Unit #2: Recursion, Induction, and Loop Invariants

CPSC 221: Algorithms and Data Structures

Will Evans

2012W1
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.
Problem: Permulate 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 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];    // to the end of S.
    S[i] = S[n-1];
    S[n-1] = tmp;
    permute(S, n-1);  // randomly permute S[0..n-2]
  }
}
```

`rand() % n` returns an integer from \( \{0, 1, \ldots, n-1\} \) uniformly at random.
Recursion Example: Fibonacci numbers

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:

Rabbit population simulation:

1. Each pair of mature rabbits has a pair of rabbits per month.
2. Rabbits mature at month two.
3. Rabbits are immortal.
Recursive Fibonacci

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

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

```c
int fib(int n) {
    if (n <= 2) return 1;
    else return fib(n-1) + fib(n-2);
}
```
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.
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.
// Pre: n >= 0.
// Post: returns n!
int fact(int n) {
    if (n == 0) return 1;
    else return n*fact(n-1);
}

Prove: fact(n) = n!

Base case: n = 0
fact(0) returns 1 and 0! = 1 by definition.

Inductive hyp: fact(n) returns n! for all n ≤ k.

Inductive step: For n = k + 1, code returns n*fact(n-1).
By IH, fact(n-1) is (n − 1)! and n! = n * (n − 1)! by definition.
**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 $O(\log n)$ time.

```c
// 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;
}
```
Note: Let $n$ be $\#$ of elements considered in the array, $n = j - i + 1$. 

```c
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(\lceil n/2 \rceil)
  else if (key > A[mid]) O(1)
    return bSearch(A, key, mid+1, j); // T(\lceil n/2 \rceil)
  else return mid; // O(1) }
```
Binary Search Problem (Worked)

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

Guess:

\[ T(n) = \lceil \log_2(n + 1) \rceil + 1 \]

\[
\begin{align*}
T(0) &= 1 \\
T(1) &= T(0) + 1 = 2 \\
T(2) &= T(1) + 1 = 3 \\
T(3) &= T(1) + 1 = 3 \\
T(4) &= T(2) + 1 = 4 \\
T(5) &= T(2) + 1 = 4 \\
T(6) &= T(3) + 1 = 4 \\
T(7) &= T(3) + 1 = 4 \\
\ldots &
\end{align*}
\]
Binary Search Problem (Worked)

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

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

**Bases:**

\( T(0) = 1 = \lceil \lg(0 + 1) \rceil + 1 \)
\( T(1) = 2 = \lceil \lg(1 + 1) \rceil + 1 \)

**IH:** \( T(n) = \lceil \lg(n + 1) \rceil + 1 \) for \( n \leq k \) (for \( k > 1 \)).

**IS:** For \( n = k + 1 \), \( T(k + 1) = T(\lfloor \frac{k+1}{2} \rfloor) + 1 = \)

\[
\begin{align*}
\text{If } k \text{ is even} & \quad \text{If } k \text{ is odd} \\
& = T\left(\frac{k}{2}\right) + 1 \, (k \text{ is even}) & = T\left(\frac{k+1}{2}\right) + 1 \, (k \text{ is odd}) \\
& = (\lceil \lg\left(\frac{k}{2} + 1\right) \rceil + 1) + 1 \, (\text{IH}) & = (\lceil \lg\left(\frac{k+1}{2} + 1\right) \rceil + 1) + 1 \, (\text{IH}) \\
& = \lceil \lg(2(\frac{k}{2} + 1)) \rceil + 1 & = \lceil \lg(2(\frac{k+1}{2} + 1)) \rceil + 1 \\
& = \lceil \lg(k + 2) \rceil + 1 & = \lceil \lg(k + 3) \rceil + 1 \\
& & = \lceil \lg(k + 2) \rceil + 1 \, (k \text{ is odd})
\end{align*}
\]
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.
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!
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 \( val = A[i] \) at the appropriate index \( j \) by shifting elements of \( A[0..i-1] \) that are larger than \( val \) 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 \( (i = i + 1) \).
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 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!
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)
    }
}

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.

```c
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

```
main, fib(4)
```
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

Line 1, n=4
main, fib(4)
Simulating Recursion with a Stack

```c
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; }
```

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

Call 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

<table>
<thead>
<tr>
<th>Line 1, n=3</th>
<th>Line 2, n=4, a=fib(3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>main, fib(4)</td>
<td></td>
</tr>
</tbody>
</table>
Simulating Recursion with a Stack

```cpp
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; }
```

<table>
<thead>
<tr>
<th>Call Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line 2, n=3, a=fib(2)</td>
</tr>
<tr>
<td>Line 2, n=4, a=fib(3)</td>
</tr>
<tr>
<td>main, fib(4)</td>
</tr>
</tbody>
</table>
Simulating Recursion with a Stack

```cpp
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; }
```

| 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

```c
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; }
```

| 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) {
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

```cpp
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; }
```

<table>
<thead>
<tr>
<th>Call Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line 3, n=3, a=1, b=fib(1)</td>
</tr>
<tr>
<td>Line 2, n=4, a=fib(3)</td>
</tr>
<tr>
<td>main, fib(4)</td>
</tr>
</tbody>
</table>
Simulating Recursion with a Stack

```cpp
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; }
```

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line 1, n=1</td>
<td>main, fib(4)</td>
</tr>
<tr>
<td>Line 3, n=3, a=1, b=fib(1)</td>
<td></td>
</tr>
<tr>
<td>Line 2, n=4, a=fib(3)</td>
<td></td>
</tr>
</tbody>
</table>

Call Stack
Simulating Recursion with a Stack

```c
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;
}
```

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

<table>
<thead>
<tr>
<th>Call Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line 1, n=1, return 1</td>
</tr>
<tr>
<td>Line 3, n=3, a=1, b=fib(1)</td>
</tr>
<tr>
<td>Line 2, n=4, a=fib(3)</td>
</tr>
<tr>
<td>main, fib(4)</td>
</tr>
</tbody>
</table>

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)
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; }

<table>
<thead>
<tr>
<th>Call Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line 3, n=3, a=1, b=1</td>
</tr>
<tr>
<td>Line 2, n=4, a=fib(3)</td>
</tr>
<tr>
<td>main, fib(4)</td>
</tr>
</tbody>
</table>

26 / 44
Simulating Recursion with a Stack

```c++
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; }
```

<table>
<thead>
<tr>
<th>Call Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>main, fib(4)</td>
</tr>
<tr>
<td>Line 2, n=4, a=fib(3)</td>
</tr>
<tr>
<td>Line 4, n=3, a=1, b=1, return 2</td>
</tr>
</tbody>
</table>

Line 4, n=3, a=1, b=1, return 2
Line 2, n=4, a=fib(3)
main, fib(4)
Simulating Recursion with a Stack

```cpp
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

- Line 3, n=4, a=2, b=fib(2)
- main, fib(4)
Simulating Recursion with a Stack

```c++
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

- Line 1, n=2
- Line 3, n=4, a=2, b=fib(2)
- main, fib(4)
Simulating Recursion with a Stack

```c
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;
}
```

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

<table>
<thead>
<tr>
<th>Call Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line 1, n=2, return 1</td>
</tr>
<tr>
<td>Line 3, n=4, a=2, b=fib(2)</td>
</tr>
<tr>
<td>main, fib(4)</td>
</tr>
</tbody>
</table>
Simulating Recursion with a Stack

```cpp
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; }
```

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

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; }

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

Call Stack
Simulating Recursion with a Stack

```c
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

```
main, cout 3
```

26 / 44
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:

```c
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;
}
```
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?
int fib(int n) {
    int result = 0;
    Stack S;
    S.push(n);
    while( !S.is_empty() ) {
        // Invariant: result + ∑_{k on Stack} fib_k = fib_n
        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): result + ∑_{k on Stack} fib_k = 0 + fib_n.
int fib(int n) {
    int result = 0;
    Stack S;
    S.push(n);
    while( !S.is_empty() ) {
        // Invariant: result + \sum_{k \text{ on Stack}} fib_k = fib_n
        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 result increases by 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 fib_k = fib_{k-1} + 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: result + \( \sum_{k \text{ on Stack}} fib_k \) = fib_n
        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 result = fib_n.
Recursion vs. Iteration

Which one is more elegant? Recursion or iteration?
Recursion vs. Iteration

Which one is more efficient? Recursion or iteration?
Recursive Fibonacci:

```c
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 \(\varphi = (1 + \sqrt{5})/2\).
int fib(int n) {
    if (n <= 2) return 1;
    else return fib(n-1) + fib(n-2);
}

Finish the recursion tree for fib(5)...
What we really want is to “share” nodes in the recursion tree:

- fib(5)
- fib(4)
- fib(3)
- fib(2)
- fib(1)
Here’s one fix that “walks up” the left of the tree:

```c
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:

```java
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];
}
```
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.
Managing the Call Stack: Tail Recursion

```cpp
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 absolute 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?

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

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

```c
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);
}
```
Side Note: Tail Calls

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

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

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.
// 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;
}