Unit #3: Recursion, Induction, and Loop Invariants

CPSC 221: Basic Algorithms and Data Structures

Anthony Estey, Ed Knorr, and Mehrdad Oveisi

2016W2

Unit Outline

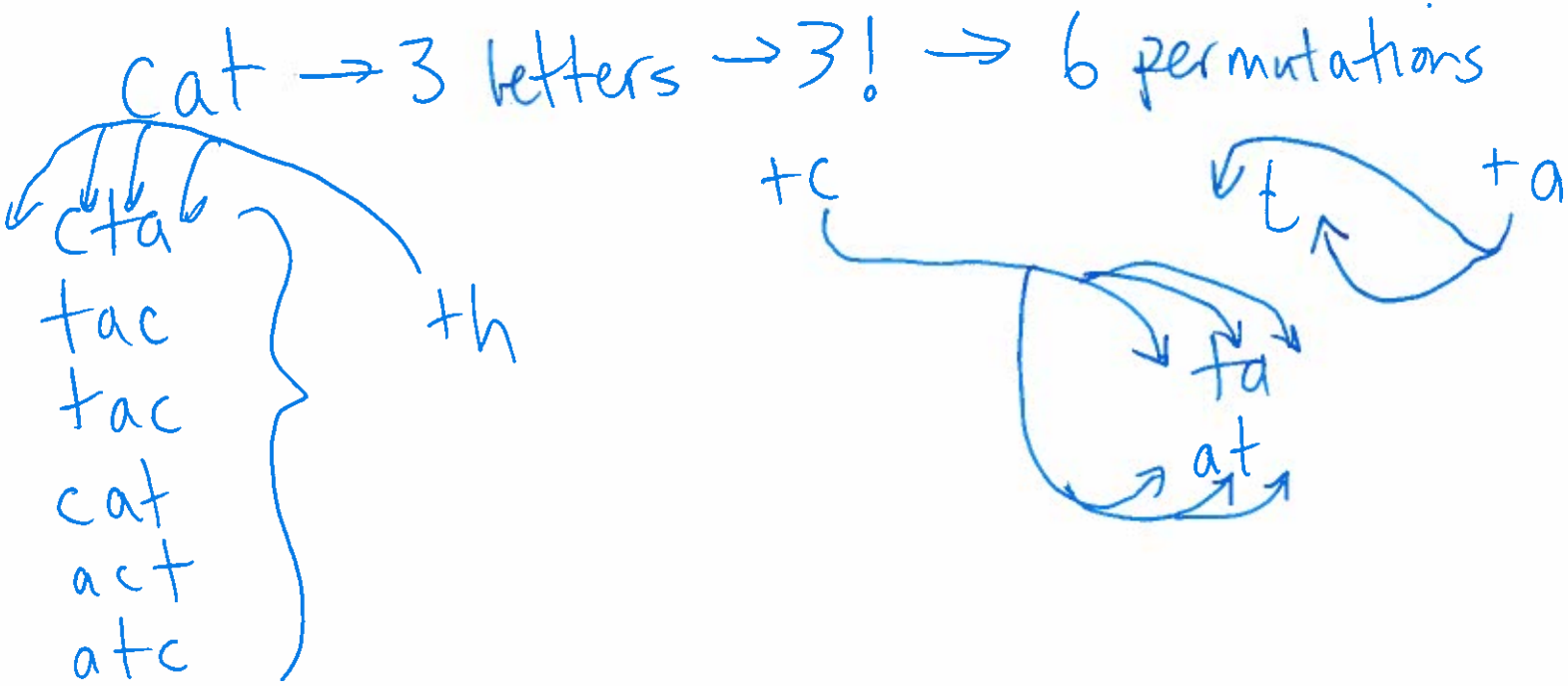
- Thinking Recursively
- Recursion Examples
- Analyzing Recursion: Induction and Recurrences
- Analyzing Iteration: Loop Invariants
- How Computers Handle Recursion
  - Recursion and the Call Stack
  - Iteration and Explicit Stacks
  - Tail Recursion

Learning Goals

- Describe the relationship between recursion and induction.
- Prove that 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.

Random Permutations (rPnastma detinoRmuo)

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



orange → 6 letters → 6! → 720 permutations

numLeaves: Count the number of leaves accessible from Node n
↳ no children tree traversal

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? ↗ Simplest case?
3. Identify and solve the (usually simple) base case(s). n is NULL ⇒ 0; n is a leaf ⇒ 1
4. Determine how to break the problem down into smaller problems of the same kind. add the # leaves in left tree with the # leaves in right subtree
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 the steps of your solution as comments, and then fill in the code for each comment.

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 output. (Every letter should be equally likely.)
2. Pick the rest of the output to be a random permutation of the remaining string (without that letter).
   ↪ no more letters to pick — we're done

It's slightly simpler if we pick a letter to be the **last** letter of the output.

cat
at
a

atc

Random Permutations (rPnastma detinoRmuo)

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

```
// randomly permute the first n characters of S
void permute(string & S, int n) {
  if( n > 1 ) {
    int i = rand() % n; // swap a random character of S
    char tmp = S[i];    // with the last character
    S[i] = S[n-1];
    S[n-1] = tmp;
    permute(S, n-1);    // randomly permute S[0..n-2]
  }
}
```

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

Induction and Recursion: Twins Separated at Birth?

Induction:	Recursion:
Base Case	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 that a Recursive Algorithm is Correct

use the structure of the recursive code to drive the inductive proof.

Just follow your code's lead and use induction.

Your base case(s)? Your code's base case(s). ← Showing and proving P(1)

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 that a Recursive Algorithm is Correct

```
// Pre: n >= 0.
// Post: returns n!
int fact(int n) {
  if (n == 0) return 1;
```

Assume: fact(n-1) = (n-1)!

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

Show that fact(n) = n!

return n * fact(n-1) = n * (n-1)*(n-2)*...*1

n * (n-1)! = n * (n-1)!

Prove: `fact(n)` $= n!$

Base case: $n = 0$:
`fact(0)` returns 1; and
$0! = 1$, by definition

Inductive Hypothesis:
`fact(n)` returns $n!$ for all $n \leq k$

Inductive Step: For $n = k + 1$, the code returns `n*fact(n-1)`. By the IH, `fact(n-1)` is $(n-1)!$ and $n! = n * (n-1)!$, by definition.

n letters ⇒ n! permutations (different way we can scramble the word)

Proving that a Recursive Algorithm is Correct

cat ⇒ cat, cta, atc, act, tac, tca

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. ← only one way to "scramble" a 1-letter word.

Induction Hypothesis: Assume that our call to `permute(S, n-1)` works (i.e., it randomly permutes (scramble) 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.

every time we will randomly choose a letter to put at the back of the string. This leaves us with n-1 letters to scramble!

Our IH assumes permute(S, n-1) will randomly scramble the remaining n-1 characters.

Recurrence Relations ... Already Covered

Unit #1

See Runtime Examples #4-6 in the Complexity lecture slides in Unit #1.

↳ Ex#5 was merge-sort. $T(n) = 2T(\frac{n}{2}) + cn$

Additional Problem: Prove that 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 the # of elements considered in the array. Thus, $n = j - i + 1$.

```
int bSearch(int A[], int key, int i, int j) {
   if (j < i) return -1;  O(1) base case
   int mid = (i + j) / 2;  O(1)
   if (key < A[mid])  O(1)
      return bSearch(A, key, i, mid-1);  T(⌊(n-1)/2⌋) ≤ T(⌊n/2⌋)
   else if (key > A[mid])  O(1)
      return bSearch(A, key, mid+1, j);  T(⌈(n-1)/2⌉) = T(⌊n/2⌋)
   else return mid;  O(1)
}
```

Binary Search Problem (Worked, cont.)

The worst-case running time:

constant (should probably be C if we are counting line numbers)

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

Guess:

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

$$T(0) = 1$$
$$T(1) = T(0) + 1 = 2$$
$$T(2) = T(1) + 1 = 3$$
$$T(3) = T(1) + 1 = 3$$
$$T(4) = T(2) + 1 = 4$$
$$T(5) = T(2) + 1 = 4$$
$$T(6) = T(3) + 1 = 4$$
$$T(7) = T(3) + 1 = 4$$
$$= 5$$
$$= 5$$
$$\vdots$$

2

4

8

Binary Search Problem (Worked, cont.)

k is even. $\lfloor \frac{8}{2} \rfloor = \lfloor \frac{9}{2} \rfloor$ $4 = 4$

k is odd $\lfloor \frac{7}{2} \rfloor \neq \lfloor \frac{8}{2} \rfloor$ $3 \neq 4$

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

Proof (by induction on n)

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

When k is even, $\lfloor \frac{k}{2} \rfloor$ is the same as $\lfloor \frac{k+1}{2} \rfloor$, but not when k is odd.

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

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

If k is even:

$= T(\frac{k}{2}) + 1 \quad (k \text{ is even})$

$\overset{(IH)}{=} (\lceil \lg(\frac{k}{2} + 1) \rceil + 1) + 1$

$= \lceil \lg(2(\frac{k}{2} + 1)) \rceil + 1$

$= \lceil \lg(k+2) \rceil + 1$

lg2

$\lg x + \lg y = \lg xy$

If k is odd:

$= T(\frac{k+1}{2}) + 1 \quad (k \text{ is odd})$

$\overset{(IH)}{=} (\lceil \lg(\frac{k+1}{2} + 1) \rceil + 1) + 1$

$= \lceil \lg(2(\frac{k+1}{2} + 1)) \rceil + 1$

$\overset{*}{=} \lceil \lg(k+3) \rceil + 1$

$= \lceil \lg(k+2) \rceil + 1 \quad (k \text{ is odd})$

annotated update

Proving that an Algorithm with Loops is Correct

come up with a "base case" and prove BC.
assume that the iterative function works for some n-1
and then try to prove/show that the function works for n.

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", that is, properties that are always true (don't vary) at particular points in the program. → prove the invariant/property at the beginning of the loop (base case)
→ want to prove it holds true for all iterations

also want to show we are making progress towards termination

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

We want to prove the invariant (show it always holds)

```
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;
  }
}
```

How long does insertionSort take?

$n-1$

$1+2+3+\ldots+n-1$

$O(n^2)$

Proving a Loop Invariant

Induction Variable: Number of times through the loop

Base Case: Prove that the invariant is true before the loop starts.

Induction Hypothesis: Assume that the invariant holds just before beginning some (unspecified) iteration.

Inductive Step: Prove that the invariant holds at the end of that iteration, for the next iteration.

Extra Bit: Make sure that the loop will eventually end!

We'll prove that Insertion Sort works, but the cool part is not proving that it actually works, but rather showing 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.

1 item is sorted!

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.

Assume invariant holds for iterations up to i

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 invariant proof for inner loops, first

Loop Termination:

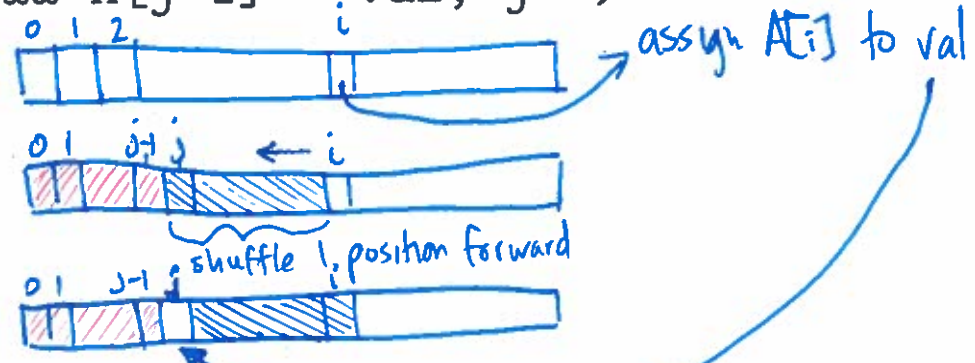
- The loop ends after `length - 1` iterations.
- When it ends, we were about to enter the (`i = length`) iteration.
- Therefore, by the newly proven invariant, when the loop ends, `A[0..length-1]` is in sorted order... which means `A` is sorted!

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;
}
```

IH says A[0..i-1] is sorted

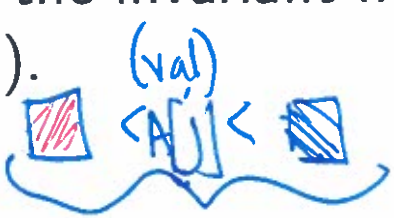
also prove this does what we are saying it's doing (invariant for inner-loop as well)



- less than val
- greater than val

Inductive Step: Prove!

- The inner loop places `val = A[i]` at the appropriate index $j < i$ by shifting elements of `A[0..i-1]` that are larger than val one position to the right.
  - from IH, we are assuming these are sorted
  - these haven't moved (still sorted)
- 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 ($i = i + 1$).



- shift moved forward, but the order hasn't changed, so they are still sorted

A[0...i] sorted when begin i+1 iteration

Practice: Proving that the Inner Loop is 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!)

↗ Invariant holds: Base case, IH, Inductive Step, Termination

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

Recursion vs. Iteration

call stack

Which one is better: recursion or iteration?

trees

list

Depends

No right answer, neither is more powerful than the other

My choice would be problem specific

↳ It's possible to solve problems either way

Simulating a Loop with Recursion

n=4

Assume foo(i) prints i

i=0 1 2 3 4

Output: 0 1 2 3

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

i=0 i=1 i=2 i=3 i=4

output: 0 1 2 3

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

where recFoo is:

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

3+1, 4

recFor(4, 4)

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.

local variables

Need to "stash" these values away and remember them when the called function finishes execution.

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

function calls

push the stuff to remember onto a stack.

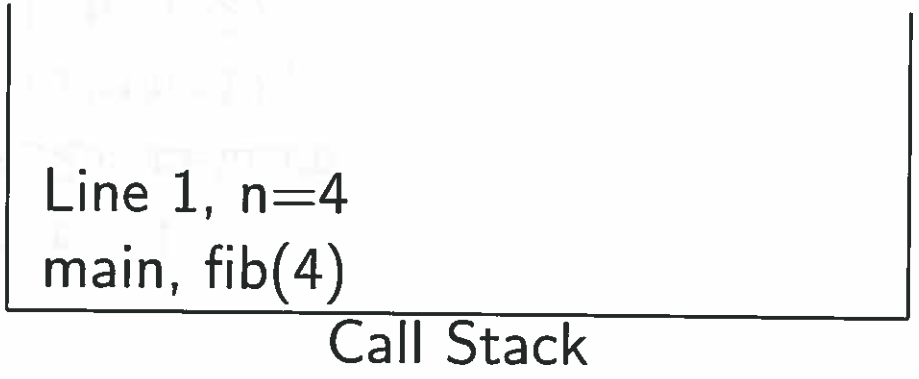
```
int main() { cout << fib(4) << endl; }
```

① call fib(4)
② print out result
③ add new line

When we call fib(4) we need to remember where to return to and "remember" a bunch of local information
↳ like local variable values.

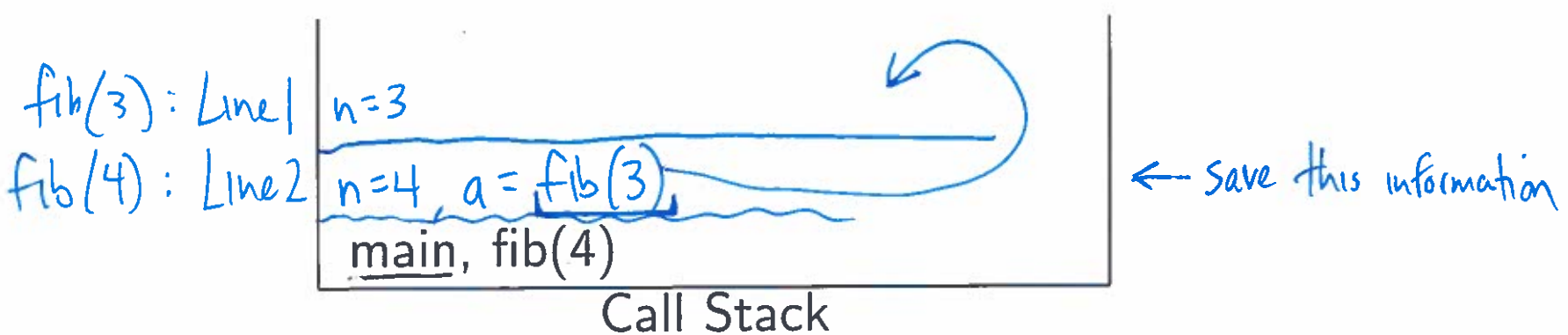
Simulating Recursion with a Stack

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

int main() { cout << fib(4) << endl; }
```



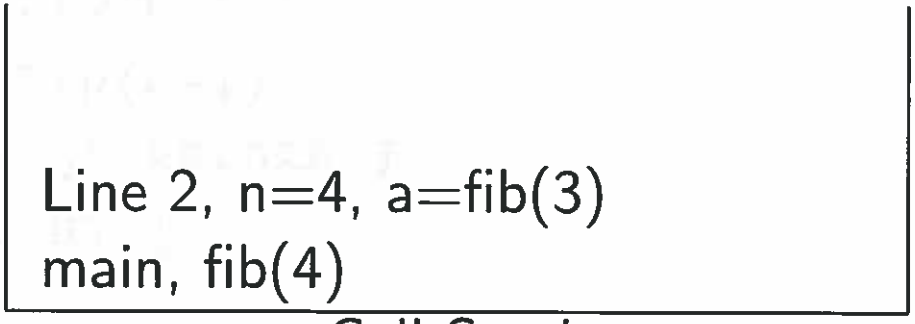
Simulating Recursion with a Stack

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

int main() { cout << fib(4) << endl; }
```



Simulating Recursion with a Stack

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

int main() { cout << fib(4) << endl; }
```

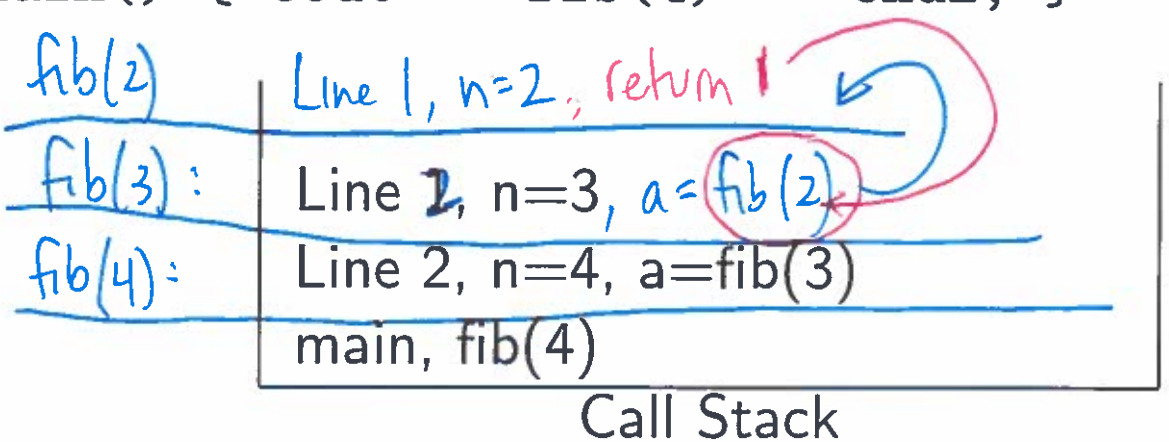


Call Stack

Simulating Recursion with a Stack

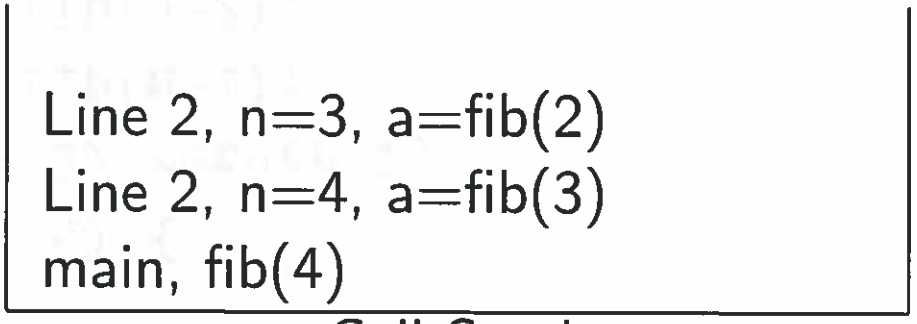
```
  int fib(int n) {
1.    if (n <= 2) return 1;
2.    int a = fib(n-1);
3.    int b = fib(n-2);
4.    return a+b;
  }
```

```
int main() { cout << fib(4) << endl; }
```



Simulating Recursion with a Stack

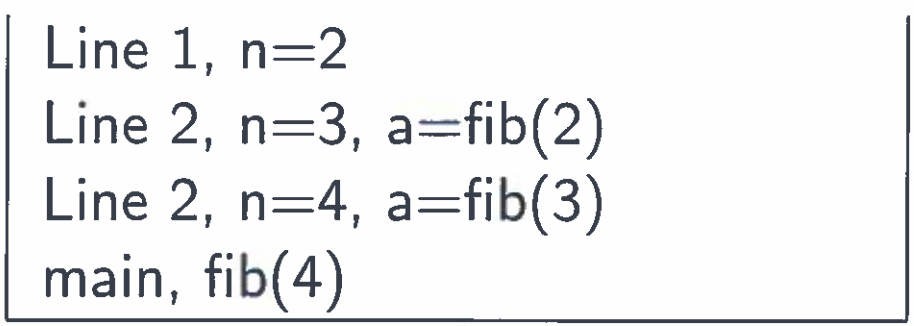
```
  int fib(int n) {
1.   if (n <= 2) return 1;
2.   int a = fib(n-1);
3.   int b = fib(n-2);
4.   return a+b;
  }

int main() { cout << fib(4) << endl; }
```



Call Stack

Simulating Recursion with a Stack

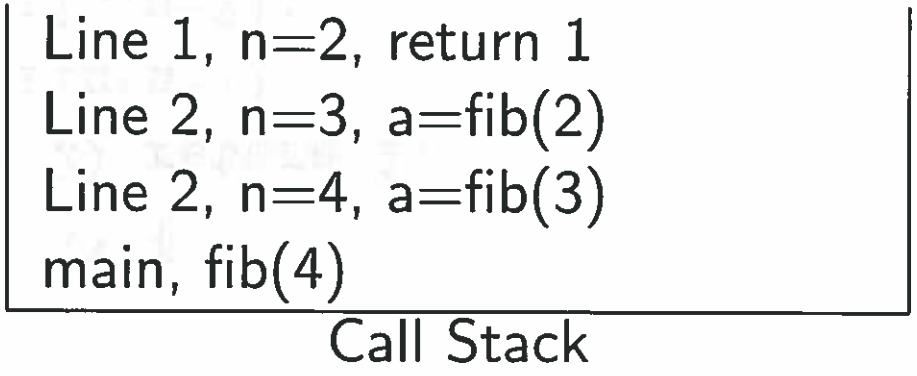
```
  int fib(int n) {
1.   if (n <= 2) return 1;
2.   int a = fib(n-1);
3.   int b = fib(n-2);
4.   return a+b;
  }

int main() { cout << fib(4) << endl; }
```



Call Stack

Simulating Recursion with a Stack

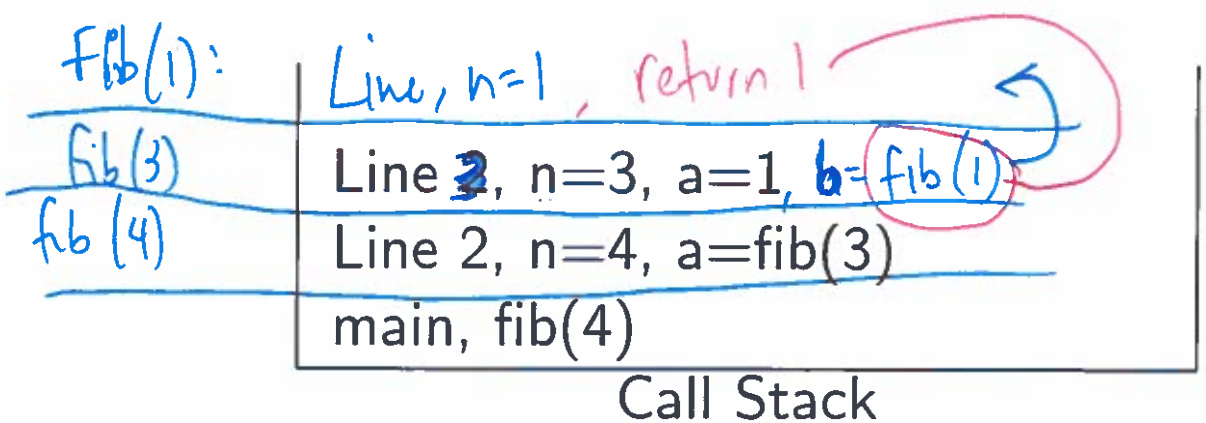
```
  int fib(int n) {
1.   if (n <= 2) return 1;
2.   int a = fib(n-1);
3.   int b = fib(n-2);
4.   return a+b;
  }

int main() { cout << fib(4) << endl; }
```



Call Stack

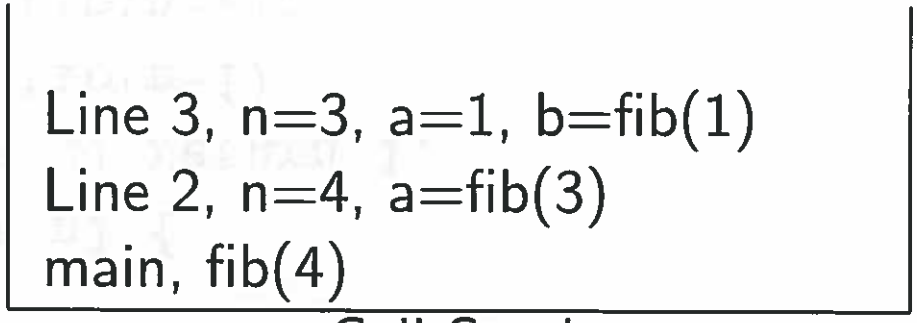
Simulating Recursion with a Stack

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

int main() { cout << fib(4) << endl; }
```



Simulating Recursion with a Stack

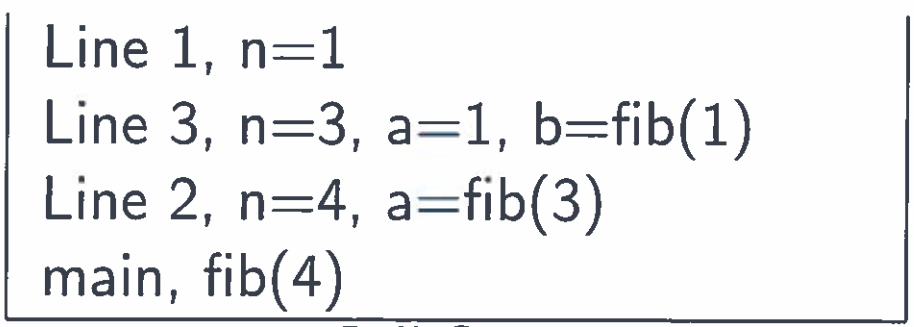
```
   int fib(int n) {
1.    if (n <= 2) return 1;
2.    int a = fib(n-1);
3.    int b = fib(n-2);
4.    return a+b;
   }

int main() { cout << fib(4) << endl; }
```



Call Stack

Simulating Recursion with a Stack

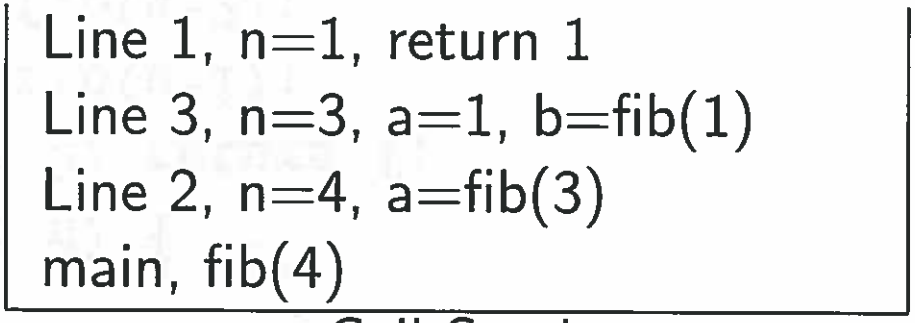
```
  int fib(int n) {
1.   if (n <= 2) return 1;
2.   int a = fib(n-1);
3.   int b = fib(n-2);
4.   return a+b;
  }

int main() { cout << fib(4) << endl; }
```



Call Stack

Simulating Recursion with a Stack

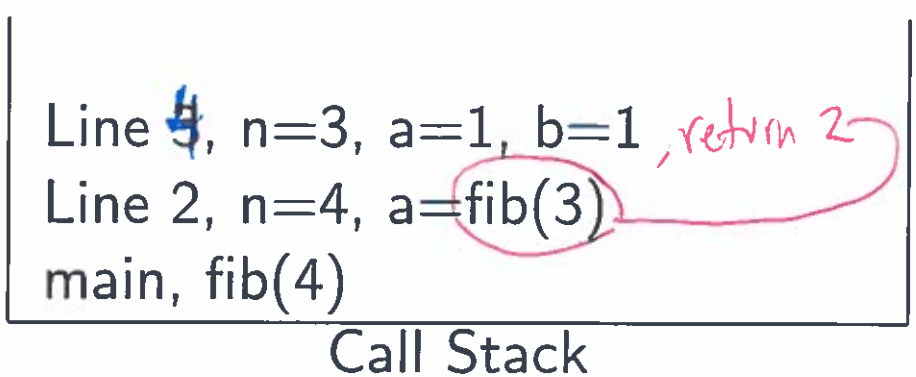
```
  int fib(int n) {
1.   if (n <= 2) return 1;
2.   int a = fib(n-1);
3.   int b = fib(n-2);
4.   return a+b;
  }

int main() { cout << fib(4) << endl; }
```



Call Stack

Simulating Recursion with a Stack

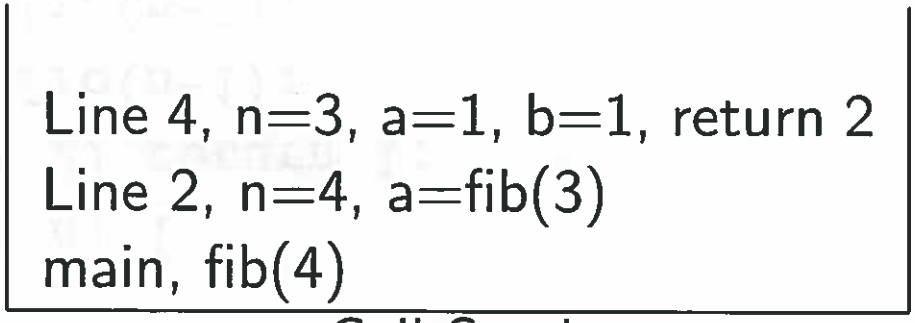
```
  int fib(int n) {
1.   if (n <= 2) return 1;
2.   int a = fib(n-1);
3.   int b = fib(n-2);
4.   return a+b;
  }

int main() { cout << fib(4) << endl; }
```



Call Stack

Simulating Recursion with a Stack

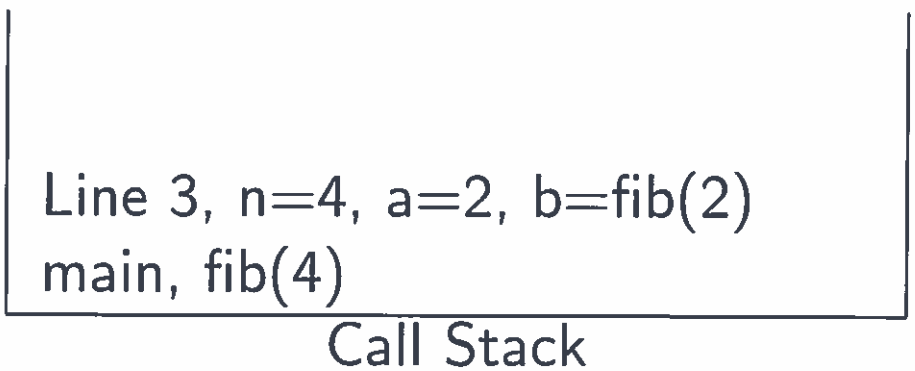
```
  int fib(int n) {
1.   if (n <= 2) return 1;
2.   int a = fib(n-1);
3.   int b = fib(n-2);
4.   return a+b;
  }

int main() { cout << fib(4) << endl; }
```



Call Stack

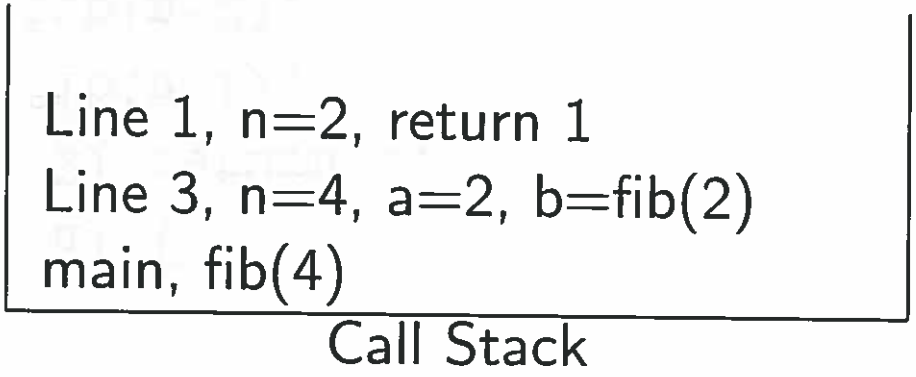
Simulating Recursion with a Stack

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

int main() { cout << fib(4) << endl; }
```



Simulating Recursion with a Stack

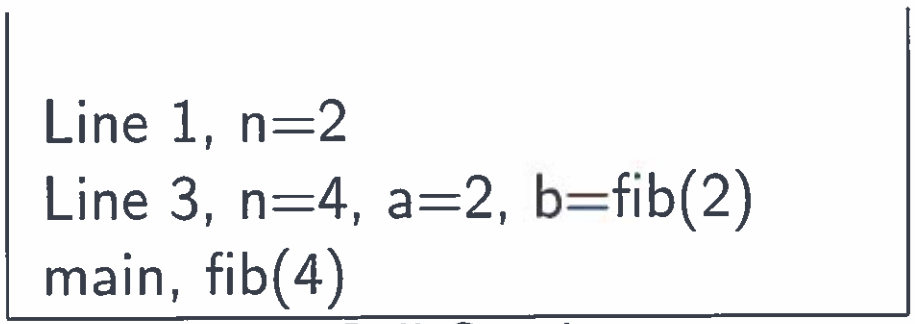
```
  int fib(int n) {
1.   if (n <= 2) return 1;
2.   int a = fib(n-1);
3.   int b = fib(n-2);
4.   return a+b;
  }

int main() { cout << fib(4) << endl; }
```



Call Stack

Simulating Recursion with a Stack

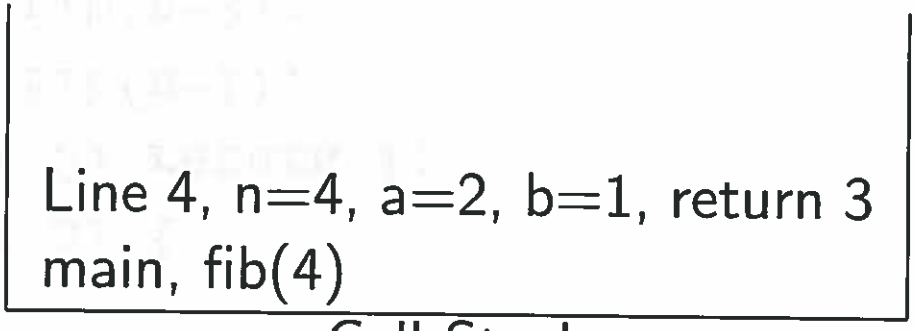
```
  int fib(int n) {
1.   if (n <= 2) return 1;
2.   int a = fib(n-1);
3.   int b = fib(n-2);
4.   return a+b;
  }

int main() { cout << fib(4) << endl; }
```



Call Stack

Simulating Recursion with a Stack

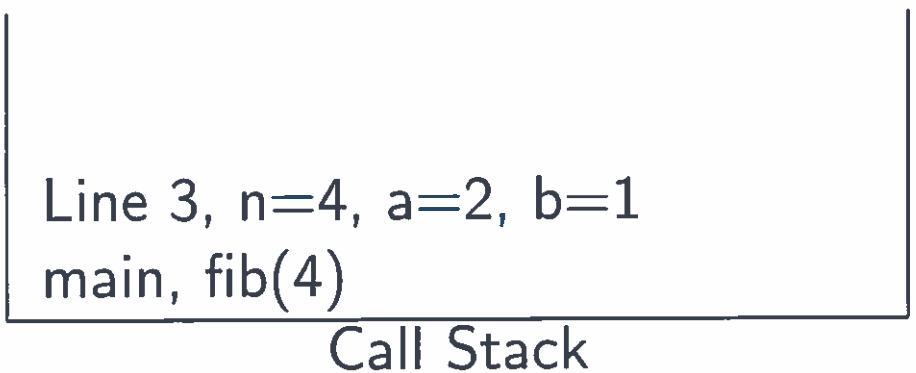
```
  int fib(int n) {
1.   if (n <= 2) return 1;
2.   int a = fib(n-1);
3.   int b = fib(n-2);
4.   return a+b;
  }

int main() { cout << fib(4) << endl; }
```



Call Stack

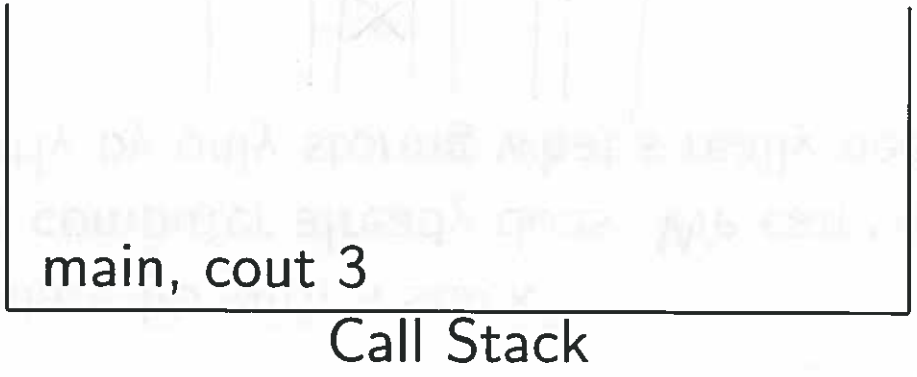
Simulating Recursion with a Stack

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

int main() { cout << fib(4) << endl; }
```

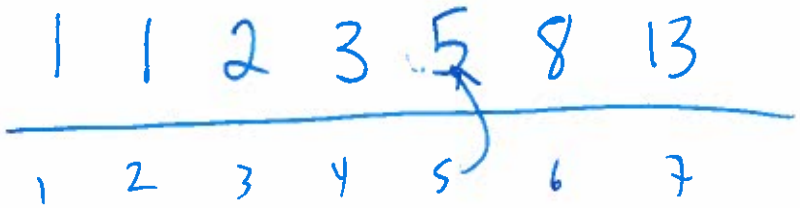


Simulating Recursion with a Stack

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

int main() { cout << fib(4) << endl; }
```



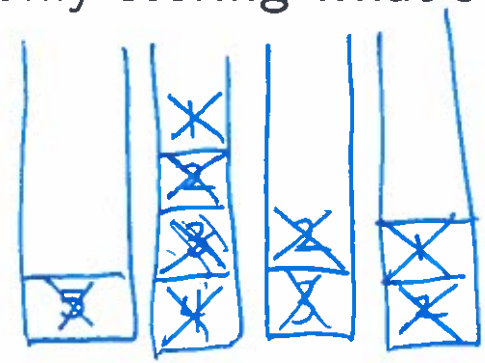
Simulating Recursion with a Stack



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:

fib(5);

```
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;
}
```



result = 5

k =

5

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

Sum of the valve of result with the sum of all of the valve on the stack fibonacci valve. Equals fibonacci of n.

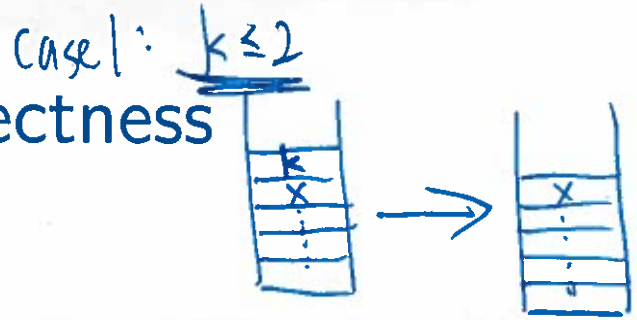
```
int fib(int n) {
  int result = 0;
  Stack S;
  S.push(n);
  while( !S.is_empty() ) {
    // Invariant:
    int k = S.pop();
    if( k <= 2 ) result++;
    else { S.push(k - 1); S.push(k - 2); }
  }
  return result;
}
```

$$\left(\text{result} + \sum_{k \text{ on Stack}} fib_k\right) = fib_n$$

Prove the invariant using induction.

Base Case (zero iterations): $\left(\overset{0}{\text{result}} + \sum_{k \text{ on Stack}}^{fib_n} fib_k\right) = 0 + fib_n.$

Loop Invariant for Correctness

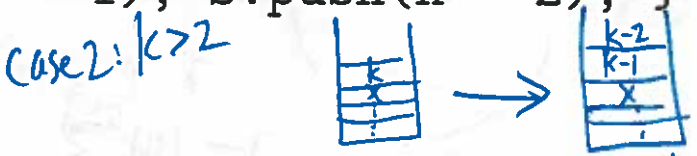



```
int fib(int n) {
  int result = 0;
  Stack S;
  S.push(n);
  while( !S.is_empty() ) {
    // Invariant:
    int k = S.pop();
    if( k <= 2 ) result++;
    else { S.push(k - 1); S.push(k - 2); }
  }
  return result;
}
```

$$\left(\text{result} + \sum_{k \text{ on Stack}} fib_k\right) = fib_n$$

$\sum_{k \text{ on } S} fib_k : y$ $\quad$ $\sum_{k \text{ on } S} fib_k : y-1$

Result: $x+y$ $\quad$ result: $x+1+y-1 = x+y$ $\quad$ $= fib_n$



result: x $\quad$ result: x

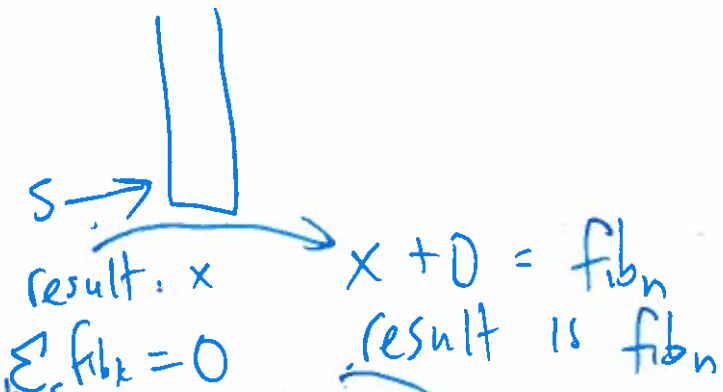
$\sum_{k \text{ on } S} fib_k = y$ $\quad$ $\sum_{k \text{ on } S} fib_k = y - fib_k + fib_{k-1} + fib_{k-2}$ $\quad$ 0 by defn of fib.

result: $x+y$ $\quad$ result: $x+y$

Prove the invariant using induction.

Inductive Step: If $k \leq 2$, then result increases by $fib_k = 1$ and k is removed from the stack. The invariant is preserved. If $k > 2$, then k is replaced by $k-1$ and $k-2$ on the stack. Since $fib_k = fib_{k-1} + fib_{k-2}$, the invariant is 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;
}
```

$$\left(\text{result} + \sum_{k \text{ on Stack}} fib_k\right) = fib_n$$

Prove the invariant using induction.

End: When the loop terminates, the stack is empty, so `result` $= fib_n$.

Recursion vs. Iteration

Which one is more *elegant*: recursion or iteration?

tree-like structures

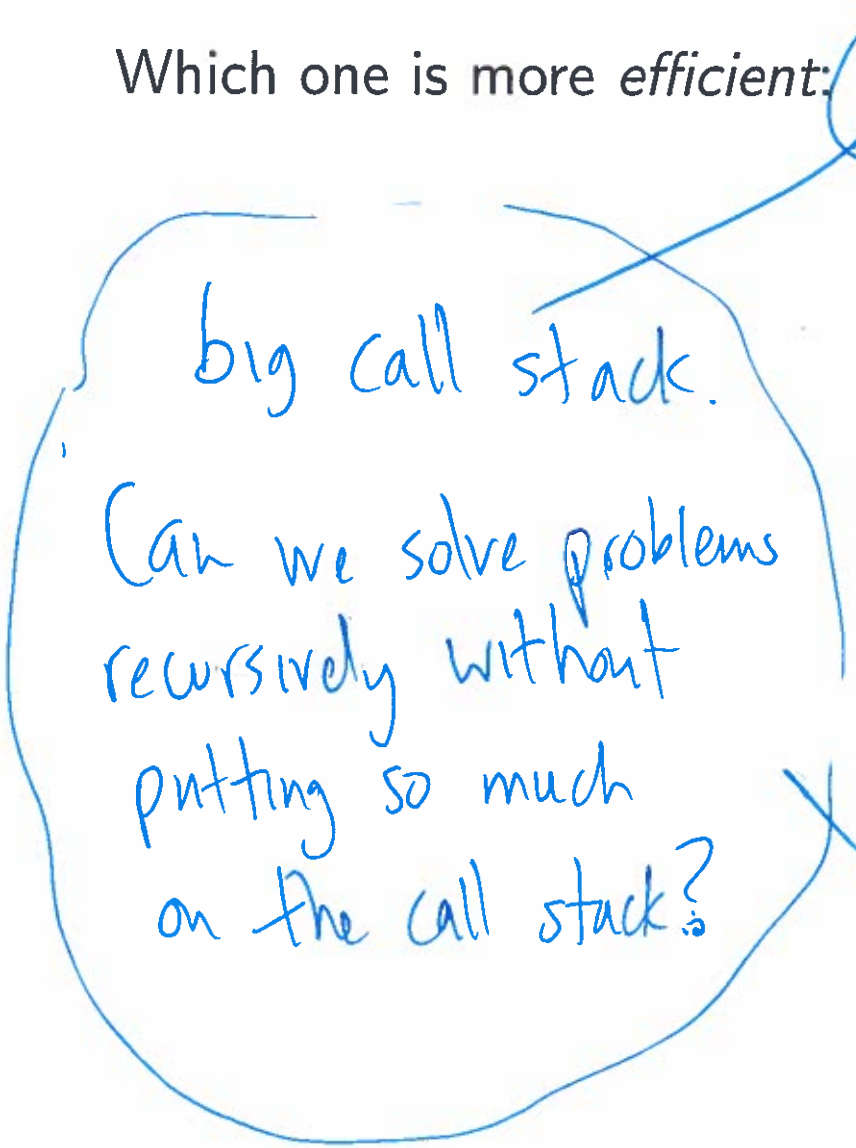
towers of hanoi

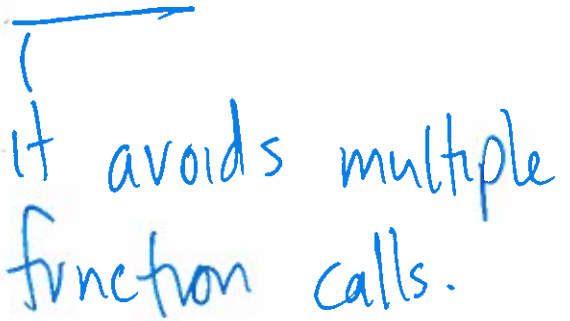
insertion-sort

linear search.

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

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

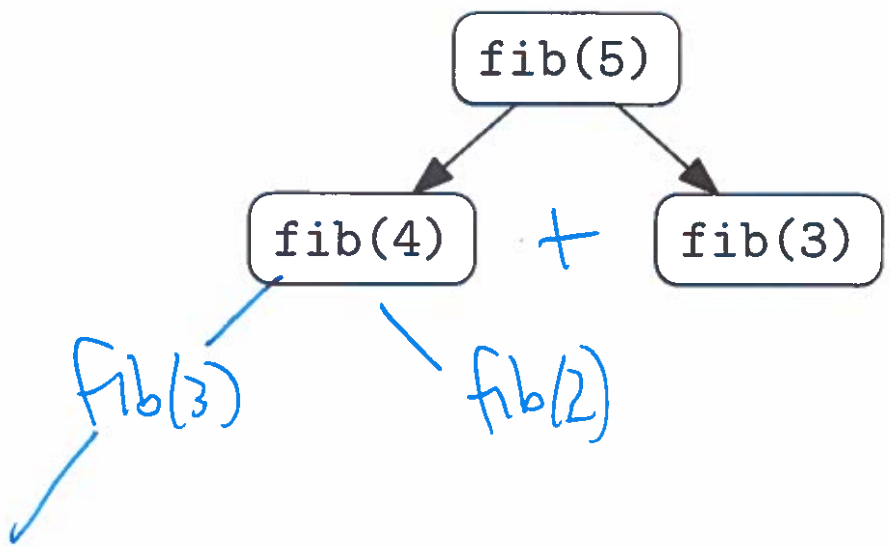
$$T(n) \geq b\varphi^{n-1}$$

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

Accidentally Making Lots of Recursive Calls; Recall ...

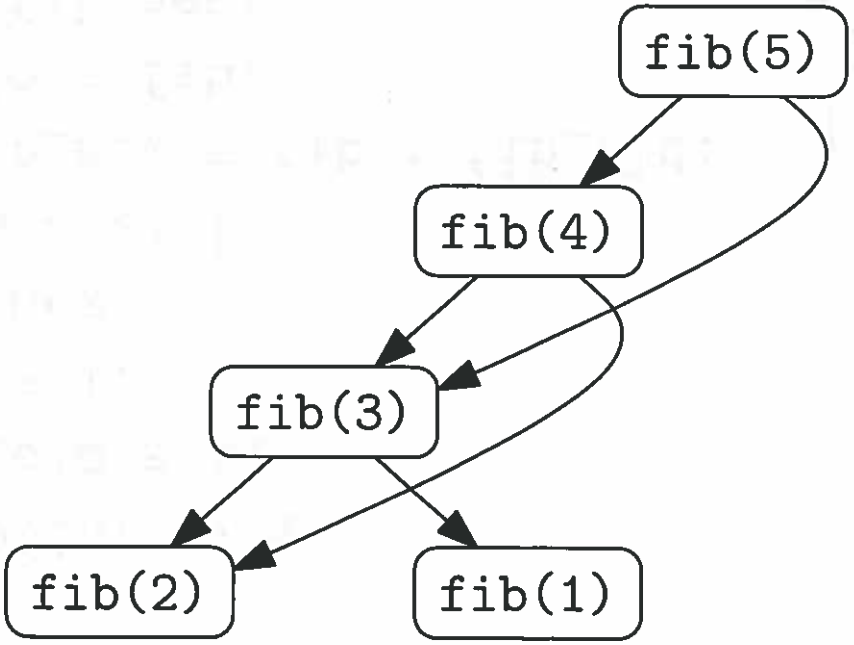
```
int fib(int n) {
    if (n <= 2) return 1;
    else        return fib(n-1) + fib(n-2);
}
```

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



Fixing `fib` with Iteration

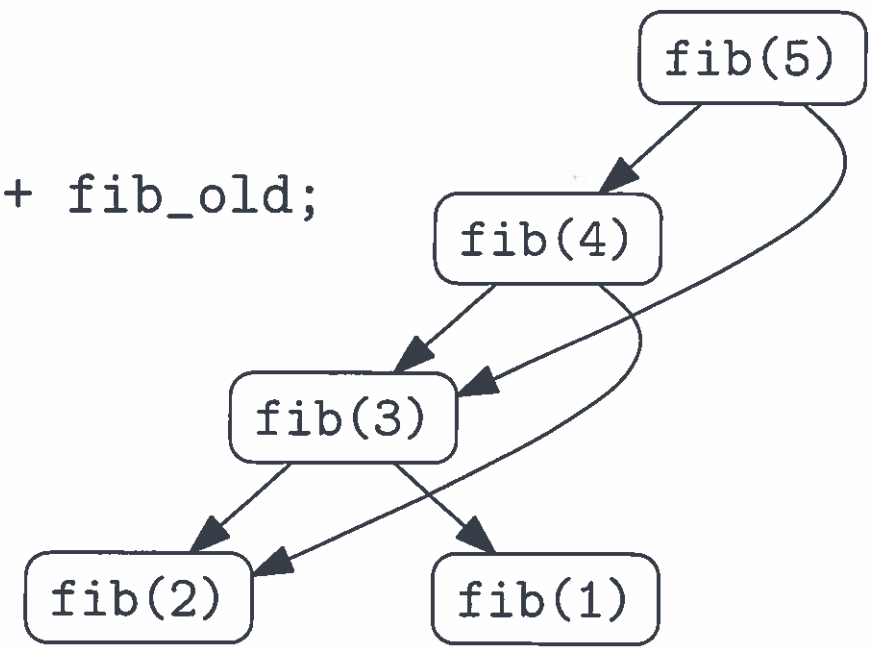
What we really want is to "share" nodes in the recursion tree:



Fixing `fib` with Iteration and "Dynamic Programming"

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

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

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

So, which one is more efficient: recursion or iteration?

It's probably easier to shoot yourself in the foot when you use recursion, and the call stack may carry around more memory than you actually need for storing things, but otherwise ...

Neither is more efficient (asymptotically).

Managing the Call Stack: Tail Recursion

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

↳ this is the last thing we do in the function

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. It won't give a stack overflow!

Tail Recursion

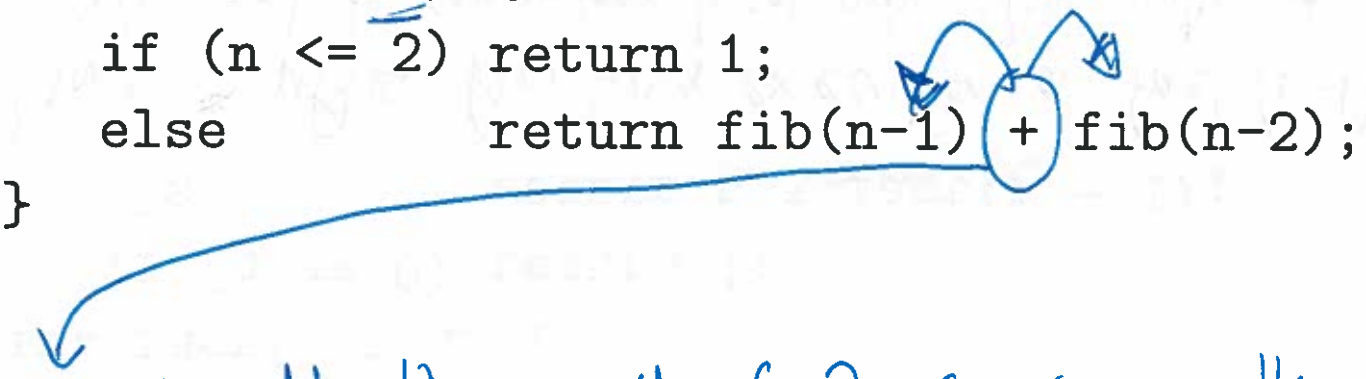
A function is "tail recursive" if for any recursive call in the function, that call is the last thing the function needs to do before returning.

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);
}
```



We must add the result of 2 recursive calls, once they are finished execution.

Tail Recursive?

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

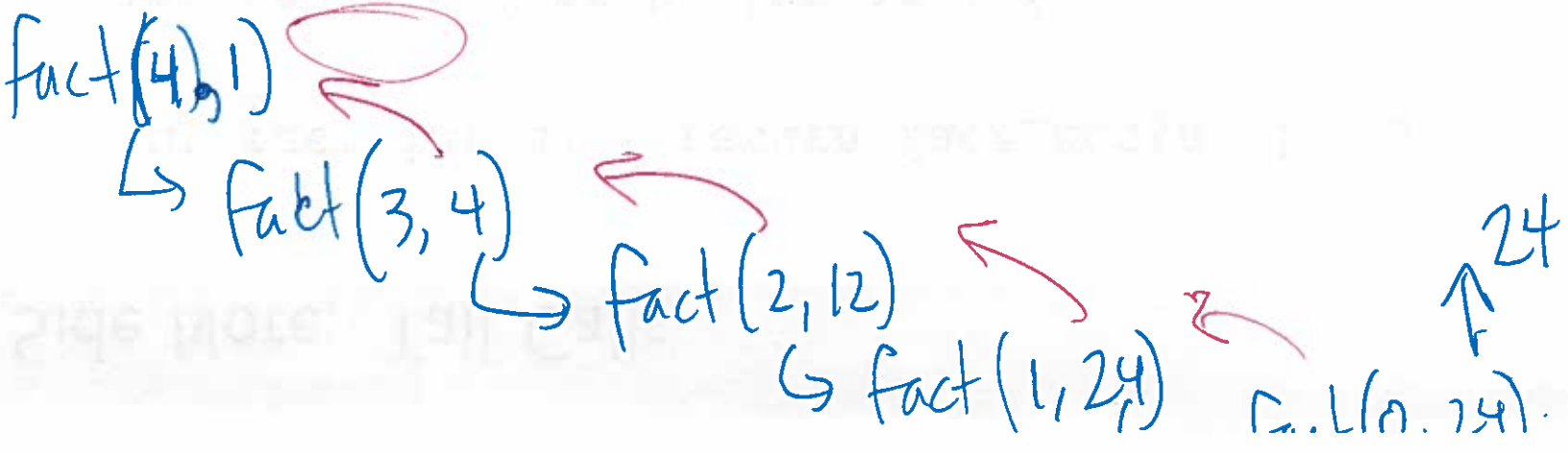
We * after finishing execution of fact(n-1).
We need to remember local info (like value of n)

Note: Koffman and Wolfgang (p. 416) call this tail recursion, but it really isn't.

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);
}
```

recursive is the last thing we do in the function
(no other operations are waiting for recursion to finish to do something with the returned result)



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 fac_acc(b - 1, acc * b);
}
```

Actually, we can talk about any function call being a "tail call", even if it's not recursive. For example, the call to `fact_acc` in `fact` is a tail call: there is 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) {
  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;
}
```

```
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;
```

Iterative Fibonacci

```
int fib_iterative(int n){
    int a=0;
    int b=1;
    int c=1;
    for(int i=0; i<n; i++){
        a=b;
        b=c;
        c=a+b;
    }
    return a;
}
```

one

what is the loop invariant?

```
int fib_tailrec(int n, int a, int b){
    if(n==0){
        return a;
    }
    return fib_tailrec(n-1, b, a+b);
}
```

next iteration:
a is set to value of b
b is set to value of (a+b)

a is initially 0

b is initially 1

fib_iterative(7); ⇒ 13

fib_tailrec(7, 0, 1); ⇒ 13

Notice the similarities between the iterative and tail-recursive solutions with respect to the values of a and b.