

# Unit #4: Recursion, Induction, and Loop Invariants

CPSC 221: Algorithms and Data Structures

Lars Kotthoff<sup>1</sup>

`larsko@cs.ubc.ca`

---

<sup>1</sup>With material from Will Evans, Steve Wolfman, Alan Hu, Ed Knorr, and Kim Voll.

# Unit Outline

- ▷ Thinking Recursively
- ▷ Recursion Examples
- ▷ Analyzing Recursion: Induction and Recurrences
- ▷ Analyzing Iteration: Loop Invariants
- ▷ How Computers Handle Recursion
  - ▷ Recursion and the Call Stack
  - ▷ Iteration and Explicit Stacks
  - ▷ Tail Recursion (KW text is wrong about this!)

# Learning Goals

- ▷ Describe the relation between recursion and induction.
- ▷ Prove a program is correct using loop invariants and induction.
- ▷ Become more comfortable writing recursive algorithms.
- ▷ Convert between iterative and recursive algorithms.
- ▷ Describe how a computer implements recursion.
- ▷ Draw a recursion tree for a recursive algorithm.



recursion



All

Images

Videos

Apps

Maps

More ▾

Search tools

About 8,940,000 results (0.55 seconds)

Did you mean: ***recursion***

## Random Permutations (randomPermutation)

**Problem:** Permute a string so that every reordering of the string is equally likely.

# Thinking Recursively

1. DO NOT START WITH CODE. Instead, write the story of the problem, in natural language.
2. Define the problem: What should be done given a particular input?
3. Identify and solve the (usually simple) base case(s).
4. Determine how to break the problem down into smaller problems of the same kind.
5. Call the function recursively to solve the smaller problems. Assume it works. Do not think about how!
6. Use the solutions to the smaller problems to solve the original problem.

Once you have all that, write out your solution in comments and then translate it into code.

## Random Permutations (rPnastma detinoRmuo)

**Problem:** Permute a string so that every reordering of the string is equally likely.

**Idea:**

1. Pick a letter to be the first letter of the string. (Every letter should be equally likely.)
2. Pick the rest of the string to be a random permutation of the remaining string (without that letter).

It's slightly simpler in C++ if we pick a letter to be the **last** letter.

## Random Permutations (rPnastma detinoRmuo)

**Problem:** Permute a string so that every reordering of the string is equally likely.

```
// randomly permutes the first n chars of S
void permute(string &S, int n) {
    if(n > 1) {
        int i = rand() % n; // random char of S
        char tmp = S[i];    // move to end of S
        S[i] = S[n-1];
        S[n-1] = tmp;
        // randomly permute S[0..n-2]
        permute(S, n-1);
    }
}
```

`rand() % n` returns an integer from  $\{0, 1, \dots, n-1\}$  uniformly at random.



# Induction and Recursion, Twins Separated at Birth?

	Induction	Recursion
Base case	Prove for some small value(s).	Calculate for some small value(s).
Inductive step	Break a larger case down into smaller ones that we assume work (the Induction Hypothesis).	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 Algorithm is Correct

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.**

Inductive hypothesis? **The recursive calls work for smaller-sized inputs.**

# Proving a Recursive Algorithm is Correct

```
// Pre:  $n \geq 0$ .  
// Post: returns  $n!$   
int fact(int n) {  
    if (n == 0)  
        return 1;  
  
    else  
        return  
            n * fact(n-1);  
}
```

Prove:  $\text{fact}(n) = n!$

Base case:  $n = 0$

$\text{fact}(0)$  returns 1 and  
 $0! = 1$  by definition.

Inductive hyp:  $\text{fact}(n)$   
returns  $n!$  for all  $n \leq k$ .

Inductive step: For  $n = k + 1$ ,  
code returns  $n * \text{fact}(n-1)$ .  
By IH,  $\text{fact}(n-1)$  is  
 $(n-1)!$  and  $n! = n * (n-1)!$   
by definition.

# Proving a Recursive Algorithm is Correct

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

Base case: strings of length 1 have only one permutation.

Induction hypothesis: Assume that our call to `permute(S, n-1)` works (randomly permutes the first  $n-1$  characters of  $S$ ).

We choose the last letter uniformly at random from the string. To get a random permutation, we need only randomly permute the remaining letters. `permute(S, n-1)` does exactly that.

## Recurrence Relations... Already Covered

See Runtime Examples #5-7.

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

```
// Search A[i..j] for key.  
// Return index of key or -1 if key not found.  
int bSearch(int A[], int key, int i, int j) {  
    if(j < i) return -1;  
    int mid = (i + j) / 2;  
    if(key < A[mid])  
        return bSearch(A, key, i, mid-1);  
    else if(key > A[mid])  
        return bSearch(A, key, mid+1, j);  
    else return mid;  
}
```

## Binary Search Problem (Worked)

Note: Let  $n$  be # of elements considered in the array,  
 $n = j - i + 1$ .

```
int bSearch(int A[], int key, int i, int j) {  
    if(j < i) return -1; // constant (base case)  
    int mid = (i + j) / 2; // constant  
    if(key < A[mid]) // constant  
        //  $T(\text{floor}(n/2))$   
        return bSearch(A, key, i, mid-1);  
    else if(key > A[mid]) // constant  
        //  $T(\text{floor}(n/2))$   
        return bSearch(A, key, mid+1, j);  
    else return mid; // constant  
}
```

## Binary Search Problem (Worked)

$$T(n) = \begin{cases} 1 & \text{if } n = 0 \\ T(\lfloor n/2 \rfloor) + 1 & \text{if } n > 0 \end{cases}$$

$$T(0) = 1$$

$$T(1) = T(0) + 1 = 2$$

$$T(2) = T(1) + 1 = 3$$

$$T(3) = T(1) + 1 = 3$$

$$T(4) = T(2) + 1 = 4$$

$$T(5) = T(2) + 1 = 4$$

$$T(6) = T(3) + 1 = 4$$

$$T(7) = T(3) + 1 = 4$$

...

## Binary Search Problem (Worked)

To guess the complexity: simplify!

Change  $\lfloor n/2 \rfloor$  to  $\frac{n}{2}$ .

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n/2) + 1 & \text{if } n > 1 \end{cases}$$

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + 1 \\ &= \left(T\left(\frac{n}{4}\right) + 1\right) + 1 \\ &= T\left(\frac{n}{4}\right) + 2 \\ &= T\left(\frac{n}{8}\right) + 3 \\ &= T\left(\frac{n}{16}\right) + 4 \\ &= T\left(\frac{n}{2^k}\right) + k \end{aligned}$$



## Binary Search Problem (Worked)

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T\left(\frac{n}{2^k}\right) + k & \text{if } n > 1 \end{cases}$$

Reach base case when  $\frac{n}{2^k} = 1 \rightarrow k = \lg n$ .

$$\begin{aligned} T(n) &= T\left(\frac{n}{2^{\lg n}}\right) + \lg n \\ &= T(1) + \lg n \\ &= \lg n + 1 \\ &\in \Theta(\lg n) \end{aligned}$$

# Binary Search Problem (Worked)

**Claim**  $T(n) = \lceil \lg(n+1) \rceil + 1$

**Proof** (by induction on  $n$ )

Base:  $T(0) = 1 = \lceil \lg(0+1) \rceil + 1$

Ind. Hyp:  $T(n) = \lceil \lg(n+1) \rceil + 1$  for  $n \leq k$  (for  $k > 0$ ).

Ind. Step: For  $n = k + 1$ ,  $T(k+1) = T(\lfloor \frac{k+1}{2} \rfloor) + 1 =$

If  $k$  is even

$$\begin{aligned} &= T\left(\frac{k}{2}\right) + 1 \text{ (} k \text{ is even)} \\ &= (\lceil \lg(\frac{k}{2} + 1) \rceil + 1) + 1 \text{ (IH)} \\ &= \lceil \lg(2(\frac{k}{2} + 1)) \rceil + 1 \\ &= \lceil \lg(k + 2) \rceil + 1 \\ &= \lceil \lg(n + 1) \rceil + 1 \text{ (} n = k + 1 \text{)} \end{aligned}$$

If  $k$  is odd

$$\begin{aligned} &= T\left(\frac{k+1}{2}\right) + 1 \text{ (} k \text{ is odd)} \\ &= (\lceil \lg(\frac{k+1}{2} + 1) \rceil + 1) + 1 \text{ (IH)} \\ &= \lceil \lg(2(\frac{k+1}{2} + 1)) \rceil + 1 \\ &= \lceil \lg(k + 3) \rceil + 1 \\ &= \lceil \lg(k + 2) \rceil + 1 \text{ (} k \text{ is odd)} \\ &= \lceil \lg(n + 1) \rceil + 1 \text{ (} n = k + 1 \text{)} \end{aligned}$$

## Binary Search Problem (Worked)

$$T(n) = \lceil \lg(n+1) \rceil + 1 \in \Theta(\lg n)$$

# Proving an Algorithm with Loops is Correct

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 it starts with a true invariant and does work to keep the invariant true for the next iteration of the loop.

# Insertion Sort

```
void insertionSort(int A[], int length) {  
    for(int i = 1; i < length; i++) {  
        // Invariant: the elements in A[0..i-1] are in  
        // sorted order.  
        int val = A[i];  
        int j;  
        for(j = i; j > 0 && A[j-1] > val; j--)  
            A[j] = A[j-1];  
        A[j] = val;  
    }  
}
```

# Proving a Loop Invariant

**Induction variable:** number of times through the loop.

**Base case:** Prove the invariant true before the loop starts.

**Induction hypothesis:** Assume the invariant holds just before beginning some (unspecified) iteration.

**Inductive step:** Prove the invariant holds at the end of that iteration for the next iteration.

**Extra bit:** Make sure the loop will eventually end!

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

# Insertion Sort

```
for(int i = 1; i < length; i++) {  
    // Invariant: the elements in A[0..i-1] are in  
        sorted order.  
    int val = A[i];  
    int j;  
    for(j = i; j > 0 && A[j-1] > val; j--)  
        A[j] = A[j-1];  
    A[j] = val;  
}
```

Base case (at the start of the ( $i = 1$ ) iteration):  $A[0..0]$  only has one element; so, it's always in sorted order.

# Insertion Sort

```
for(int i = 1; i < length; i++) {  
    // Invariant: the elements in A[0..i-1] are in  
        sorted order.  
    int val = A[i];  
    int j;  
    for(j = i; j > 0 && A[j-1] > val; j--)  
        A[j] = A[j-1];  
    A[j] = val;  
}
```

Induction Hypothesis: At the start of iteration  $i$  of the loop,  $A[0..i-1]$  are in sorted order.



# Insertion Sort

```
for(int i = 1; i < length; i++) {  
    // Invariant: the elements in A[0..i-1] are in  
        sorted order.  
    int val = A[i];  
    int j;  
    for(j = i; j > 0 && A[j-1] > val; j--)  
        A[j] = A[j-1];  
    A[j] = val;  
}
```

Inductive Step: The inner loop places  $\text{val} = A[i]$  at the appropriate index  $j < i$  by shifting elements of  $A[0..i-1]$  that are larger than  $\text{val}$  one position to the right. Since  $A[0..i-1]$  is sorted (by IH),  $A[0..i]$  ends up in sorted order and the invariant holds at the start of the next iteration ( $i = i + 1$ ).

# Insertion Sort

```
for(int i = 1; i < length; i++) {  
    // Invariant: the elements in A[0..i-1] are in  
        sorted order.  
    int val = A[i];  
    int j;  
    for(j = i; j > 0 && A[j-1] > val; j--)  
        A[j] = A[j-1];  
    A[j] = val;  
}
```

Loop termination: The loop ends after  $\text{length} - 1$  iterations. When it ends, we were about to enter the  $(i = \text{length})$  iteration. Therefore, by the newly proven invariant, when the loop ends,  $A[0..\text{length}-1]$  is in sorted order...which means A is sorted!

## Practice: Prove the Inner Loop Correct

```
for(int i = 1; i < length; i++) {  
    // Invariant: the elements in A[0..i-1] are in  
    // sorted order.  
    int val = A[i];  
    int j;  
    for(j = i; j > 0 && A[j-1] > val; j--)  
        // What's the invariant? Something like  
        // "A[0..j-1] + A[j+1..i] = the old A[0..i-1]  
        // and val <= A[j+1..i]"  
        A[j] = A[j-1];  
    A[j] = val;  
}
```

Prove by induction that the inner loop operates correctly. (This may feel unrealistically easy!)

Finish the proof! (As we did for the outer loop, talk about what the invariant means when the loop ends.)

# Recursion vs. Iteration

Which one is better? Recursion or iteration?

# Simulating a Loop with Recursion

```
int i = 0
while (i < n) {
    foo(i)
    i++
}
```

recFoo(0, n)

where recFoo is:

```
void recFoo(int i, int n) {
    if (i < n) {
        foo(i)
        recFoo(i + 1, n)
    }
}
```

# Simulating a Loop with Recursion

```
recFoo(0, n)
```

where recFoo is:

```
int i = 0
while(i < n) {
    foo(i)
    i++
}
```

```
void recFoo(int i, int n) {
    if(i < n) {
        foo(i)
        recFoo(i + 1, n)
    }
}
```

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

# Simulating Recursion with a Stack

How does recursion work in a computer?

Each function call generates an *activation record* – 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;  
    int a = fib(n-1);  
    int b = fib(n-2);  
    return a+b;  
}  
  
int main() { cout << fib(4) << endl; }
```

# Function/Method calls

A function or method call is an interruption or aside in the execution flow of a program:

```
int a, b, c, d;  
a = 3;  
b = 6;  
c = foo(a, b);  
d = 9;
```

```
int foo(int x, int y) {  
    while(x > 0) {  
        y++;  
        x--;  
    }  
    return y;  
}
```



# Function Calls in Daily Life

How do you handle interruptions in daily life?

- ▷ You're at home, working on CPSC221 project.
- ▷ You stop to look up something in the book.
- ▷ Your roommate/spouse/partner/parent/etc. asks for your help moving some stuff.
- ▷ Your buddy calls.
- ▷ The doorbell rings.

# Function Calls in Daily Life

How do you handle interruptions in daily life?

- ▷ You're at home, working on CPSC221 project.
- ▷ You stop to look up something in the book.
- ▷ Your roommate/spouse/partner/parent/etc. asks for your help moving some stuff.
- ▷ Your buddy calls.
- ▷ The doorbell rings.

You stop what you're doing, you make a note of where you were in your task, you handle the interruption, and then you go back to what you were doing.

# Activation Records in Daily Life

I am working on line 11 of my `stack.cpp` file...

# Activation Records in Daily Life

I am working on line 12 of my `stack.cpp` file...

# Activation Records in Daily Life

I am reading about the delete function in Koffman p. 26.

I am working on line 12 of my `stack.cpp` file...

# Activation Records in Daily Life

I am reading about the delete function in Koffman p. 27.

I am working on line 12 of my `stack.cpp` file...

## Activation Records in Daily Life

I have moved 20lbs of steer manure to the garden.

I am reading about the delete function in Koffman p. 27.

I am working on line 12 of my `stack.cpp` file...

## Activation Records in Daily Life

I have moved 40lbs of steer manure to the garden.

I am reading about the delete function in Koffman p. 27.

I am working on line 12 of my `stack.cpp` file...



# Activation Records in Daily Life

My buddy is telling me some insane story about last night.

I have moved 40lbs of steer manure to the garden.

I am reading about the delete function in Koffman p. 27.

I am working on line 12 of my `stack.cpp` file...

## Activation Records in Daily Life

My buddy is just about to get to the point where he pukes. . .

I have moved 40lbs of steer manure to the garden.

I am reading about the delete function in Koffman p. 27.

I am working on line 12 of my `stack.cpp` file. . .

# Activation Records in Daily Life

I am signing for a FedEx package.

My buddy is just about to get to the point where he pukes. . .

I have moved 40lbs of steer manure to the garden.

I am reading about the delete function in Koffman p. 27.

I am working on line 12 of my `stack.cpp` file. . .

## Activation Records in Daily Life

My buddy is just about to get to the point where he pukes. . .

I have moved 40lbs of steer manure to the garden.

I am reading about the delete function in Koffman p. 27.

I am working on line 12 of my `stack.cpp` file. . .

# Activation Records in Daily Life

My buddy has finally finished his story.

I have moved 40lbs of steer manure to the garden.

I am reading about the delete function in Koffman p. 27.

I am working on line 12 of my `stack.cpp` file...

## Activation Records in Daily Life

I have moved 40lbs of steer manure to the garden.

I am reading about the delete function in Koffman p. 27.

I am working on line 12 of my `stack.cpp` file...

## Activation Records in Daily Life

I have moved 60lbs of steer manure to the garden.

I am reading about the delete function in Koffman p. 27.

I am working on line 12 of my `stack.cpp` file...

## Activation Records in Daily Life

I have moved 80lbs of steer manure to the garden.

I am reading about the delete function in Koffman p. 27.

I am working on line 12 of my `stack.cpp` file...



# Activation Records in Daily Life

I am reading about the delete function in Koffman p. 27.

I am working on line 12 of my `stack.cpp` file...

# Activation Records in Daily Life

I am reading about the delete function in Koffman p. 28.

I am working on line 12 of my `stack.cpp` file...

# Activation Records in Daily Life

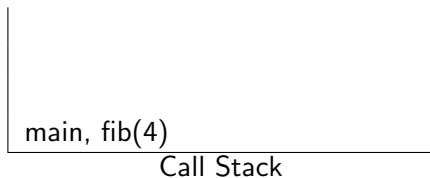
I am working on line 12 of my `stack.cpp` file...

# Activation Records in Daily Life

I have finished `stack.cpp`!

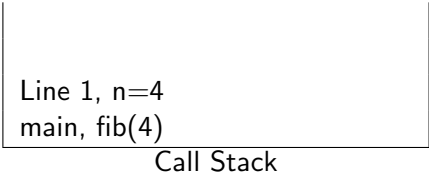
## Simulating Recursion with a Stack

```
int fib(int n) {  
    if (n <= 2) return 1; // 1.  
    int a = fib(n-1);      // 2.  
    int b = fib(n-2);      // 3.  
    return a+b;            // 4.  
}  
  
int main() { cout << fib(4) << endl; }
```



## Simulating Recursion with a Stack

```
int fib(int n) {  
    if (n <= 2) return 1; // 1.  
    int a = fib(n-1);      // 2.  
    int b = fib(n-2);      // 3.  
    return a+b;            // 4.  
}  
  
int main() { cout << fib(4) << endl; }
```



Line 1, n=4  
main, fib(4)

Call Stack

## Simulating Recursion with a Stack

```
int fib(int n) {  
    if (n <= 2) return 1; // 1.  
    int a = fib(n-1);      // 2.  
    int b = fib(n-2);      // 3.  
    return a+b;            // 4.  
}  
  
int main() { cout << fib(4) << endl; }
```

Line 2, n=4, a=fib(3)  
main, fib(4)

Call Stack

## Simulating Recursion with a Stack

```
int fib(int n) {  
    if (n <= 2) return 1; // 1.  
    int a = fib(n-1);      // 2.  
    int b = fib(n-2);      // 3.  
    return a+b;            // 4.  
}  
  
int main() { cout << fib(4) << endl; }
```

Line 1, n=3 Line 2, n=4, a=fib(3) main, fib(4)
--

Call Stack



## Simulating Recursion with a Stack

```
int fib(int n) {  
    if (n <= 2) return 1; // 1.  
    int a = fib(n-1);      // 2.  
    int b = fib(n-2);      // 3.  
    return a+b;            // 4.  
}  
  
int main() { cout << fib(4) << endl; }
```

Line 2, n=3, a=fib(2) Line 2, n=4, a=fib(3) main, fib(4)
--

Call Stack

## Simulating Recursion with a Stack

```
int fib(int n) {  
    if (n <= 2) return 1; // 1.  
    int a = fib(n-1);      // 2.  
    int b = fib(n-2);      // 3.  
    return a+b;            // 4.  
}  
  
int main() { cout << fib(4) << endl; }
```

Line 1, n=2
Line 2, n=3, a=fib(2)
Line 2, n=4, a=fib(3)
main, fib(4)

Call Stack

# Simulating Recursion with a Stack

```
int fib(int n) {  
    if (n <= 2) return 1; // 1.  
    int a = fib(n-1);      // 2.  
    int b = fib(n-2);      // 3.  
    return a+b;            // 4.  
}  
  
int main() { cout << fib(4) << endl; }
```

Line 1, n=2, return 1
Line 2, n=3, a=fib(2)
Line 2, n=4, a=fib(3)
main, fib(4)

Call Stack

# Simulating Recursion with a Stack

```
int fib(int n) {  
    if (n <= 2) return 1; // 1.  
    int a = fib(n-1);      // 2.  
    int b = fib(n-2);      // 3.  
    return a+b;            // 4.  
}  
  
int main() { cout << fib(4) << endl; }
```

Line 2, n=3, a=1 Line 2, n=4, a=fib(3) main, fib(4)
---

Call Stack

## Simulating Recursion with a Stack

```
int fib(int n) {  
    if (n <= 2) return 1; // 1.  
    int a = fib(n-1);      // 2.  
    int b = fib(n-2);      // 3.  
    return a+b;            // 4.  
}  
  
int main() { cout << fib(4) << endl; }
```

Line 3, n=3, a=1, b=fib(1)  
Line 2, n=4, a=fib(3)  
main, fib(4)

Call Stack

## Simulating Recursion with a Stack

```
int fib(int n) {  
    if (n <= 2) return 1; // 1.  
    int a = fib(n-1);      // 2.  
    int b = fib(n-2);      // 3.  
    return a+b;           // 4.  
}  
  
int main() { cout << fib(4) << endl; }
```

Line 1, n=1
Line 3, n=3, a=1, b=fib(1)
Line 2, n=4, a=fib(3)
main, fib(4)

Call Stack

## Simulating Recursion with a Stack

```
int fib(int n) {  
    if (n <= 2) return 1; // 1.  
    int a = fib(n-1);      // 2.  
    int b = fib(n-2);      // 3.  
    return a+b;           // 4.  
}  
  
int main() { cout << fib(4) << endl; }
```

Line 1, n=1, return 1
Line 3, n=3, a=1, b=fib(1)
Line 2, n=4, a=fib(3)
main, fib(4)

Call Stack

## Simulating Recursion with a Stack

```
int fib(int n) {  
    if (n <= 2) return 1; // 1.  
    int a = fib(n-1);      // 2.  
    int b = fib(n-2);      // 3.  
    return a+b;            // 4.  
}  
  
int main() { cout << fib(4) << endl; }
```

Line 3, n=3, a=1, b=1 Line 2, n=4, a=fib(3) main, fib(4)
--

Call Stack



# Simulating Recursion with a Stack

```
int fib(int n) {  
    if (n <= 2) return 1; // 1.  
    int a = fib(n-1);      // 2.  
    int b = fib(n-2);      // 3.  
    return a+b;           // 4.  
}  
  
int main() { cout << fib(4) << endl; }
```

Line 4, n=3, a=1, b=1, return 2
Line 2, n=4, a=fib(3)
main, fib(4)

Call Stack

## Simulating Recursion with a Stack

```
int fib(int n) {  
    if (n <= 2) return 1; // 1.  
    int a = fib(n-1);      // 2.  
    int b = fib(n-2);      // 3.  
    return a+b;            // 4.  
}  
  
int main() { cout << fib(4) << endl; }
```

Line 3, n=4, a=2, b=fib(2)  
main, fib(4)

Call Stack

## Simulating Recursion with a Stack

```
int fib(int n) {  
    if (n <= 2) return 1; // 1.  
    int a = fib(n-1);      // 2.  
    int b = fib(n-2);      // 3.  
    return a+b;           // 4.  
}  
  
int main() { cout << fib(4) << endl; }
```

Line 1, n=2

Line 3, n=4, a=2, b=fib(2)

main, fib(4)

Call Stack

## Simulating Recursion with a Stack

```
int fib(int n) {  
    if (n <= 2) return 1; // 1.  
    int a = fib(n-1);      // 2.  
    int b = fib(n-2);      // 3.  
    return a+b;           // 4.  
}  
  
int main() { cout << fib(4) << endl; }
```

Line 1, n=2, return 1 Line 3, n=4, a=2, b=fib(2) main, fib(4)
---

Call Stack

## Simulating Recursion with a Stack

```
int fib(int n) {  
    if (n <= 2) return 1; // 1.  
    int a = fib(n-1);      // 2.  
    int b = fib(n-2);      // 3.  
    return a+b;            // 4.  
}  
  
int main() { cout << fib(4) << endl; }
```

Line 3, n=4, a=2, b=1  
main, fib(4)

Call Stack

## Simulating Recursion with a Stack

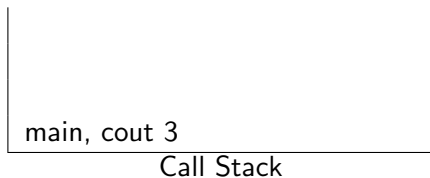
```
int fib(int n) {  
    if (n <= 2) return 1; // 1.  
    int a = fib(n-1);      // 2.  
    int b = fib(n-2);      // 3.  
    return a+b;            // 4.  
}  
  
int main() { cout << fib(4) << endl; }
```

Line 4, n=4, a=2, b=1, return 3  
main, fib(4)

Call Stack

## Simulating Recursion with a Stack

```
int fib(int n) {  
    if (n <= 2) return 1; // 1.  
    int a = fib(n-1);      // 2.  
    int b = fib(n-2);      // 3.  
    return a+b;            // 4.  
}  
  
int main() { cout << fib(4) << endl; }
```



# 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?

1. Returns 1 immediately.
2. Runs forever (infinite recursion).
3. Stops running when n “wraps around” to positive values.
4. Bombs when the computer runs out of stack space.
5. None of these.



## Aside: Efficiency and the Call Stack

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

			fib(2)		fib(1)					
	fib(4)	fib(3)	fib(3)	fib(3)	fib(3)	fib(3)		fib(2)		
	main	main	main	main	main	main	main	main	main	main

The number of calls pushed on the call stack tells us something about running time. (Assuming each call takes constant time, the running time is  $\Theta(\# \text{ of calls})$ .)

# 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) {  
    int result = 0;  
    Stack S;  
    S.push(n);  
    while(!S.is_empty()) {  
        int k = S.pop();  
        if(k <= 2) result++;  
        else {  
            S.push(k - 1);  
            S.push(k - 2);  
        }  
    }  
    return result;  
}
```

# Loop Invariant for Correctness

```
int fib(int n) {  
    int result = 0;  
    Stack S;  
    S.push(n);  
    while(!S.is_empty()) {  
        // Invariant: ??  
        int k = S.pop();  
        if(k <= 2) result++;  
        else {  
            S.push(k - 1);  
            S.push(k - 2);  
        }  
    }  
    return result;  
}
```

What is the loop invariant?

# Loop Invariant for Correctness

```
int fib(int n) {  
    int result = 0;  
    Stack S;  
    S.push(n);  
    while(!S.is_empty()) {  
        // Invariant:  
        int k = S.pop();  
        if(k <= 2) result++;  
        else {  
            S.push(k - 1);  
            S.push(k - 2);  
        }  
    }  
    return result;  
}
```

Prove Invariant using induction.

Base (zero iterations):  $\left( \text{result} + \sum_{k \text{ on Stack}} \text{fib}_k \right) = 0 + \text{fib}_n.$

# Loop Invariant for Correctness

```
int fib(int n) {  
    int result = 0;  
    Stack S;  
    S.push(n);  
    while(!S.is_empty()) {  
        // Invariant:  
        int k = S.pop();  
        if(k <= 2) result++;  
        else {  
            S.push(k - 1);  
            S.push(k - 2);  
        }  
    }  
    return result;  
}
```

Prove Invariant using induction.

Ind. Step: If  $k \leq 2$  then  $\text{result}$  increases by  $\text{fib}_k = 1$  and  $k$  is removed from stack. Invariant preserved. If  $k > 2$  then  $k$  is replaced by  $k - 1$  and  $k - 2$  on stack. Since  $\text{fib}_k = \text{fib}_{k-1} + \text{fib}_{k-2}$ , invariant preserved.

# Loop Invariant for Correctness

```
int fib(int n) {  
    int result = 0;  
    Stack S;  
    S.push(n);  
    while(!S.is_empty()) {  
        // Invariant:  
        int k = S.pop();  
        if(k <= 2) result++;  
        else {  
            S.push(k - 1);  
            S.push(k - 2);  
        }  
    }  
    return result;  
}
```

Prove Invariant using induction.

End: When loop terminates, stack is empty so  $\text{result} = \text{fib}_n$ .

# Recursion vs. Iteration

Which one is more elegant? Recursion or iteration?

# Recursion vs. Iteration

Which one is more efficient? Recursion or iteration?



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

Recursive Fibonacci:

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

Lower bound analysis

$$T(n) \geq \begin{cases} b & \text{if } n = 0, 1 \\ T(n-1) + T(n-2) + c & \text{if } n > 1 \end{cases}$$

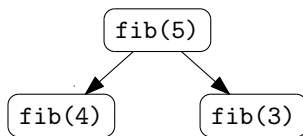
$$T(n) \geq b\varphi^{n-1}$$

where  $\varphi = (1 + \sqrt{5})/2$ .

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

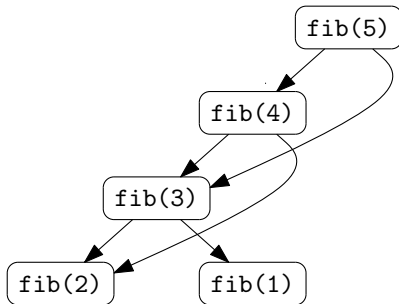
```
int fib(int n) {  
    if (n <= 2) return 1;  
    else         return fib(n-1) + fib(n-2);  
}
```

Finish the recursion tree for `fib(5)`...



## Fixing Fib with Iteration

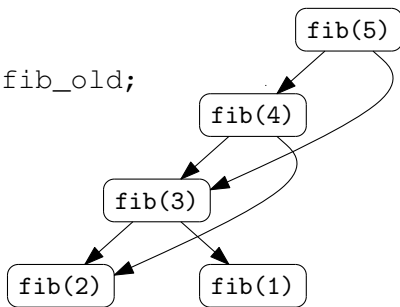
What we really want is to “share” nodes in the recursion tree:



# Fixing Fib with Iteration and “Dynamic Programming”

Here's one fix that “walks down” the left of the tree:

```
int fib_dp(int n) {  
    int fib_old = 1;  
    int fib = 1;  
    int fib_new;  
    while(n > 2) {  
        int fib_new = fib + fib_old;  
        fib_old = fib;  
        fib = fib_new;  
        --n;  
    }  
    return fib;  
}
```



## Fixing Fib with Recursion and “Memoizing”

Here's another fix that stores solutions it has calculated before:

```
// init to 0
int* fib_solns = new int[big_enough]();

fib_solns[1] = 1;
fib_solns[2] = 1;

int fib_memo(int n) {
    // If we don't know the answer, compute it.
    if(fib_solns[n] == 0)
        fib_solns[n] = fib_memo(n-1) +
                        fib_memo(n-2);
    return fib_solns[n];
}
```

# 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 more memory than you really need to store, but otherwise. . .

Neither is more efficient.

# Managing the Call Stack: Tail Recursion

```
void endlesslyGreet() {  
    cout << "Hello, world!" << endl;  
    endlesslyGreet();  
}
```

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

But...why have a call stack? There's no (need to) return to the caller.

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

# Tail Recursion

A function is “tail recursive” if for any recursive call in the function, that call is the last thing the function needs to do before returning.

In that case, why bother pushing a new activation record? There's no reason to return to the caller. Just use the current record.

That's what most compilers will do.



## Tail Recursive?

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

## Tail Recursive?

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

## Tail Recursive?

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

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

## Side Note: Tail Calls

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

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

Actually we can talk about any function call being a “tail call”, even if it’s not recursive. E.g., the call to `fact_acc` in `fact` is a tail call: no need to extend the stack.

# Eliminating Tail Recursion

```
// Search A[i..j] for key.  
// Return index of key or -1 if key not found.  
int bSearch(int A[], int key, int i, int j) {
```

```
    if(j < i) return -1;  
    int mid = (i + j) / 2;  
    if(key < A[mid])  
        return  
        bSearch(A, key, i, mid-1);  
    else if(key > A[mid])  
        return  
        bSearch(A, key, mid+1, j);  
    else return mid;  
  
}
```

```
    while(j >= i) {  
        int mid = (i + j) / 2;  
        if(key < A[mid])  
            j = mid - 1;  
        else if(key > A[mid])  
            i = mid + 1;  
        else return mid;  
    }  
    return -1;
```