i): y equals the i-th Fibonacci number, f_i, and x equals the (i-1)-th Fibonacci number, f_{i-1}.

Prove by induction over i:

Base case: $i=1$ $P(1)$ $i=1$ $x = 0 = f_0$ $y = 1 = f_1$ by inspection

Strong IH Assume $\forall\ 1 \leq i \leq k\ P(i)$ for some specific $k \geq 1$

$P(1) \wedge P(2) \wedge \dots \wedge P(k)$ $k < n$

Iterative Fibonacci Code Correctness

Fibonacci sequence: $f_0=0$, $f_1=1$ and $f_n = f_{n-1} - f_{n-2}$

```
int iterative_fibonacci(int n){
   If (n =0 or n = 1) return n;   /* base cases*/
   x = 0; y = 1; i =1;
  while (i < n) {                 /*recursive cases*/
    i++;
    z = x + y;
    x = y;
    y = z;
  }
  return y;
}
```

Loop Invariant: For any i greater than or equal to 1, P(i): y equals the i-th Fibonacci number, f_i, and x equals the (i-1)-th Fibonacci number, f_{i-1}.

Prove by induction over i: Inductive step: Show P(k+1) holds

Since P(k) holds, $i = k$ ∧ $x = f_{k-1}$ ∧ $y = f_k$.

By inspection i) since i is incremented $i = k+1$ ✓

ii) $z = x + y = f_{k-1} + f_k = f_{k+1}$ by def of Fibonacci sequence

iii) $x = y = f_k$ iv) $y = z = f_{k+1}$ ⇒ P(k+1) holds

Iterative Fibonacci Code Correctness

Fibonacci sequence: $f_0=0$, $f_1=1$ and $f_n = f_{n-1} - f_{n-2}$

```
int iterative_fibonacci(int n){
   If (n =0 or n = 1) return n;   /* base cases*/
   x = 0; y = 1; i =1;
  while (i < n) {                 /*recursive cases*/
     i++;
     z = x + y;
     x = y;
     y = z;
  }
  return y;
}
```

3. The code must run the loop n-1 times and ends the loop when i = n. Show P(n) ensures the correctness of the entire function.

If P(n) holds then $i = n \wedge x = f_{n-1} \wedge y = f_n$

code by inspection returns $y = f_n$, nth fibonacci number.

Iterative Linear Search Correctness

Given an array of numbers A[1,…,n], search for if it contains value x
Using 1 based indexing (not 0). If the array contains x, return its position. Otherwise return 0 for not found.

```
int Search(A,x){
    int i = 1;
    for (i=1; i<= n; i++){
        if (A[i] == x) return i;
    }
    return 0
}
```

Claim: Search(A,x) returns index of element x if it is found and 0 it it is not found.

Complete as an exercise:

$\forall\ i \geq 1$

1. State the loop invariant: For any i greater than or equal to 1, P(i): $1 \leq i \leq n+1$ and x is not in A[1,…, i-1]
2. Prove the loop invariant by induction over i.
3. The code must make at most n iterations. Show that P(n+1) ensures the correctness of the code.

Induction

- Induction and Recursive Functions
- Loop Invariants
- Algorithmic Correctness
- **Well-Ordering Principle**

Well-ordering Principle

Well-ordering principle: Every nonempty set of nonnegative integers has a smallest element.

To show that for any n greater than or equal to 0, P(n) holds using the well-ordering principle:

1. Define a set of counterexamples, C = {n | P(n) false}
2. Assume C is nonempty and use proof by contradiction.
3. By the well-ordering principle, there exists a smallest element, a*, in C.
4. Show any contradiction.
5. Conclude that C must be empty, i.e. no counterexample exists.

Well-ordering Principle Proof Example

Show for any n greater than or equal to 0, P(n): $\forall n \geq 0$ $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$

Proof by contradiction using the well-ordering principle:

Let C be the set of counterexamples, i.e. {n: P(n) is false}. Assume C is non-empty.

By the well-ordering principle there must be a least (or smallest) integer in C. Let n^* be the least integer such that P(n^*) is false, i.e. the smallest integer that is a counterexample to the claim.

Can n^* be 0?

$P(0) = \sum_{i=0}^{0} i = 0 = \frac{0(0+1)}{2} = 0$ ✓ true

$\Rightarrow n^* > 0$

Well-ordering Principle Proof Example

Show for any n greater than or equal to 0, P(n): $\sum_{i=0}^{n} i =$

Proof by contradiction using the well-ordering principle (continued):

The least integer for which the claim is false is n* > 0.

Let's consider what occurs for n*-1. Since n* is the first integer such that P(n*) is false, P(n*-1) must be true.

Accordingly, let's consider what this implies for n* since the claim holds for n*-1.

$P(n^*) = \sum_{i=0}^{n^*} i = \sum_{i=0}^{n^*-1} i + n^* = \frac{(n^*-1)n^*}{2} + n^* = \frac{n^*(n^*+1)}{2}$

since $P(n^*-1)$ true

Algebra check ✓

$\frac{(n^*-1)n^*}{2} + \frac{2n^*}{2} = \frac{n^*(n^*-1+2)}{2} =$

$\Rightarrow\Leftarrow$ n^* is not a counterexample since $P(n^*)$

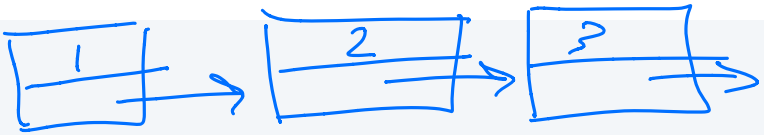
$\Rightarrow C = \emptyset \Rightarrow$ statement is true

Recursive Definitions and the Well-ordering Principle

Suppose we want to give a recursive definition of a set of objects. The well-ordering principle implies that there will be a smallest object.

The smallest values or objects represent the base cases in recursive or inductive definitions.

Examples:



Linked Lists : Base case: empty list of size 0
Recursive case: Linked list with an element added to it

Strings: Base case: empty string λ ε ""
Recursive case: String with a character added to it. "a" = "" a

Fibonacci sequence: Base case: $f_0=0$, $f_1=1$ or ($f_1=1$ $f_2=1$)
Recursive case: $f_n = f_{n-1} + f_{n-2}$

Induction

- Induction and Recursive Functions
- Loop Invariants
- Algorithmic Correctness
- Well-Ordering Principle