csci 210: Data Structures

Recursion

Summary

- Topics
  - recursion overview
  - simple examples
  - Sierpinski gasket
  - counting blobs in a grid
  - Hanoi towers

- READING:
  - GT textbook chapter 3.5

Recursion

- A method of defining a function in terms of its own definition
- Example: the Fibonacci numbers
  - f (n) = f(n-1) + f(n-2)
  - f(0) = f(1) = 1 ← base case
- In programming recursion is a method call to the same method. In other words, a recursive method is one that calls itself.

- Why write a method that calls itself?
- Recursion is a good problem solving approach
  - solve a problem by reducing the problem to smaller subproblems; this results in recursive calls.
- Recursive algorithms are elegant, simple to understand and prove correct, easy to implement
  - But! Recursive calls can result in a an infinite loop of calls
    - recursion needs a base-case in order to stop
- Recursion (repetitive structure) can be found in nature
  - shells, leaves

Recursive algorithms

- To solve a probleme recursively
  - break into smaller problems
  - solve sub-problems recursively
  - assemble sub-solutions

Problem solving technique: Divide-and-Conquer

```
recursive-algorithm(input) {
    //base-case
    if  (isSmallEnough(input))
        compute the solution and return it
    else
        //recursive case
        break input into simpler instances input1, input 2,...
        solution1 = recursive-algorithm(input1)
        solution2 = recursive-algorithm(input2)
        ...
        figure out solution to this problem from solution1, solution2,...
        return solution
}
```

Example

- Write a function that computes the sum of numbers from 1 to n

```
int sum (int n)
```

1. use a loop
2. recursively

Example

- Write a function that computes the sum of numbers from 1 to n

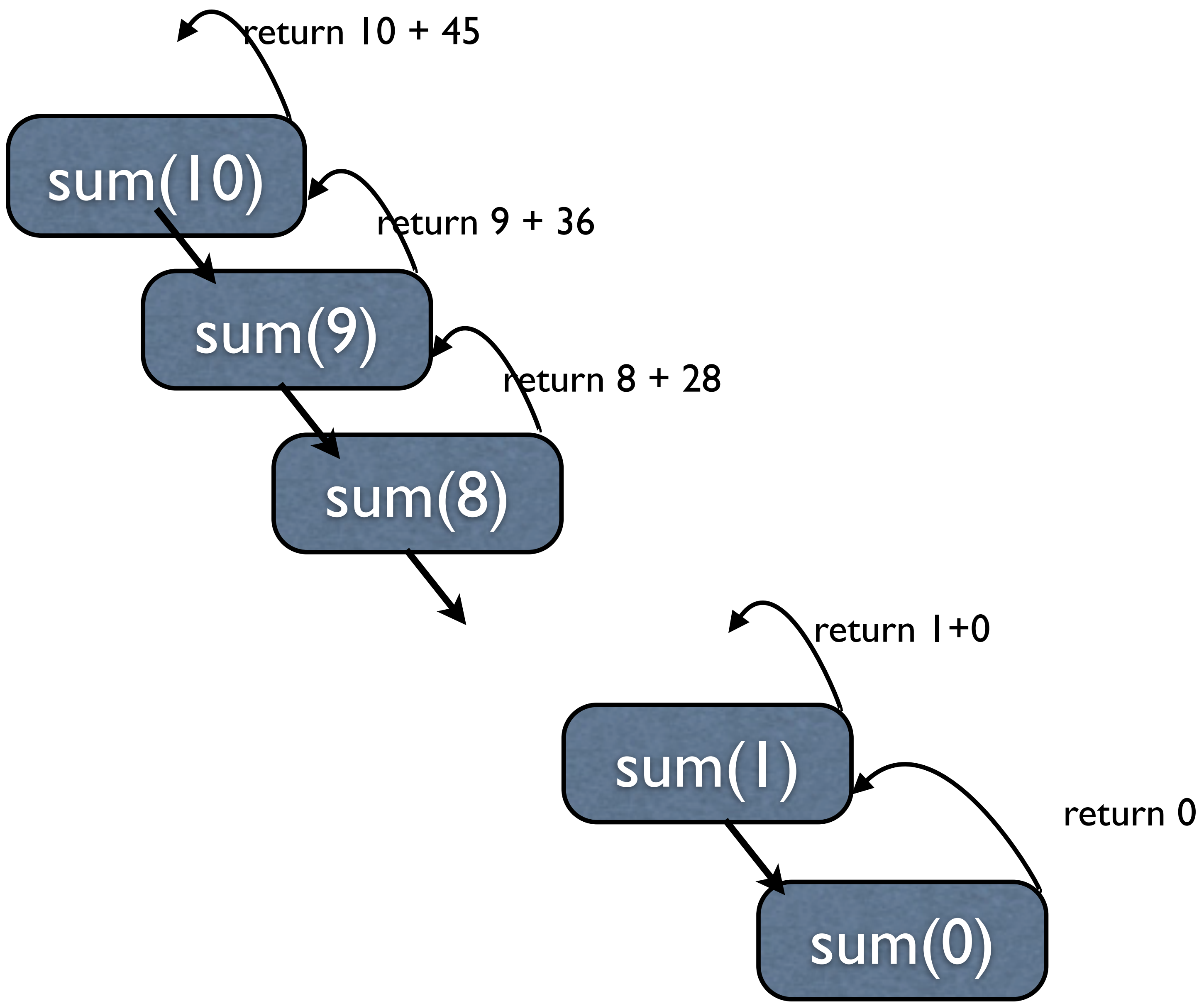
```
int sum (int n)
```

1. use a loop
2. recursively

```
//with a loop
int sum (int n) {
    int s = 0;
    for (int i=0; i<n; i++)
        s+= i;
    return s;
}
```

```
//recursively
int sum (int n) {
    int s;
    if (n == 0) return 0;
    //else
    s = n + sum(n-1);
    return s;
}
```

How does it work?


return 10 + 45
sum(10)
return 9 + 36
sum(9)
return 8 + 28
sum(8)
return 1+0
sum(1)
return 0
sum(0)

Recursion

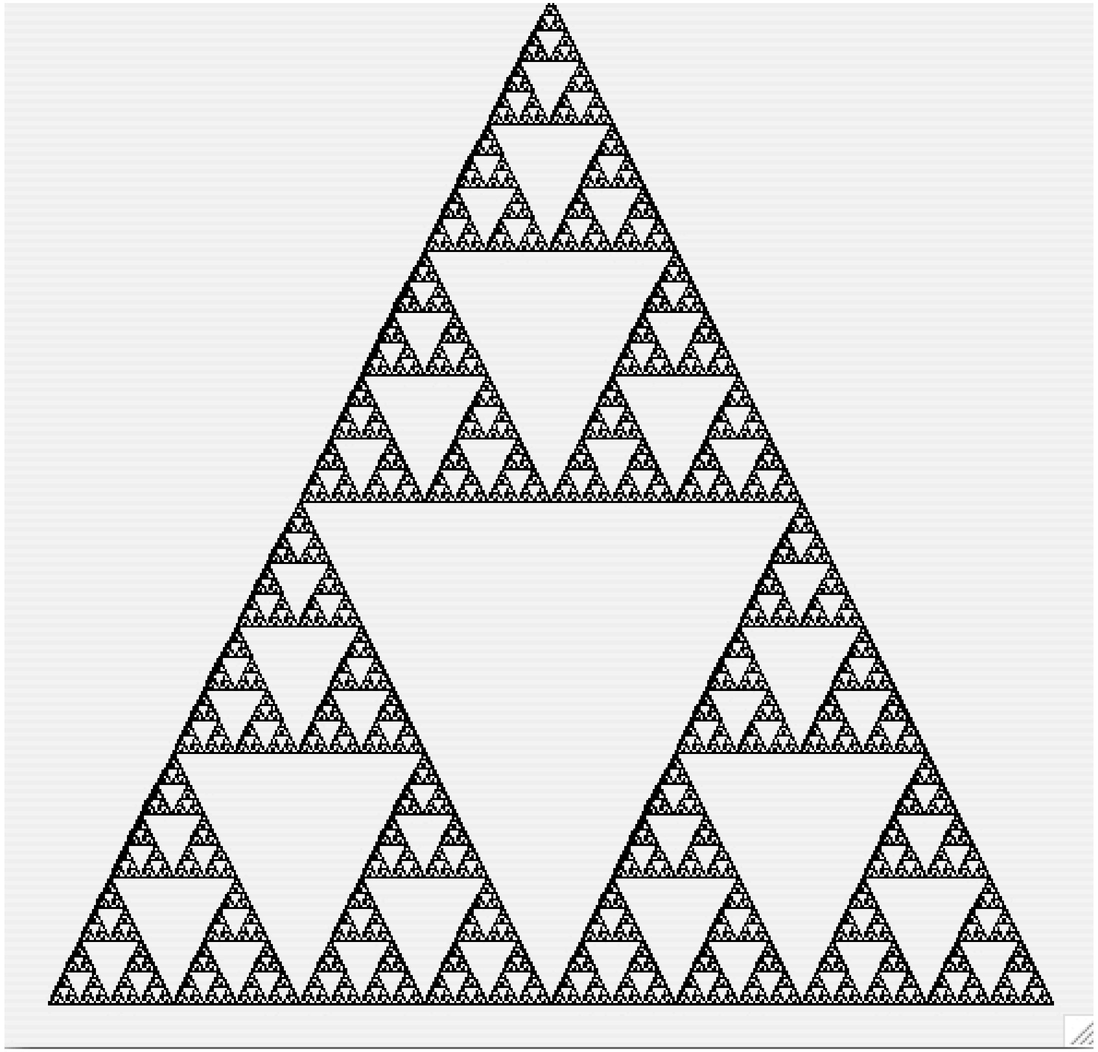
- How it works
  - Recursion is no different than a function call
  - The system keeps track of the sequence of method calls that have been started but not finished yet (active calls)
    - order matters
- Recursion pitfalls
  - miss base-case
    - infinite recursion, stack overflow
  - no convergence
    - solve recursively a problem that is not simpler than the original one

Perspective

- Recursion leads to solutions that are
  - compact
  - simple
  - easy-to-understand
  - easy-to-prove-correct
- Recursion emphasizes thinking about a problem at a high level of abstraction
- Recursion has an overhead (keep track of all active frames). Modern compilers can often optimize the code and eliminate recursion.
- First rule of code optimization:
  - Don't optimize it..yet.
- Unless you write super-duper optimized code, recursion is good
- Mastering recursion is essential to understanding computation.

Class-work: Sierpinski gasket

- Fill in the code to create this pattern



Blob check

- Problem: you have a 2-dimensional grid of cells, each of which may be filled or empty. Filled cells that are connected form a "blob" (for lack of a better word).

- Write a recursive method that returns the size of the blob containing a specified cell (i,j)

- Example

	0	1	2	3	4
0				x	x
1				x	
2	x	x			
3	x	x	x		x
4	x	x			x

BlobCount(0,3) = 3
BlobCount(0,4) = 3
BlobCount(3,4) = 1
BlobCount(4,0) = 7

- Solution ?
  - essentially you need to check the current cell, its neighbors, the neighbors of its neighbors, and so on
  - think RECURSIVELY

Blob check

- when calling BlobCheck(i,j)
  - (i,j) may be outside of grid
  - (i,j) may be EMPTY
  - (i,j) may be FILLED

- When you write a recursive method, always start from the base case
  - What are the base cases for counting the blob?
    - given a call to BlobCkeck(i,j): when is there no need for recursion, and the function can return the answer immediately ?

- Base cases
  - (i,j) is outside grid
  - (i,j) is EMPTY

Blob check

- blobCheck(i,j): if (i,j) is FILLED
  - 1 (for the current cell)
  - + count its 8 neighbors

```
//first check base cases
if (outsideGrid(i,j)) return 0;
if (grid[i][j] != FILLED) return 0;
blobc = 1
for (l = -1; l <= 1; l++)
    for (k = -1; k <= 1; k++)
        //skip of middle cell
        if (l==0 && k==0) continue;
        //count neighbors that are FILLED
        if (grid[i+l][j+k] == FILLED) blobc++;
```

X		X
X	X	
	X	

- Does not work: it does not count the neighbors of the neighbors, and their neighbors, and so on.
- Instead of adding +1 for each neighbor that is filled, need to count its blob recursively.

Blob check

- blobCheck(i,j): if (i,j) is FILLED
  - 1 (for the current cell)
  - + count blobs of its 8 neighbors

```
//first check base cases
if (outsideGrid(i,j)) return 0;
if (grid[i][j] != FILLED) return 0;
blobc = 1
for (l = -1; l <= 1; l++)
    for (k = -1; k <= 1; k++)
        //skip of middle cell
        if (l==0 && k==0) continue;
        blobc  += blobCheck(i+k, j+l);
```

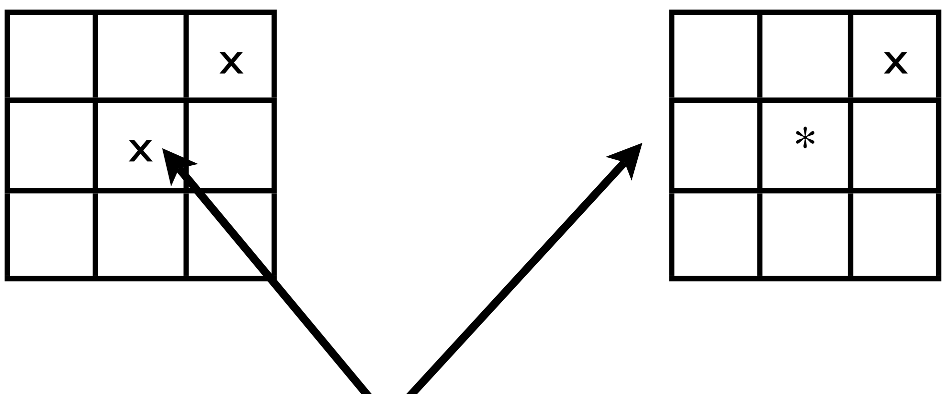
X		X
X	X	
	X	

- Example: blobCheck(1,1)
  - blobCount(1,1) calls blobCount(0,2)
  - blobCount(0,2) calls blobCount(1,1)
- Does it work?
  - Problem: infinite recursion. Why? multiple counting of the same cell

Marking your steps

- Idea: once you count a cell, mark it so that it is not counted again by its neighbors.

blobCheck(1,1)



count it and mark it

blobc=1

then find counts of neighbors, recursively

+ blobCheck(0,0)
+ blobCheck(0,1)
+blobCheck(0,2)
...

Correctness

- blobCheck(i,j) works correctly if the cell (i,j) is not filled
- if cell (i, j) is FILLED
  - mark the cell
  - the blob of this cell is 1 + blobCheck of all neighbors
  - because the cell is marked, the neighbors will not see it as FILLED
  - ==> a cell is counted only once

- Why does this stop?
  - blobCheck(i,j) will generate recursive calls to neighbors
  - recursive calls are generated only if the cell is FILLED
  - when a cell is marked, it is NOT FILLED anymore, so the size of the blob of filled cells is one smaller
  - ==> the blob when calling blobCheck(neighbor of i,j) is smaller that blobCheck(i,j)

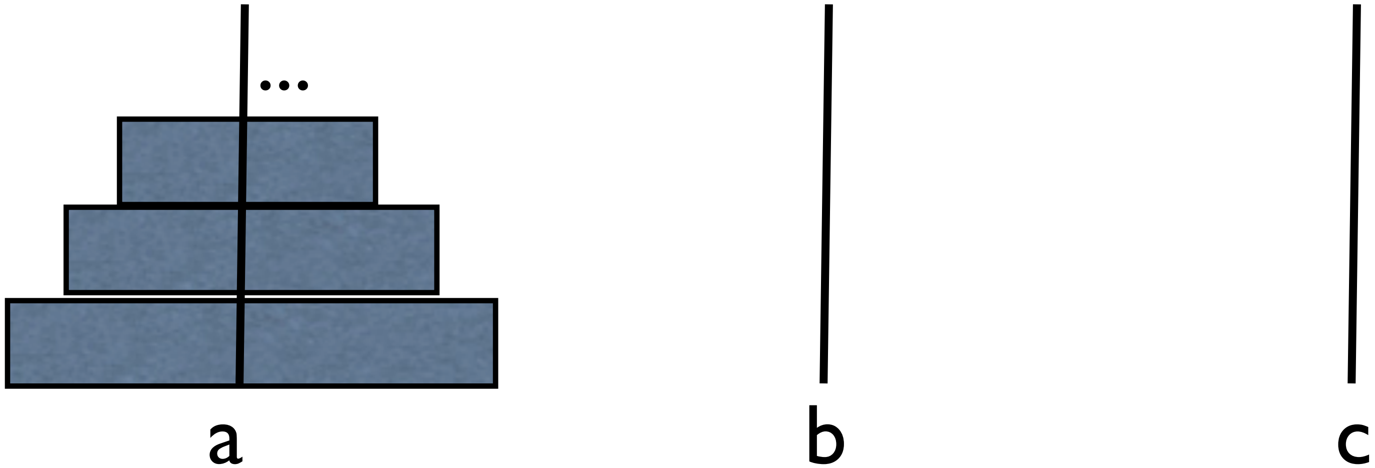
- Note: after one call to blobCheck(i,j) the blob of (i,j) is all marked
  - need to do one pass and restore the grid

Try it out!

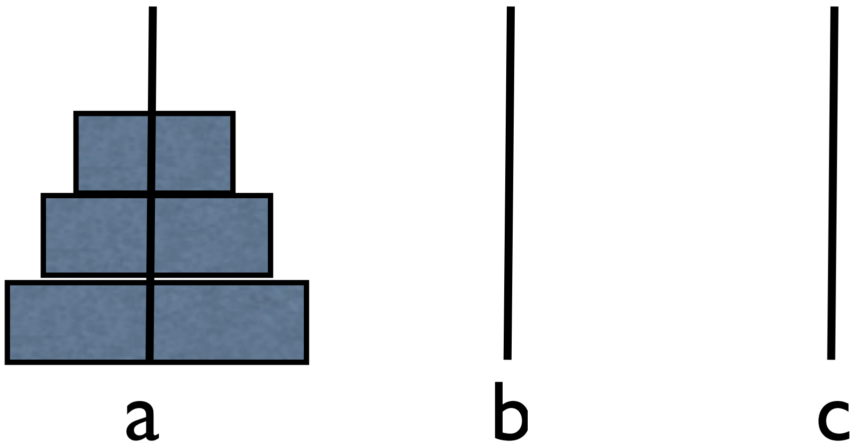
- Download blobCheckSkeleton from class website
- Fill in method blobCount(i,j)

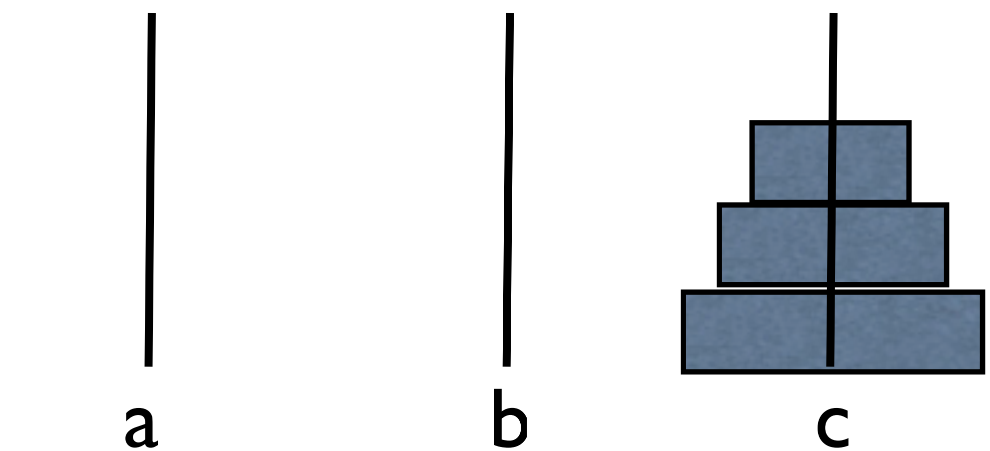
Towers of Hanoi

- Consider the following puzzle
  - There are 3 pegs (posts) a, b, c and n disks of different sizes
  - Each disk has a hole in the middle so that it can fit on any peg
  - At the beginning of the game, all n disks are on peg a, arranged such that the largest is on the bottom, and on top sit the progressively smaller disks, forming a tower
  - Goal: find a set of moves to bring all disks on peg c in the same order, that is, largest on bottom, smallest on top
    - constraints
      - the only allowed type of move is to grab one disk from the top of one peg and drop it on another peg
      - a larger disk can never lie above a smaller disk, at any time
- The legend says that the world will end when a group of monks, somewhere in a temple, will finish this task with 64 golden disks on 3 diamond pegs. Not known when they started.



Find the set of moves for n=3





Solving the problem for any n

- Problem: move n disks from A to C using B
- Think recursively.
- Can you express the problem in terms of a smaller problem?
  - Subproblem: move n-1 disks from X to Y using Z

Solving the problem for any n

- Problem: move n disks from A to C using B
- Think recursively.
- Can you express the problem in terms of a smaller problem?
    - Subproblem: move n-1 disks from X to Y using Z

- Recursive formulation of Towers of Hanoi : move n disks from A to C using B
    - move top n-1 disks from A to B
    - move bottom disks from A to C
    - move n-1 disks from B to C using A

- Correctness
    - How would you go about proving that this is correct?

Hanoi-skeleton.java

- Look over the skeleton of the Java program to solve the Towers of Hanoi
- It's supposed to ask you for n and then display the set of moves
  - no graphics

- finn in the gaps in the method

```
public void move(sourcePeg, storagePeg, destinationPeg)
```

Correctness

- Proving recursive solutions correct is done with mathematical induction
- Induction: a technique of proving that some statement is true for any n (natural number)
  - known from ancient times (the Greeks)
- Induction proof:
  - Base case: prove that the statement is true for some small value of n, usually n=1
  - The induction step: assume that the statement is true for all integers <= n-1. Then prove that this implies that it is true for n.
- Exercise: try proving by induction that $1 + 2 + 3 + ... + n = n (n+1)/2$

- Proof sketch for Towers of Hanoi:
  - Base case: It works correctly for moving one disk.
  - Assume it works correctly for moving n-1 disks. Then we need to argue that it works correctly for moving n disks.

- A recursive solution is similar to an inductive proof; just that instead of "inducting" from values smaller than n to n, we "reduce" from n to values smaller than n (think n = input size)
  - the base case is crucial: mathematically, induction does not hold without it; when programming, the lack of a base-case causes an infinite recursion loop

Analysis

- How close is the end of the world? Let's estimate running time.

- The running time of recursive algorithms is estimated using recurrent functions.
- Let $T(n)$ be the time to compute the sequence of moves to move n disks from one peg to another.
- We have
  - $T(n) = 2T(n-1) + 1$, for any $n > 1$
  - $T(1) = 1$ (the base case)

- The recurrence solves to $T(n) = O(2^n)$ [Csci 231]
  - It can be shown by induction that $T(n) = 2^n - 1$ [Math 200, Csci 231]
- This means, the running time is exponential in n
  - slow...

- Exercise:
  - 1GHz processor, $n = 64 \Rightarrow 2^{64} \times 10^{-9} =$ .... a log time; hundreds of years