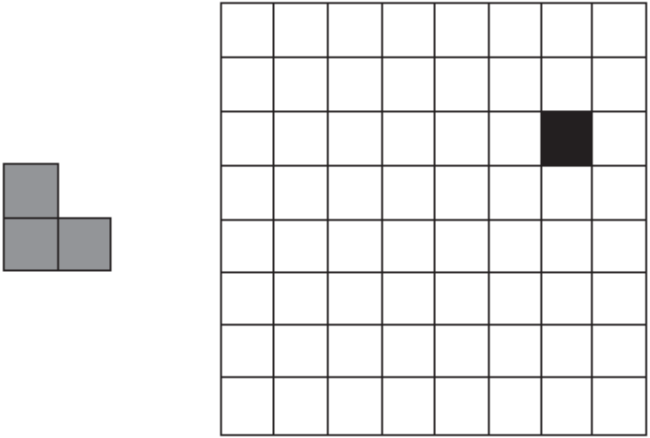
Recursion

CS 2110: Software Development Methods

April 8, 2019

Given an 8 × 8 board with one piece missing, tile with L-shaped trominoes.

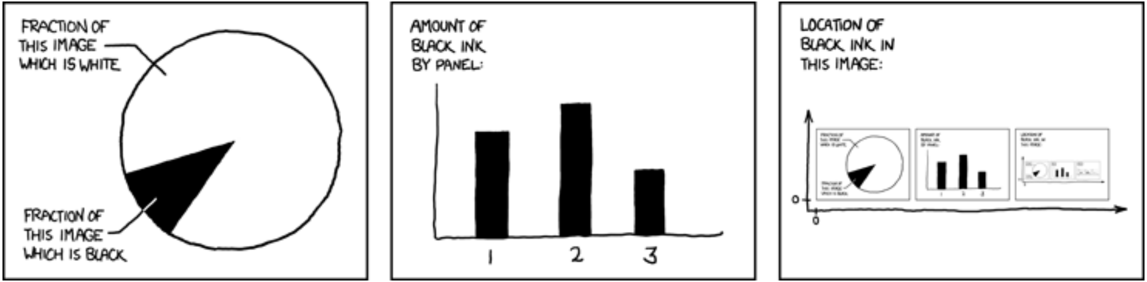


Interactive: http://goo.gl/npFQUD

Concurrency and Locking

Reviewing Friday’s In-Class Activity

Recursion



https://www.xkcd.com/688/

Recursive Mindset

- Recursion breaks a difficult problem into one or more simpler versions of itself

Recursive Mindset

Remember Binary Search? Quickly turn to page 394 (without magic)

When do we stop? How would we formalize this in pseudocode?

Recursive Mindset

Recursive Binary Search: Quickly turn to page 394 (without magic)

```
find(page_number,book):
    page = flip to middle
    if page == page_number
        return found
    if page_number is before page
        return find(page_number, first half)
    if page_number is after page
        return find(page_number, second half)
```

Definition (Don’t Write This One Down!)

Recursion

Definition (Don’t Write This One Down!)

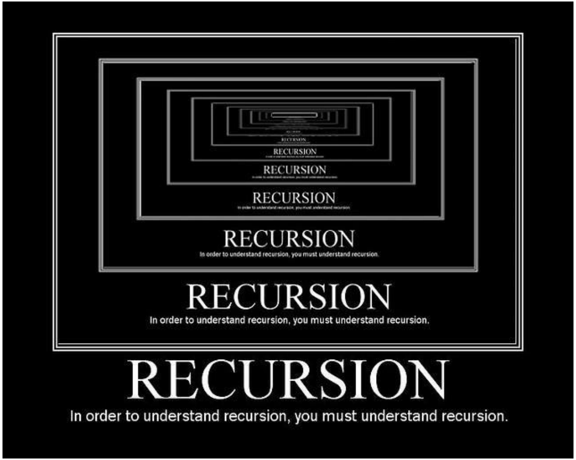
Recursion

- see recursion.

Definition (Don't Write This One Down!)

Recursion

- see recursion.



Recursion

- Recursion breaks a difficult problem into one or more simpler versions of itself
- A definition is **recursive** if it is defined in terms of itself
- Questions to ask yourself:
    - How can we reduce the problem into smaller version of the same problem?
    - How does each call make the problem smaller?
    - What is the **base case**?
    - Will we always reach the base case?

Example: Factorial

$n!$

Example: Factorial

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$$

Example: Factorial

$$n! = n \times (n-1)!$$

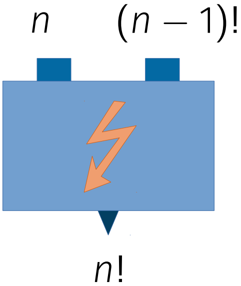
Example: Factorial

$$n! = n \times (n - 1)!$$

- Solve by multiplying two numbers

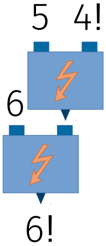
Example: Factorial

$$n! = n \times (n - 1)!$$

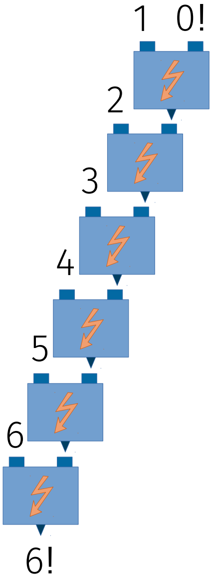


Example: Factorial

Example: Factorial



Example: Factorial



Definitions

Base case

The case for which the solution can be stated nonrecursively

Definitions

Base case

The case for which the solution can be stated nonrecursively

Recursive case

The case for which the solution is expressed in terms of a smaller version of itself

Example: Factorial

$$n! = n \times (n-1)!$$

- Base case:
- Recursive case:

Example: Factorial

$n! = n \times (n-1)!$

- Base case: $n = 0 \Rightarrow 0! = 1$
- Recursive case:

Example: Factorial

$$n! = n \times (n-1)!$$

- Base case: $n = 0 \Rightarrow 0! = 1$
- Recursive case: $n > 0 \Rightarrow n! = n \times (n-1)!$

Example: Factorial

$n! = n \times (n - 1)!$

- Base case: $n = 0 \Rightarrow 0! = 1$
- Recursive case: $n > 0 \Rightarrow n! = n \times (n - 1)!$

Advice: always put the base case first!

Translate to Code

Base case: $n = 0 \Rightarrow 0! = 1$

Translate to Code

Base case: $n = 0 \Rightarrow 0! = 1$

```
if (n == 0)
    return 1;
```

Translate to Code

Base case: $n = 0 \Rightarrow 0! = 1$

```
if (n == 0)
    return 1;
```

Magic box (recursive case): $n! = n \times (n-1)!$

Translate to Code

Base case: $n = 0 \Rightarrow 0! = 1$

```
if (n == 0)
    return 1;
```

Magic box (recursive case): $n! = n \times (n - 1)!$

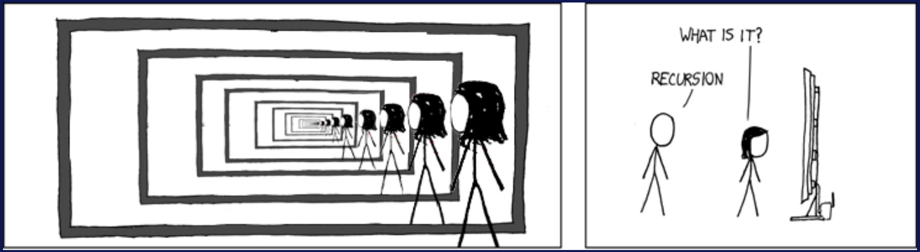
```
return n * factorial(n-1);
```

Translate to Code

```
public int factorial(int n) {
    // base case (always first)
    if (n == 0)
        return 1;

    // recursive case
    return n * factorial(n-1);
}
```

Recursion can be tricky!

Always need to stop at a base case



Translate to Code

```
public int factorial(int n) {
    // base case (always first)
    if (n == 0)
        return 1;

    // recursive case
    return n * factorial(n-1);
}
```

Translate to Code

```
public int factorial(int n) {
    // base case (always first)
    if (n <= 0)
        return 1;

    // recursive case
    return n * factorial(n-1);
}
```

Recursion vs. Iteration

Recursion

```
public int factorial(int n) {
    // base case
    if (n <= 0)
        return 1;

    // recursive case
    return n * factorial(n-1);
}
```

Iteration

Recursion vs. Iteration

Recursion

```java
public int factorial(int n) {
    // base case
    if (n <= 0)
        return 1;

    // recursive case
    return n * factorial(n-1);
}
```

Iteration

```java
public int factorial(int n) {
    int fact_n = 1;

    for (int i = 1; i <= n; i++){
        fact_n = fact_n * i;
    }
    return fact_n;
}
```

Recursion vs. Iteration

Recursion

```
public int factorial(int n) {
    // base case
    if (n <= 0)
        return 1;

    // recursive case
    return n * factorial(n-1);
}
```

Build solution from top down

Iteration

```
public int factorial(int n) {
    int fact_n = 1;

    for (int i = 1; i <= n; i++){
        fact_n = fact_n * i;
    }
    return fact_n;
}
```

Build solution from bottom up

Views of Recursion

- **Recursive definition**: definition in terms of itself, such as $n! = n \times (n-1)!$
- **Recursive procedure**: a procedure that calls itself ex: `factorial(int n)`
- **Recursive data structure**: a data structure that contains a pointer to an instance of itself:

```
public class ListNode {
    Object nodeItem;
    ListNode next, previous;
    ...
}
```

Example: Fibonacci

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ...

Example: Fibonacci

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, …

- $fib_1 = 1$
- $fib_2 = 1$
- $fib_n = fib_{n-1} + fib_{n-2}$

Example: Fibonacci

$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, \ldots$

- $fib_1 = 1$
- $fib_2 = 1$
- $fib_n = fib_{n-1} + fib_{n-2}$

Let's formalize:

- Base case:
- Recursive case:

Example: Fibonacci

$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, \ldots$

- $fib_1 = 1$
- $fib_2 = 1$
- $fib_n = fib_{n-1} + fib_{n-2}$

Let's formalize:

- Base case: $n <= 2 \Rightarrow fib_n = 1$
- Recursive case:

Example: Fibonacci

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ...

- $fib_1 = 1$
- $fib_2 = 1$
- $fib_n = fib_{n-1} + fib_{n-2}$

Let's formalize:

- Base case: $n <= 2 \Rightarrow fib_n = 1$
- Recursive case: $n > 2 \Rightarrow fib_n = fib_{n-1} + fib_{n-2}$

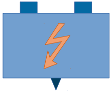
Example: Fibonacci

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ...

- $fib_1 = 1$
- $fib_2 = 1$
- $fib_n = fib_{n-1} + fib_{n-2}$

Let's formalize:

- Base case: $n <= 2 \Rightarrow fib_n = 1$
- Recursive case: $n > 2 \Rightarrow fib_n = fib_{n-1} + fib_{n-2}$



Translate to Code

Base case: $n <= 2 \Rightarrow fib_n = 1$

Translate to Code

Base case: $n <= 2 \Rightarrow fib_n = 1$

```
if (n <= 2)
    return 1;
```

Translate to Code

Base case: $n <= 2 \Rightarrow fib_n = 1$

```
if (n <= 2)
    return 1;
```

Recursive case: $n > 2 \Rightarrow fib_n = fib_{n-1} + fib_{n-2}$

Translate to Code

Base case: $n <= 2 \Rightarrow fib_n = 1$

```
if (n <= 2)
    return 1;
```

Recursive case: $n > 2 \Rightarrow fib_n = fib_{n-1} + fib_{n-2}$

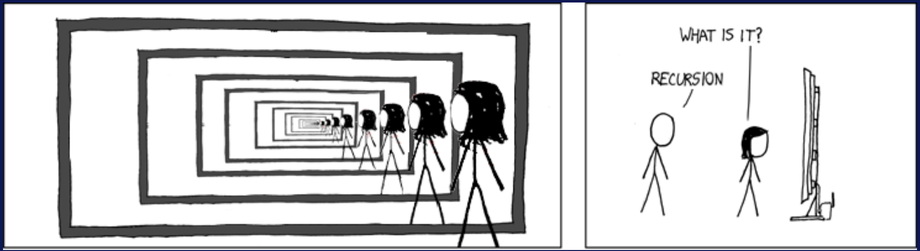
```
return fibonacci(n-1) + fibonacci(n-2);
```

Translate to Code

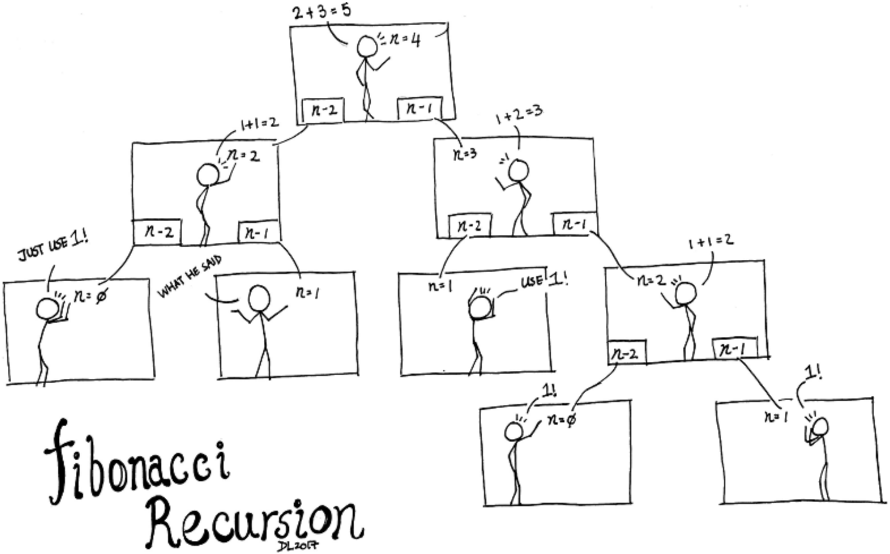
```
public int fibonacci(int n) {
    // base case (always first)
    if (n <= 2)
        return 1;

    // recursive case
    return fibonacci(n-1) + fibonacci(n-2);
}
```

Recursion can be tricky!

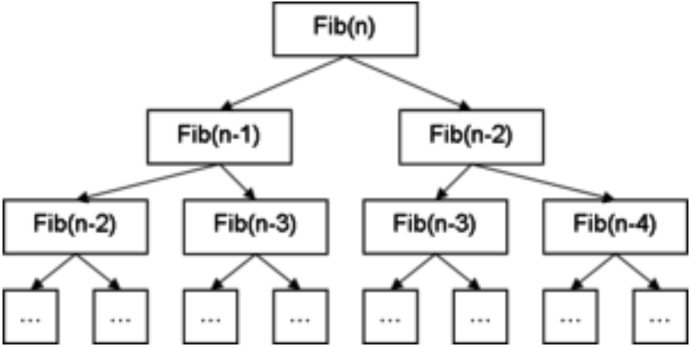
It may not be the best solution.



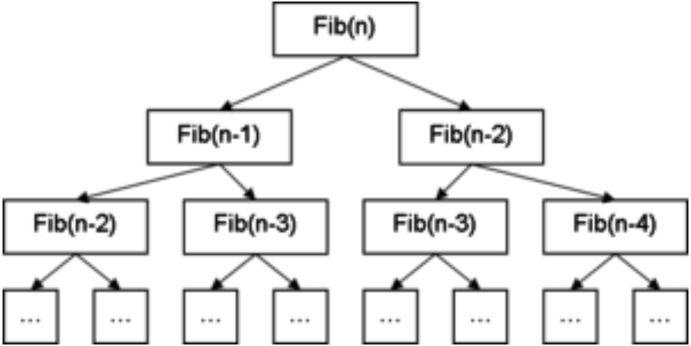
Example: Fibonacci



Example: Fibonacci

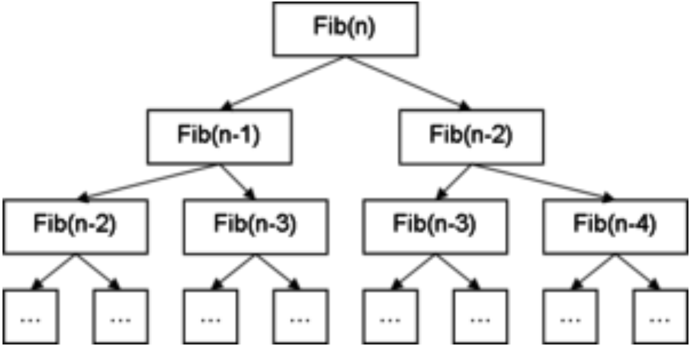


Example: Fibonacci



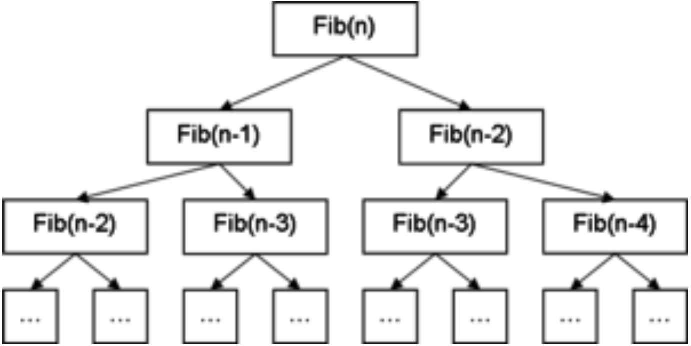
Lots of redundant calculations! We should not repeat them!

Example: Fibonacci



Lots of redundant calculations! We should not repeat them! $O(2^n)$

Example: Fibonacci



Lots of redundant calculations! We should not repeat them! $O(2^n)$

memoization: remember the things you've already calculated

Example: Iterative Fibonacci

```
public long fib(int n) {
    if ( n < 2 ) return 1;
    long answer;
    long prevFib=1, prev2Fib=1; // fib(0) & fib(1)
    for (int k = 2; k <= n; k++) {
        answer = prevFib + prev2Fib;
        prev2Fib = prevFib;
        prevFib = answer;
    }
    return answer;
}
```

Recursion vs. Iteration

- Any recursive solution may be written using iteration
- Recursive algorithm may appear simpler, more intuitive

Recursion vs. Iteration

- Any recursive solution may be written using iteration
- Recursive algorithm may appear simpler, more intuitive

Recursion

- “loop” is stopped by base case

```
recurse(int i) {
    if (i < 1)
        return;
    // do something
    recurse(i-1);
    return;
}
```

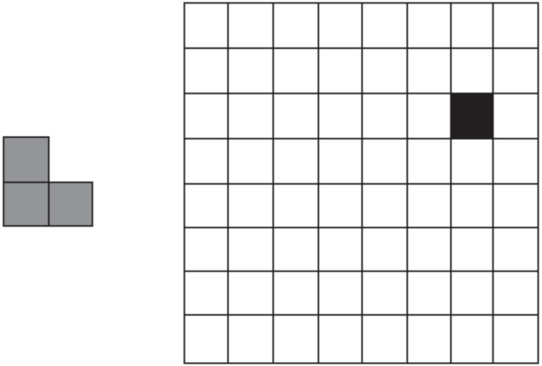
Iteration

- loop condition determines when to stop

```
while (i > 0) {
    // do something
    i--;
}
```

Tromino Puzzle

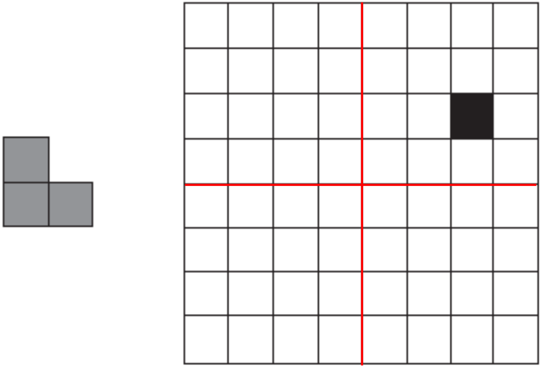
Given an 8 × 8 board with one piece missing, tile with L-shaped trominoes.



Interactive: http://goo.gl/npFQUD

Tromino Puzzle

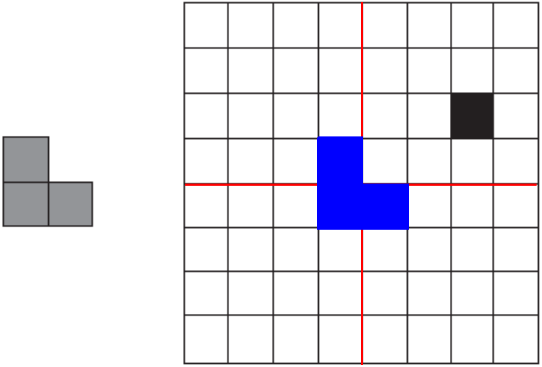
Given an 8 × 8 board with one piece missing, tile with L-shaped trominoes.



Interactive: http://goo.gl/npFQUD

Tromino Puzzle

Given an 8 × 8 board with one piece missing, tile with L-shaped trominoes.



Interactive: http://goo.gl/npFQUD

Other Recursive Examples

- Towers of Hanoi
- Euclid’s Algorithm
- Fractals

Towers of Hanoi

The objective is to transfer entire tower A to the peg B, moving only one disk at a time and never moving a larger one onto a smaller one

- The algorithm to transfer n disks from A to B in general: We first transfer $n - 1$ smallest disks to peg C, then move the largest one to the peg B and finally transfer the $n - 1$ smallest back onto largest (peg B)
- The number of necessary moves to transfer n disks can be found by $T(n) = 2^n - 1$

Euclid’s Algorithm

Calculating the greatest common divisor (gcd) of two positive integers is the largest integer that divides evenly into both of them

- E.g. greatest common divisor of 102 and 68 is 34 since both 102 and 68 are multiples of 34, but no integer larger than 34 divides evenly into 102 and 68
- Logic: If p > q, the gcd of p and q is the same as the gcd of q and $p\%q$ (where % is the remainder operator)
- Stop recursion once q becomes zero; at which point return p

Summary

- Recursion breaks a difficult problem into one or more simpler versions of itself
- Recursive definition: A definition in which something is defined in terms of smaller versions of itself
- Recursive problem can be broken into two parts:
    - Base case: The case for which the solution can be stated nonrecursively
    - Recursive case: The case for which the solution is expressed in terms of a smaller version of itself

Summary

Recursion is tricky!

- Always put base case first
- Base case should eventually happen given any input
- Recursive solution may not always be the best