CS221: Algorithms and Data Structures Lecture #2 (Tail) Recursion, Induction, Loop Invariants, and Call Stacks

Steve Wolfman

2014W1

Today's Outline

- Thinking Recursively
- Recursion Examples
- Analyzing Recursion: Induction and Recurrences
- Analyzing Iteration: Loop Invariants
- Mythbusters:
  "Recursion's not as efficient as iteration"??
  - Recursion and the Call Stack
  - Iteration and Explicit Stacks
  - Tail Recursion (but our KW text is wrong about this!)

Random String Permutations (reigniouS mPRrtmnsdtan aot)

Problem: Permute a string so that every reordering of the string is equally likely. You may use a function `randrange(n)`, which selects a number `[0,n)` uniformly at random.

Random String Permutations Understanding the Problem

A string is:
an empty string **or** a letter plus the rest of the string.

We want every letter to have an equal chance to end up first. We want all permutations of the rest of the string to be equally likely to go after.

And.. there's only one empty string.

(Tests: tricky, but result should always have same letters as orginal.)

Random String Permutations Algorithm

PERMUTE(s):
if s is empty, just return s
else:
use randRange to choose a random first letter
permute the rest of the string
(minus that random letter)
return a string that starts with the random letter
and continues with the permuted rest of the string

Random String Permutations Converting Algorithm to Code

```
PERMUTE(s):
  if s is empty, just return s
  else:
    choose random letter
    permute the rest
    return random letter + rest
```

Thinking Recursively

DO NOT START WITH CODE. Write the *story* of the problem, including the data definition!

Define the problem: What should be done given a particular input?

Solve some example cases by hand.

Identify and solve the (usually simple) base case(s).

Figure out how to break the complex cases down in terms of **any smaller case(s)**. For the smaller cases, call the function recursively and **assume it works**. Do **not** think about how!

Implementing Recursion (REMINDER!)

Once you have all that, write out your solution in comments (a "template"). Then fill out the code and test.

(Should be easy… if it's hard, maybe you're not assuming your recursive call works!)

Recursion Example: Fibs (SKIPPING in class)

Problem: Calculate the n[th] Fibonacci number, from the sequence 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

First two numbers are 1; each succeeding number is the sum of the previous two numbers:

$$fib_n = \begin{cases} 1 & ifn = 1 \\ 1 & ifn = 2 \\ fib_{n-1} + fib_{n-2} & ifn \geq 3 \end{cases}$$

Fibs, Worked, First Pass (SKIPPING in class)

Problem: Calculate the n[th] Fibonacci number, from the sequence 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

$$fib_n = \begin{cases} 1 & ifn = 1 \\ 1 & ifn = 2 \\ fib_{n-1} + fib_{n-2} & ifn \geq 3 \end{cases}$$

```
int fib(int n) {
  if (n <= 2) return 1;
  else        return fib(n-1) + fib(n-2);
}
```


Today's Outline

- Thinking Recursively
- Recursion Examples
- Analyzing Recursion: Induction and Recurrences
- Analyzing Iteration: Loop Invariants
- Mythbusters: "Recursion's not as efficient as iteration"??
  - Recursion and the Call Stack
  - Iteration and Explicit Stacks
  - Tail Recursion (but our KW text is wrong about this!)

Induction and Recursion, Twins Separated at Birth?

Base case

Prove for some small value(s).

Inductive Step

Break a larger case down into smaller ones that we assume work (the Induction Hypothesis).

Base case

Calculate for some small value(s).

Otherwise, break the problem down in terms of itself (smaller versions) and then call this function to solve the smaller versions, assuming it will work.

Proving a Recursive Function Correct with Induction is **EASY**

Just follow your code's lead and use induction.

Your base case(s)? Your code's base case(s).

How do you break down the inductive step? However your code breaks the problem down into smaller cases.

What do you assume? That the recursive calls just work (for smaller input sizes as parameters, which better be how your recursive code works!).

Reminder: Factorial

One definition…

$$n! = 1 * 2 * \cdots * n$$

which is not very useful for recursion.

This one *is* useful. It gives us the "insight into how to break the problem down":

$$n! = \begin{cases} 1, & n = 0 \\ n * (n-1)!, & otherwise \end{cases}$$

Proving a Recursive Function Correct with Induction is **EASY**

```
// Precondition: n >= 0.
// Postcondition: returns n!
int factorial(int n)
{
  if (n == 0)
    return 1;

  else
    return n*factorial(n-1);
}
```

ALWAYS connect **what the code does** with what you want to prove.

Prove: `factorial(n)` = n!

Base case: n = 0.

Our code returns 1 when n = 0, and 0! = 1 by definition. ✓

Inductive step: For some k > 0, our code returns `k*factorial(k-1)`. By IH, `factorial(k-1)` = (k-1)! and k! = k*(k-1)! by definition. **QED**

Proving A Recursive Algorithm Works

Problem: Prove that our algorithm for randomly permuting a string gives an equal chance of returning every permutation (assuming **`randrange(n)`** works as advertised).

Recurrence Relations… Already Covered

See METYCSSA #5-7.

Additional Problem: Prove binary search takes O(lg n) time.

```
// Search array[left..right] for target.
// Return its index or the index where it should go.
int bSearch(int array[], int target, int left, int right)
{
  if (right < left) return left;
  int mid = (left + right) / 2;
  if (target <= array[mid])
    return bSearch(array, target, left, mid-1);
  else
    return bSearch(array, target, mid+1, right);
}
```


Binary Search Problem (Worked)

Note: Let n be # of elements considered in the array (right – left + 1).

```
int bSearch(int array[], int target, int left, int right)
{
  if (right < left) return left;   O(1), base case
  int mid = (left + right) / 2;    O(1)
  if (target <= array[mid])        O(1)
    return bSearch(array, target, left, mid-1);   ~T(n/2)
  else
    return bSearch(array, target, mid+1, right);  ~T(n/2)
}
```

Binary Search Problem (Worked)

For n=0: T(0) = 1
For n>0: T(n) = T(⌊n/2⌋) + 1

To guess at the answer, we simplify:

Change ⌊n/2⌋ to n/2.
Change base case to T(1)
(We'll never reach 0 by dividing by 2!)

For n=1: T(1) = 1
For n>1: T(n) = T(n/2) + 1
T(n) = (T(n/4) + 1) + 1
T(n) = T(n/4) + 2
T(n) = T(n/8) + 3
T(n) = T(n/16) + 4
T(n) = T(n/(2^i)) + i

Sub in T(n/2) = T(n/4)+1

Sub in T(n/4) = T(n/8)+1

Sub in T(n/8) = T(n/16)+1

Binary Search Problem (Worked)

To guess at the answer, we simplify:

For n=1: T(1) = 1
For n>1: T(n) = T(n/2) + 1
For n>1: T(n) = T(n/(2^i)) + i

To reach the base case, let n/2^i = 1
n = 2^i means i = lg n

Why did that work out so well?

T(n) = T(n/2^lg n) + lg n = T(1) + lg n = lg n + 1
T(n) ∈ O(lg n)

Binary Search Asymptotic Performance, Proof by Induction

For n=0: T(0) = 1
For n>0: T(n) = T(⌊n/2⌋) + 1

T(1) = T(0) + 1 = 2
T(2) = T(3) = T(1) + 1 = 3.

Set $n_0 = 2$ because lg 0 is undefined and lg 1 = 0.

Prove T(n) ∈ O(lg n)

We want c lg 2 ≥ T(2) and we know T(2) = 3.

Let c = 3, $n_0 = 2$.
Show for all n ≥ n_0, T(n) ≤ c lg n.

Solve for c: c ≥ 3/lg 2 = 3.

Base cases: T(2) = 3 = 3 lg 2 ✓
Base cases: T(3) = 3 ≤ 3 lg 3 ✓

Binary Search Problem (Worked)

Note: T(0) = 1, T(1) = 2, T(2) = 3, T(3) = 3
For n>3: T(n) = T(⌊n/2⌋) + 1 c = 3, $n_0 = 2$

Base cases: prev slide ✓
WLOG, let $n \in \mathbb{Z}$, assume $n > 3$ (skipping the base cases)
Induction Hypothesis: assume T(⌊n/2⌋) ≤ 3 lg ⌊n/2⌋;
since n ≥ 4, then ⌊n/2⌋ ≥ 2. (assuming what we need!)
Inductive step: T(n) = T(⌊n/2⌋) + 1
≤ 3 lg(⌊n/2⌋) + 1
≤ 3 lg(n/2) + 1
= 3 lg n − 3 lg 2 + 1

Today's Outline

- Thinking Recursively
- Recursion Examples
- Analyzing Recursion: Induction and Recurrences
- Analyzing Iteration: Loop Invariants
- Mythbusters:
  "Recursion's not as efficient as iteration"??
  - Recursion and the Call Stack
  - Iteration and Explicit Stacks
  - Tail Recursion (but our KW text is wrong about this!)

(Tail) Recursive → Iterative

It's often simple to convert a recursive function to an iterative one (and vice versa).

```
int bSearch(int array[], int target, int left, int right)
{
  while (!(right < left))
    int mid = (left + right) / 2;
    if (target <= array[mid])
      right = mid - 1;
    else
      left = mid + 1;
  }
  return left;
}
```

```
if (right < left) return left;
int mid = (left + right) / 2;
if (target <= array[mid])
  return bSearch(array,target,left, mid-1);
else
  return bSearch(array, target, mid+1, right);
```

Analyzing Loops

Maybe we can use the same techniques we use for proving correctness of recursion to prove correctness of loops...

We do this by stating and proving "invariants", properties that are always true (don't vary) at particular points in the program.

One way of thinking of a loop is that we spend each loop iteration *fixing* the invariant for the next iteration.

Insertion Sort

```
int insertionSort(int array[], int length)
{
  // Invariant: before each test i < length (including the last
  // one), the elements in array[0..i-1] are in sorted order.
  for (int i = 1; i < length; i++)
  {
    // i is about to go up by 1 but array[i] may be out of order!
    // gotta fix it gotta fix it gotta fix it!!!
    int val = array[i];
    int newIndex = bSearch(array, val, 0, i);
    for (int j = i; j > newIndex; j--)
      array[j] = array[j-1];
    array[newIndex] = val;
  }
}
```

(invariant)

(invariant anxiety)

Proving a Loop Invariant

Induction variable: number of times through the loop.

Base case: Prove the invariant true before the first loop guard test.

Induction hypothesis: Assume the invariant holds just before some (unspecified) iteration's loop guard test.

Inductive step: Prove the invariant holds at the end of that iteration (just before the next loop guard test).

Extra bit: Make sure the loop will eventually end!

We'll prove insertion sort works, but the cool part is not proving it works (duh). The cool part is that the proof is a natural way to think about it working!

Proving Insertion Sort Works

```
// Invariant: before each test i < length (including the last
// one), the elements in array[0..i-1] are in sorted order.
for (int i = 1; i < length; i++)
{
  // i is about to go up by 1 but array[i] may be out of order!
  int val = array[i];
  int newIndex = bSearch(array, val, 0, i);
  for (int j = i; j > newIndex; j--)
    array[j] = array[j-1];
  array[newIndex] = val;
}
```

Base case (just before "`i < length`"): array[0..0] has one element; so, it's always in sorted order.

What's the niggly detail we skipped here?

Proving Insertion Sort Works

```
// Invariant: before each test i < length (including the last
// one), the elements in array[0..i-1] are in sorted order.
for (int i = 1; i < length; i++)
{
  // i is about to go up by 1 but array[i] may be out of order!
  int val = array[i];
  int newIndex = bSearch(array, val, 0, i);
  for (int j = i; j > newIndex; j--)
    array[j] = array[j-1];
  array[newIndex] = val;
}
```

Induction hypothesis: just before we test `k < length`, array[0..k-1] are in sorted order. (When the loop starts, i = k.)

Proving Insertion Sort Works

```
// Invariant: before each test i < length (including the last
// one), the elements in array[0..i-1] are in sorted order.
for (int i = 1; i < length; i++)
{
  // i is about to go up by 1 but array[i] may be out of order!
  int val = array[i];
  int newIndex = bSearch(array, val, 0, i);
  for (int j = i; j > newIndex; j--)
    array[j] = array[j-1];
  array[newIndex] = val;
}
```

(surprisingly: linear search may be a better choice here; ask after class!)

Inductive Step: bSearch gives the appropriate index at which to put array[i]. So, the new element ends up in sorted order, and the rest of array[0..i] stays in sorted order.

(A bit hand-wavy… what *should* we have done?)

Proving Insertion Sort Works

```
// Invariant: before each test i < length (including the last
// one), the elements in array[0..i-1] are in sorted order.
for (int i = 1; i < length; i++)
{
  // i is about to go up by 1 but array[i] may be out of order!
  int val = array[i];
  int newIndex = bSearch(array, val, 0, i);
  for (int j = i; j > newIndex; j--)
    array[j] = array[j-1];
  array[newIndex] = val;
}
```

Loop termination: The loop ends when i == length (which it must be eventually since length is non-negative and i increases). At which point, array[0..i-1] is sorted… which is array[0..length-1] or the whole array

Practice: Prove the Inner Loop Correct

```
for (int i = 1; i < length; i++)
{
  // i is about to go up by 1 but array[i] may be out of order!
  int val = array[i];
  int newIndex = bSearch(array, val, 0, i);
  // What's the invariant?  Maybe: just before j > newIndex,
  // "array[0..j-1] + array[j+1..i] = the old array[0..i-1]"
  for (int j = i; j > newIndex; j--)
    array[j] = array[j-1];
  array[newIndex] = val;
}
```

We just waved our hands at the inner loop. Prove it's correct! (This may feel unrealistically easy!)

Do note that j is going *down*, not up.

Today's Outline

- Thinking Recursively
- Recursion Examples
- Analyzing Recursion: Induction and Recurrences
- Analyzing Iteration: Loop Invariants
- Mythbusters: "Recursion's not as efficient as iteration"??
  - Recursion and the Call Stack
  - Iteration and Explicit Stacks
  - Tail Recursion (but our KW text is wrong about this!)

Mythbusters: Recursion vs. Iteration

Which one can ***do*** more? Recursion or iteration?

Mythbusters: Simulating a Loop with Recursion

```
int i = 0
while (i < n)
  doFoo(i)
  i++
```

```
recDoFoo(0, n)
```

Where `recDoFoo` is:

```
void recDoFoo(int i, int n)
{
  if (i < n) {
    doFoo(i)
    recDoFoo(i + 1, n)
  }
}
```

Anything we can do with iteration, we can do with recursion.

Mythbusters: Simulating Recursion with a Stack (Going Quick.. Already Discussed)

How does fib actually work?

Each function call generates a *stack frame* (also known as *activation record* or, just between us, *function pancake*) holding local variables and the program point to return to, which is pushed on a stack (the call stack) that tracks the current chain of function calls.

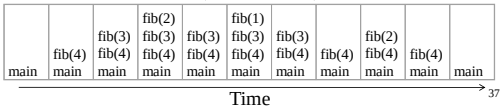
```
int fib(int n) {
  if (n <= 2) return 1;
  else        return fib(n-1) + fib(n-2);
}
cout << fib(4) << endl;
```

Mythbusters: Simulating Recursion with a Stack (Going Quick.. Already Discussed)

How does fib actually work?

```
int fib(int n) {
  if (n <= 2) return 1;
  else        return fib(n-1) + fib(n-2);
}
cout << fib(4) << endl;
```

The call (or "run-time") stack

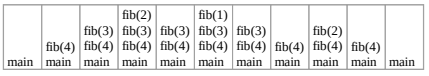


Time

Aside: Efficiency and the Call Stack

The *height* of the call stack tells us the maximum memory we use storing the stack.

height = 4 frames



The number of calls that go through the call stack tells us something about time usage. (The # of calls multiplied by worst-case time per call bounds the asymptotic complexity.)

So, when calculating memory usage, we must consider stack space!
But only the non-tail-calls count… see later slides on tail recursion and tail calls.

Aside: Limits of the Call Stack

```
int fib(int n) {
  if (n == 1)       return 1;
  else if (n == 2)  return 1;
  else              return fib(n-1) + fib(n-2);
}
cout << fib(0) << endl;
```

What will happen?

a. Returns 1 immediately.
b. Runs forever (infinite recursion)
c. Stops running when n "wraps around" to positive values.
d. Bombs when the computer runs out of stack space.
e. None of these.

Mythbusters: Simulating Recursion with a Stack

How do we simulate fib with a stack?

That's what our computer *already* does. We can sometimes do it a bit more efficiently by only storing what's really needed on the stack:

```
int fib(int n)
  result = 0
  push(n)
  while not isEmpty
    n = pop
    if (n <= 2) result++;
    else push(n - 1); push(n - 2)
  return result
```

OK, this is cheating a bit (in a good way).
To get down and dirty, see CPSC 313 + 311.

Mythbusters: Recursion vs. Iteration

Which one is more elegant? Recursion or iteration?

Mythbusters: Recursion vs. Iteration

Which one is more efficient? Recursion or iteration?

Accidentally Making **Lots** of Recursive Calls; Recall...

- Recursive Fibonacci:

```
int Fib(n)
  if (n == 0 or n == 1) return 1
  else return Fib(n - 1) + Fib(n - 2)
```

- *Lower* bound analysis
- `T(0), T(1) >= b`

  `T(n) >= T(n - 1) + T(n - 2) + c` *if* `n > 1`
- Analysis

  let ϕ be `(1 + √5)/2` which satisfies $\phi^2 = \phi + 1$

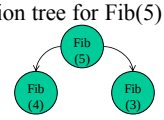
  show by induction on `n` that `T(n) >= bϕ^(n - 1)`

Already discussed Day 1.. Skipping.

Accidentally Making **Lots** of Recursive Calls; Recall...

```
int Fib(n)
  if (n == 1 or n == 2) return 1
  else return Fib(n - 1) + Fib(n - 2)
```

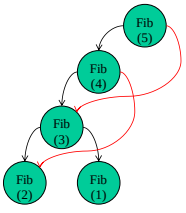
Finish the recursion tree for Fib(5)…



Already discussed Day 1.. Skipping.

Fixing Fib: Requires Iteration?

What we really want is to "share" nodes in the recursion tree:

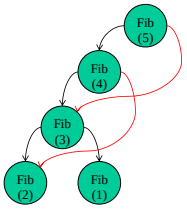


Already discussed Day 1.. Skipping.

Fixing Fib with Iteration and "Dynamic Programming"

Here's one fix that "walks up" the left of the tree:

```
int fib_dp(int n)
{
  if (n == 1) return 1;
  int fib = 1, fib_old = 1;
  int i = 2;
  while (i < n) {
    int fib_new = fib + fib_old;
    fib_old = fib;
    fib = fib_new;
    i++;
  }
  return fib;
}
```

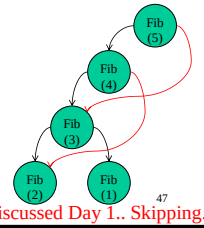


Already discussed Day 1.. Skipping.

Fixing Fib with Recursion and "Memoizing"

Here's another fix that just takes note of problems it's solved before:

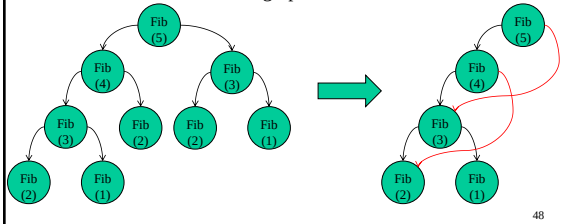
```
int[] fib_solns = new int[large_enough]; // init to 0
fib_solns[1] = 1;
fib_solns[2] = 1;
int fib_memo(int n)
{
  // If we don't know the answer…
  if (fib_solns[n] == 0)
    fib_solns[n] = fib_memo(n-1) +
                   fib_memo(n-2);
  return fib_solns[n];
}
```



Already discussed Day 1.. Skipping.

Fixing Fib with Recursion and Pure Functional Programming

In a "pure functional" programming language (like Haskell and a subset of Racket), the interpreter *can* (but may not) notice that nodes in the graph are the same and share them.



Why? Because Fib(n) can *never* return two different values in a pure functional language.

Mythbusters: Recursion vs. Iteration

Which one is more ***efficient***? Recursion or iteration?

It's probably easier to shoot yourself in the foot without noticing when you use recursion, and the call stack may carry around a bit more (a constant factor more) memory than you really need to store, but otherwise…

Neither is more efficient.

Before we move, on, let's lather, rinse, and repeat.

Managing the Call Stack: Tail Recursion

```
void shampoo()
{
  cout << "Lather, Rinse" << endl;
  shampoo();
}
```

This is clearly infinite recursion. The call stack will get as deep as it can get and then bomb, right?

But... why? What *work* is the call stack doing?

There's *nothing* to remember on the stack!

Try compiling it with at least –O2 optimization and running. It won't give a stack overflow!

Tail **Call** (should be CPSC 110 review!)

A function call is a "tail call" if that call is the absolute last thing the function needs to do before returning.

In that case, why bother pushing a new stack frame? There's nothing to remember. Just re-use the old frame.

That's what most compilers will do (although Java has had some issues with this!).

Tail **Recursion** (should be CPSC 110 review!)

A function is "tail recursive" if all of its recursive calls are tail calls (i.e., each recursive call in the function is the absolute last thing the function needs to do before returning).

Can two calls **both** be the last thing that needs to be done before the function returns?

Sure: consider binary search: we recurse to the left or to the right, but not **both**.

WARNING: Koffman and Wolfgang text is wrong about this! Having a (or even "the") recursive call on the last line is ***not*** enough.

Tail Recursive?

```
int fib(int n) {
  if (n <= 2) return 1;
  else        return fib(n-1) + fib(n-2);
}
```

Tail recursive?

a. Yes.
b. No.
c. Not enough information.

Tail Recursive?

```
int factorial(int n) {
  if (n == 0) return 1;
  else        return n * factorial(n – 1);
}
```

Tail recursive?

a. Yes.
b. No.
c. Not enough information.

Tail Recursive?

```
int factorial(int n) { return fact_acc(n, 1); }

int fact_acc (int n, int acc) {
  if (n == 0) return acc;
  else        return fact_acc(n - 1, acc * n);
}
```

Tail recursive?

a. Yes.
b. No.
c. Not enough information.

Tail Calls

```
int fact(int n) { return fact_acc(n, 1); }

int fact_acc (int n, int acc) {
  if (n == 0) return acc;
  else        return fact_acc(n - 1, acc * n);
}
```

The call to `fact_acc` in `factorial` is a tail call and therefore also need not consume extra stack space.

To Do

- CPSC 121 Review: Epp 4.2-4.4, 5.1-5.2, 7.1-7.2
- Read: Epp 4.5, Koffman/Wolfgang Chapter 7

Coming Up

- One of (TBD!):
  - Priority Queues
  - Trees