

# Programming Abstractions

CS106B

Cynthia Bailey Lee  
Julie Zelenski

# Today's topics:

- Recursion Week Fortnight comes to its thrilling conclusion!
- Today:
  - › Visualizing recursive backtracking as a decision tree
  - › Applying our recursive backtracking template to new problems
  - › Theme and variations: state speller code example
- Admin
  - › Assign 4 out and due this Friday
    - Assignment parade takes a breather, Assign 5 released after diagnostic
  - › Mid-quarter diagnostic next week
    - Any 3-hour block within 24-hour window Tue Oct 26<sup>th</sup> 5pm - Wed Oct 27<sup>th</sup> 5pm
    - Sample posted on Gradescope later this week

# Backtracking template

```
bool backtrackingRecursiveFunction(args) {
```

- › Base case test for success: `return true`
  - › Base case test for failure: `return false`
  - › Loop over available options for “what to do next”:
    1. Tentatively “**choose**” one option
    2. if (“**explore**” with recursive call returns true) `return true`
    3. else That tentative idea didn’t work, so “**un-choose**” that option,  
*but don’t return false yet!--let the loop explore the other options before giving up!*
  - › None of the options we tried in the loop worked, so `return false`
- }



# One template, many applications

- **Combination lock**
  - › Goal: find combo that unlocks
  - › Choose/unchoose: {0-9} which digit to extend combo
  - › Base case: combo is full length, does it open lock?
- **Gift card**
  - › Goal: spend card down to zero
  - › Choose/unchoose: {yes-no} whether to buy item
  - › Base cases: no money on gift card, no items left to consider
- **Solve maze**
  - › Goal: exit maze
  - › Choose/unchoose: {N-S-E-W} which direction to move
  - › Base case: found exit

# Recursive exploration as “decision tree”

- **Count of horizontal branches at each decision point is *width***
  - › More branches = more options to choose from
- **Count of vertical levels is *depth***
  - › Taller tree = more decisions to make
- **Exponential growth**
  - › If  $W$  is count of options and  $D$  is count of decisions, exhaustive exploration of entire tree is  $O(W^D)$ 
    - That can be a **lot** of work...!
    - What is impact on performance of larger  $W$ ? of larger  $D$ ?
  - › How much of tree is explored to find a solution? to find all solutions?
    - How deep does function call stack get?

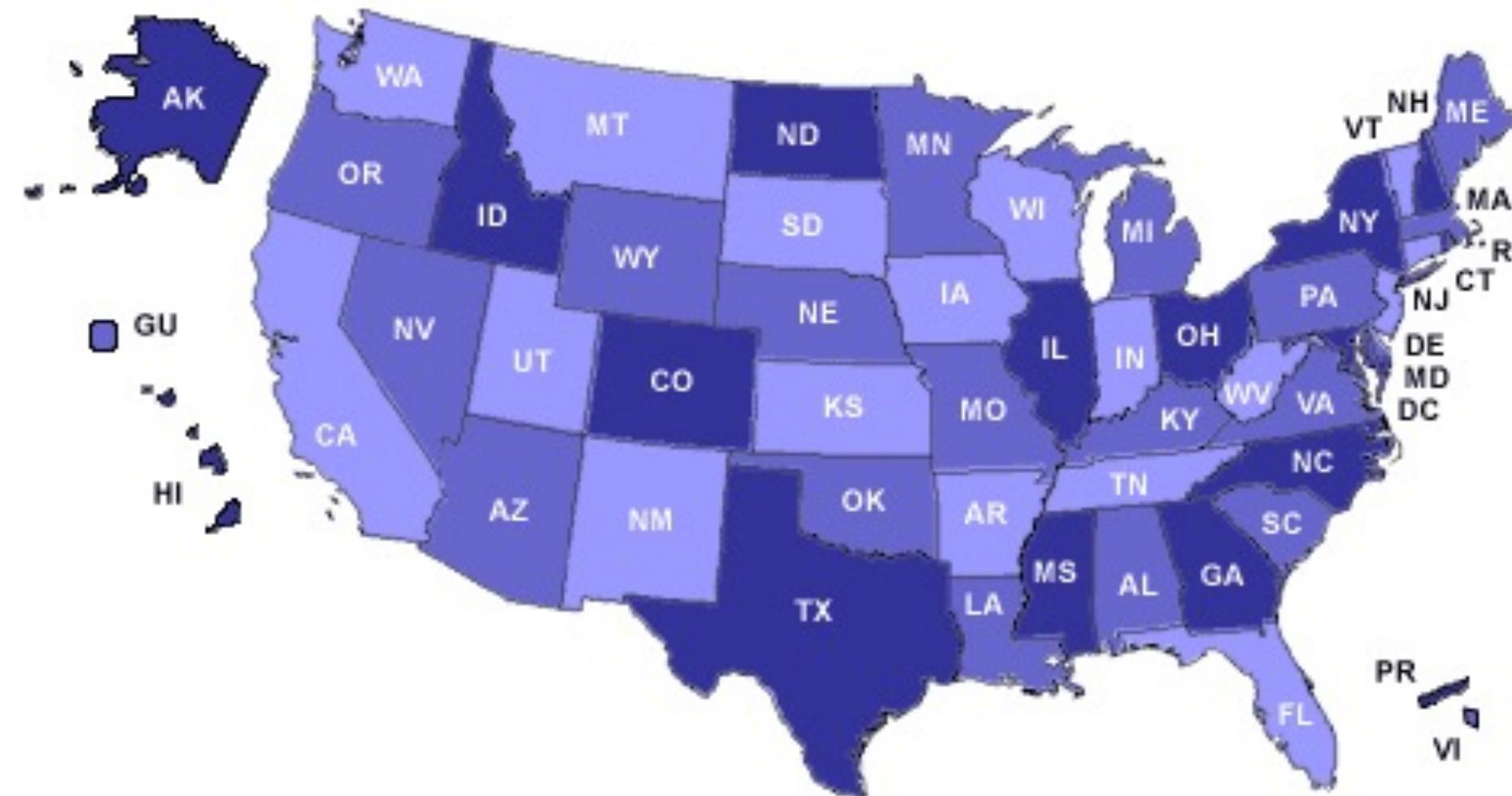
# Code Example

STATE SPELLER



# State speller

Which words can be spelled out of state postal codes? CO + DE = CODE!



# State speller

- You are given:
  - › Set<string> state: postal codes AL, CA, FL, ...
  - › Lexicon of English words
- This first version explores all combos of length n and prints those that are words

```
void printStateWords(int n, Set<string>& states, Lexicon& lex, stringsofar)
{
    if (n == 0) {
        if (lex.contains(sofar)) {
            cout <<sofar << endl;
        }
    } else {
        for (string option : states) {
            printStateWords(n - 1, states, lex, sofar + option);
        }
    }
}
```

# State speller

- **What are some variations we can apply to this code?**
  - › What do we change ...
  - › to print all words of **any** length
  - › to build a **set** of words
  - › to return **count** of words
  - › to **prune** dead ends (not valid prefix)
  - › to allow/disallow **repeat** of postal codes in word  
( combos vs permute )
  - › to stop at **first** word found, return true/false
  - › to stop at **first** word found, return **word**
  - › to return **longest** word found
  - › and many others...
- **Let's do this together in Qt!!**

# Summary: Recursive backtracking in practice

- **Identify how problem has recursive, self-similar structure**
  - › Diagram as decision tree, sequence of decisions is path down tree
  - › Nibble off one decision, recurse on rest
  - › Each decision progresses to smaller/simpler version of same problem
- **Fit to backtracking template**
  - › Base cases: success and failure
  - › Choose/explore/unchoose
- **How to model state of exploration**
  - › Update/communicate state into and out of recursive calls
  - › How to loop/enumerate options
- **Theme and variations**
  - › Print all, count, find one, find all, find optimal

# Programming Abstractions

CS106B

Cynthia Bailey Lee  
Julie Zelenski

# Today's topics:

- Recursion Week Fortnight continues!
- Today:
  - › More recursive *backtracking* code examples:
    - Gift card spending optimization
    - Maze solving

## Code Example #1

GIFT CARD SPENDING  
OPTIMIZATION



# Gift card spending optimization



- You've been given a gift card for your birthday, yay!
- The store has a rule that you must use it in one trip, and any unused balance is forfeited
- You'll be given:
  - › `Set<int> itemsForSale`: A set of prices of items for sale (assume only one of each item is in stock)
  - › `int giftCardAmt`: The amount of the gift card
- Can you find a collection of items to buy that will sum to EXACTLY the amount on the gift card??
- Return:
  - › `bool`: true if you can find such a collection, otherwise false

# Gift card spending optimization



- You've been given a gift card for your birthday, yay!
- The store has a rule that you must use it in one trip, and any unused balance is forfeited
- You'll be given:
  - › `int giftCardAmt`: The amount of the gift card
  - › `Set<int> itemsForSale`: A set of prices of items for sale (assume only one of each item is in stock)
- **Task:** Can you find a collection of items to buy that will sum to EXACTLY the amount on the gift card?
- Return:
  - › `bool`: true if you can find such a collection, otherwise false

## Your Turn:

Help me write some test cases for this function. Come up with at least one basic correctness test, and a couple tricky/edge cases. **Submit yours at [pollev.com/cs106b](http://pollev.com/cs106b).** One test case per submission, you may submit multiple times.

### Format example:

4, {1, 2, 5} = false

# Backtracking template

```
bool backtrackingRecursiveFunction(args) {
```

- › Base case test for success: `return true`
- › Base case test for failure: `return false`
- › Loop over several options for “what to do next”:
  1. Tentatively “**choose**” one option
  2. if (“**explore**” with recursive call returns true) `return true`
  3. else That tentative idea didn’t work, so “**un-choose**” that option,  
*but don’t return false yet!--let the loop explore the other options before giving up!*
- › None of the options we tried in the loop worked, so `return false`

```
}
```



# Backtracking template: applied to Gift Card problem

```
bool backtrackingRecursiveFunction(args) {
```

- › Base case test for success: `return true`      What is success for this problem?
- › Base case test for failure: `return false`
- › Loop over several options for “what to do next”:
  1. Tentatively “**choose**” one option
  2. if (“**explore**” with recursive call returns true) `return true`
  3. else That tentative idea didn’t work, so “**un-choose**” that option,  
*but don’t return false yet!--let the loop explore the other options before giving up!*
- › None of the options we tried in the loop worked, so `return false`

```
}
```

# Backtracking template: applied to Gift Card problem

```
bool backtrackingRecursiveFunction(args) {
```

- › Base case test for success: `return true`      Exactly \$0 left on card
- › Base case test for failure: `return false`
- › Loop over several options for “what to do next”:
  1. Tentatively “**choose**” one option
  2. if (“**explore**” with recursive call returns true) `return true`
  3. else That tentative idea didn’t work, so “**un-choose**” that option,  
*but don’t return false yet!--let the loop explore the other options before giving up!*
- › None of the options we tried in the loop worked, so `return false`

```
}
```

# Backtracking template: applied to Gift Card problem

```
bool backtrackingRecursiveFunction(args) {
```

- › Base case test for success: `return true`      Exactly \$0 left on card
- › Base case test for failure: `return false`      What is failure for this problem?
- › Loop over several options for “what to do next”:
  1. Tentatively “**choose**” one option
  2. if (“**explore**” with recursive call returns true) `return true`
  3. else That tentative idea didn’t work, so “**un-choose**” that option,  
*but don’t return false yet!--let the loop explore the other options before giving up!*
- › None of the options we tried in the loop worked, so `return false`

```
}
```

# Backtracking template: applied to Gift Card problem

```
bool backtrackingRecursiveFunction(args) {
```

- › Base case test for success: `return true`      Exactly \$0 left on card
- › Base case test for failure: `return false`      Overspend/negative balance, or no items left to choose.
- › Loop over several options for “what to do next.”
  1. Tentatively “**choose**” one option
  2. if (“**explore**” with recursive call returns true) `return true`
  3. else That tentative idea didn’t work, so “**un-choose**” that option,  
*but don’t return false yet!--let the loop explore the other options before giving up!*
- › None of the options we tried in the loop worked, so `return false`

```
}
```

# Backtracking template: applied to Gift Card problem

```
bool backtrackingRecursiveFunction(args) {
```

- › Base case test for success: `return true`      Exactly \$0 left on card
- › Base case test for failure: `return false`      Overspend/negative balance, or no items left to choose.
- › Loop over several options for “what to do next.”
  1. Tentatively “**choose**” one option
  2. if (“**explore**” with recursive call returns true) `return true`
  3. else That tentative idea didn’t work, so “**un-choose**” that option,  
*but don’t return false yet!--let the loop explore the other options before giving up!*
- › None of the options we tried in the loop worked, so `return false`

```
}
```

# What is “one step” in the Gift Card problem?

- We can imagine lining up all the items for sale, and our task is basically to make a binary yes/no decision for purchasing each item
  - › The yes'es and no's can come in any combination, we have to find a combination that sums to our gift card amount

Items:



\$1

Y/N:



\$5

Y/N:



\$3

Y/N:



\$2

Y/N:



\$10

Y/N:

# What is “one step” in the Gift Card problem?

- We can imagine lining up all the items for sale, and our task is basically to make a binary yes/no decision for purchasing each item
  - The yes'es and no's can come in any combination, we have to find a combination that sums to our gift card amount

Items:



\$1

Y/N: **Y**

\$5

Y/N:   

\$3

Y/N:   

\$2

Y/N:   

\$10

Y/N:   

One  
step/decision

Delegate the rest to  
recursion

# What is “one step” in the Gift Card problem?

- We can imagine lining up all the items for sale, and our task is basically to make a binary yes/no decision for purchasing each item
  - The yes'es and no's can come in any combination, we have to find a combination that sums to our gift card amount

Items:



\$1

Y/N: **Y**

\$5

Y/N:

\$3

Y/N:

\$2

Y/N:

\$10

Y/N:

One  
step/decision

If recursion comes back with the answer that no combination works for this set and the remaining funds, reconsider our Y on the banana.

# What is “one step” in the Gift Card problem?

- We can imagine lining up all the items for sale, and our task is basically to make a binary yes/no decision for purchasing each item
  - The yes'es and no's can come in any combination, we have to find a combination that sums to our gift card amount

Items:



\$1

Y/N: **Y**

\$5

Y/N:   

\$3

Y/N:   

\$2

Y/N:   

\$10

Y/N:   

One  
step/decision

Conclusion: one step/decision has two options to “loop” over: Y and N (for one item).

# Backtracking template: applied to Gift Card problem

```
bool backtrackingRecursiveFunction(args) {
```

- › Base case test for success: `return true` Exactly \$0 left on card
- › Base case test for failure: `return false` Overspend/negative balance, or no items left to choose.
- › Loop over several options for “what to do next.”
  1. Tentatively “**choose**” one option
  2. if (“**explore**” with recursive call re
  3. else That tentative idea didn’t work  
*but don’t return false yet!--let the loop continue--*
- › None of the options we tried in the loop worked, so `return false`

```
}
```

If both Y and N options for an item fail, we’ve exhausted all possibilities, so `return false`.

```

bool canUseFullGiftCard(int giftCardAmt, Set<int>& itemsForSale, Set<int>& itemsToBuy)
{
    // base case success: card amount is spent down to 0 exactly
    if (giftCardAmt == 0) {
        return true;
    }
    // base case failure: we either overspent, or we need to spend more but there are
    // no more items for sale, so we can't succeed
    if (giftCardAmt < 0 || itemsForSale.size() == 0) {
        return false;
    }

    // recursive case: consider 1 next item
    int item = itemsForSale.first();
    Set<int> newItemsForSale = itemsForSale Try both Y and N Loop over several options for "what to do next":
    itemsToBuy += item; 1. Tentatively "choose" one option
    if (canUseFullGiftCard(giftCardAmt - item, newItemsForSale)) return true; 2. if ("explore" with recursive call returns true) return true
    } 3. else That tentative idea didn't work, so "un-choose" that option,
    itemsToBuy -= item; but don't return false yet!--let the loop explore the other options before giving up
    // N: do not buy this item
    if (canUseFullGiftCard(giftCardAmt, newItemsForSale, itemsToBuy)) {
        return true;
    }
    return false;
}

```

## Comparing our solution and the design template

**bool backtrackingRecursiveFunction(args) {**

    Base case test for success: **return true**

    Base case test for failure: **return false**

    Loop over several options for "what to do next":

        1. Tentatively "choose" one option

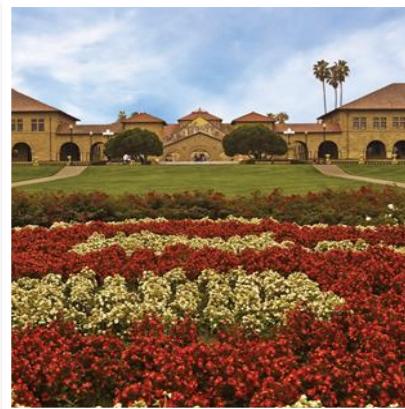
        2. if ("explore" with recursive call returns true) **return true**

        3. else That tentative idea didn't work, so "un-choose" that option,  
            but don't return false yet!--let the loop explore the other options before giving up

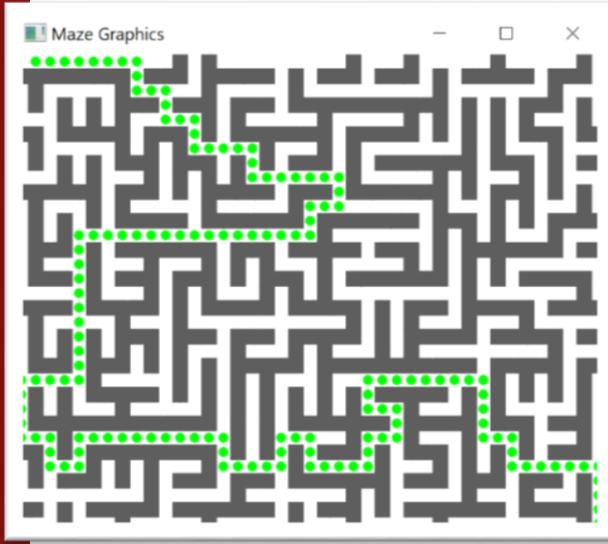
    None of the options we tried in the loop worked, so **return false**

## Code Example #2

SAY HELLO AGAIN TO YOUR  
FRIEND, ASSIGNMENT 2 MAZE



# Backtracking template: applied to Maze problem



```
bool backtrackingRecursiveFunction(args) {
```

- › Base case test for success: `return true`
  - › Base case test for failure: `return false`
  - › Loop over several options for “what to do next”:
    1. Tentatively “choose” one option
    2. if (“explore” with recursive call returns true) `return true`
    3. else That tentative idea didn’t work, so “un-choose” that option,  
*but don’t return false yet!--let the loop explore the other options before giving up!*
  - › None of the options we tried in the loop worked, so `return false`
- }

- › If at the exit, return true (*no false base case needed*)
- › Loop over N, W, E, S directions that are valid moves
  1. Choose: add that move to our path
  2. Recursively explore from there (maybe return true)
  3. Unchoose: remove that move from path
- › If no valid move reached end, return false

```

bool solveMazeHelper(Grid<bool>& maze, Stack<GridLocation>& path, GridLocation cur) {
    MazeGraphics::highlightPath(path, "blue");
    GridLocation exitLoc = {maze.numRows() - 1, maze.numCols() - 1};

    // Base case: we are at the exit
    if (cur == exitLoc) {
        MazeGraphics::highlightPath(path, "green");
        return true;
    }

    // Mark that we have visited this place so we don't retrace steps
    maze[cur] = false;

    // We get valid neighbors (as applicable) sorted as: N, S, E, W
    Set<GridLocation> validNeighbors = generateValidMoves();
    for (GridLocation cell : validNeighbors) {
        // Choose
        path.push(cell);
        // Explore
        if (solveMazeHelper(maze, path, cell)) {
            return true;
        }
        // Unchoose (undo)
        path.pop();
    }

    // Unmark
    maze[cur] = true;

    return false;
}

```

## Comparing our solution and the design template

**bool backtrackingRecursiveFunction(args) {**

    → Base case test for success: **return true**

    → Base case test for failure: **return false**

    → Loop over several options for “what to do next”:

        1. Tentatively “**choose**” one option

        2. if (“**explore**” with recursive call returns true) **return true**

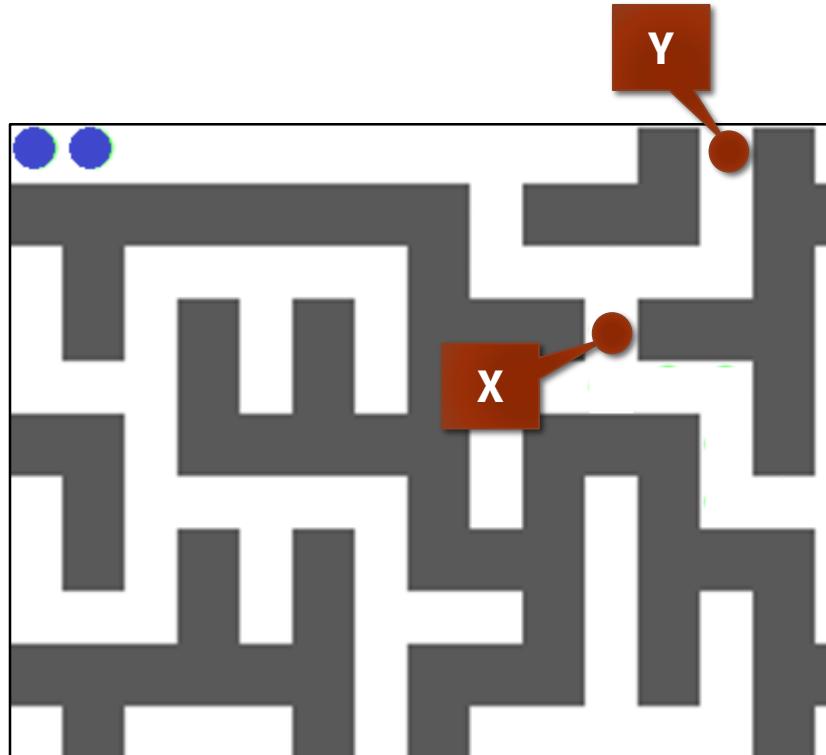
        3. else That tentative idea didn’t work, so “**un-choose**” that option,  
            but don’t return false yet!--let the loop explore the other options before giving up

    → None of the options we tried in the loop worked, so **return false**

## Your Turn: tracing the recursion in DFS maze-solver

Assume that the `generateValidMoves` function we use provides the valid moves (as applicable) sorted in this order: N, W, E, S.

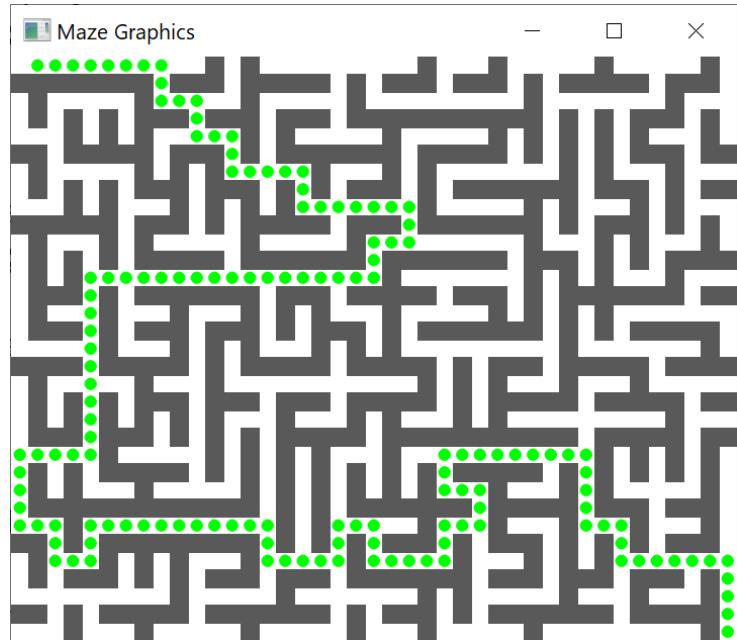
- In which order does the **DFS** visit the points marked X and Y?
  - A. visits X before Y
  - B. visits Y before X
  - C. doesn't visit both X and Y
- In which order does the **BFS** (like your homework) visit the points marked X and Y?
  - A. visits X before Y
  - B. visits Y before X
  - C. doesn't visit both X and Y



## Your Turn: tracing the recursion in DFS maze-solver

*Imagine the recursive call stack as we push/pop (call/return) in our recursive function as we solve this maze*

- What is the **most number of stack frames** on the stack at any point?
  - A. Equal to the number of cells in the maze
  - B. Equal to the number of “forks in the road” we encounter as we explore
  - C. Equal to the length of the path at its longest in our exploration
  - D. Equal to the length of the final solution path



# Depth-first vs. Breadth-first (DFS vs BFS)

- There's no one universal winner in terms of efficiency
  - › We can design a maze that is instantly solvable with BFS, but where DFS would take a very long time, and vice versa
  - › DFS heads off boldly in one direction
    - If that turns out to be right, very fast!
    - If it's wrong, may take a long time to course correct
- BFS has one key advantage: it is guaranteed to find the shortest path
  - › DFS just finds any working path (which can sometimes make it faster)

# Programming Abstractions

CS106B

Cynthia Bailey Lee  
Julie Zelenski

# Today's topics:

- Recursion Week Fortnight continues!
- Today:
  - › Wrap-up of Loops + recursion for *generating sequences and combinations*
  - › Loops + recursion for *recursive backtracking*

# Generating all possible coin flip die roll sequences



```
void generateAllSequences(int length, Vector<string>& allSequences)
{
    string sequence;
    generateAllSequences(length, allSequences, sequence);
}

void generateAllSequences(int length, Vector<string>& allSequences, string sequence)
{
    // base case: this sequence is full-length and ready to add
    if (sequence.size() == length) {
        allSequences.add(sequence);
        return;
    }
    // recursive cases: add 1-6 and continue
    for (int i = 1; i <= 6; i++) {
        sequence += integerToString(i);
        generateAllSequences(length, allSequences, sequence);
        sequence.erase(sequence.size() - 1);
    }
}
```



Much nicer!!

# Generating all possible coin flip die roll sequences



```
void generateAllSequences(int length, Vector<string>& allSequences)
{
    string sequence;
    generateAllSequences(length, allSequences, sequence);
}

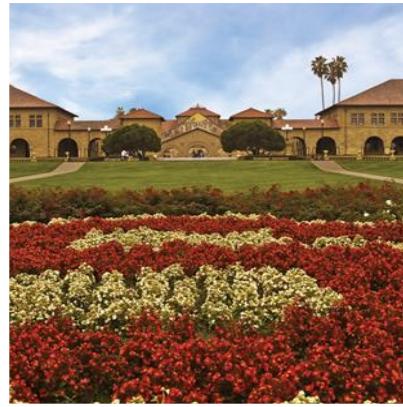
void generateAllSequences(int length, Vector<string>& allSequences, string sequence)
{
    // base case: this sequence is full-length and ready to add
    if (sequence.size() == length) {
        allSequences.add(sequence);
        return;
    }
    // recursive cases: add 1-6 and continue
    for (int i = 1; i <= 6; i++) {
        sequence += integerToString(i);
        generateAllSequences(length, allSequences, sequence);
        sequence.erase(sequence.size() - 1);
    }
}
```

A small icon of a silver gift box with a red ribbon and bow is positioned next to the recursive call in the first code block.

Notice that this loop **does not replace** the recursion. It just controls how many times the recursion launches.

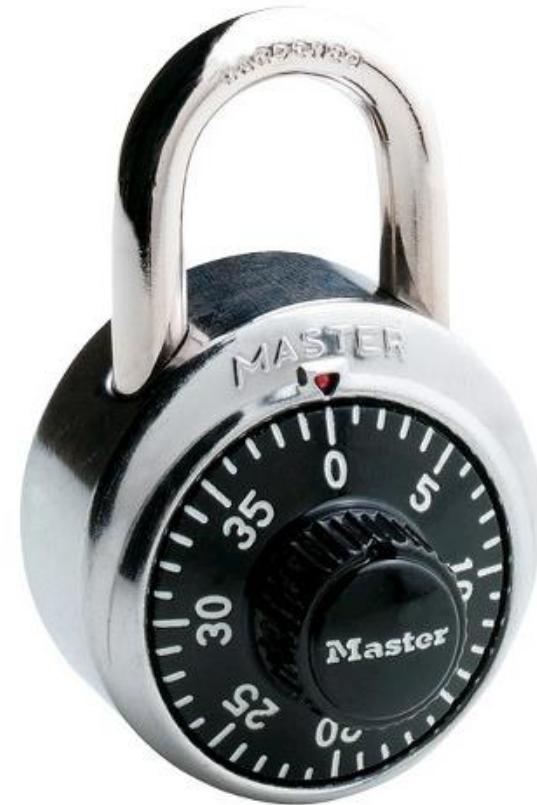
## Crack the combo lock!

TRYING TO FIND THE ONE  
SEQUENCE THAT WORKS



# Crack the combo lock!

- You forgot the combo to your locker 😞
- It consists of 3 numbers, in the range 1-39
  - › 1,1,1
  - › 39,39,39
  - › 2,3,4
  - › 2,32,17
  - › etc...
- We have no choice but to try all possible combos until we find one that unlocks the lock!
- When we find the successful combo, we save the combo in a `Vector<int>` of size 3, and return true. (*If we try all and it none works, the lock must be broken, return false.*)



# Trying all 1-39 combos sounds very similar to generating all 1-6 die roll sequences!



- We'll use the die-roll code as a starting point
- Which parts we will save, and which parts need a rewrite?

```
void generateAllSequences(int length, Vector<string>& allSequences, string sequence)
{
    // base case: this sequence is full-length and ready to add
    if (sequence.size() == length) {
        allSequences.add(sequence);
        return;
    }
    // recursive cases: add 1-6 and continue
    for (int i = 1; i <= 6; i++) {
        sequence += integerToString(i);
        generateAllSequences(length, allSequences, sequence);
        sequence.erase(sequence.size() - 1);
    }
}
```



# Trying all 1-39 combos sounds very similar to generating all 1-6 die roll sequences!



Return true/false,  
so make this bool.

die-roll code

When parts we will save, a

Don't need this  
parameter, our combo  
length is always 3.

and a rewrite?

Make this a pass-by-  
reference `Vector<int>`,  
so the caller gets the  
working combo.

```
void generateAllSequences(int length, Vector<string>& allSequences, string sequence)
{
    // base case: this sequence is full-length
    if (sequence.size() == length) {
        allSequences.add(sequence);
        return;
    }
    // recursive cases: add 1-6 and continue
    for (int i = 1; i <= 6; i++) {
        sequence += integerToString(i);
        generateAllSequences(length, allSequences, sequence);
        sequence.erase(sequence.size() - 1);
    }
}
```

Don't need this  
collection parameter,  
we are only looking for  
one working combo.



# Trying all 1-39 combos sounds very similar to generating all 1-6 die roll sequences!



We still want to detect when our combo is full-length (3), but it may not be the *right* full-length combo, so we need to check it.

```
bool findCombo(Vector<int>& combo)
{
    // base case: this sequence is full-length and ready to add
    if (sequence.size() == length) {
        allSequences.add(sequence);
        return;
    }
    // recursive cases: add 1-6 and continue
    for (int i = 1; i <= 6; i++) {
        sequence += integerToString(i);
        generateAllSequences(length, allSequences, sequence);
        sequence.erase(sequence.size() - 1);
    }
}
```



# Trying all 1-39 combos sounds very similar to generating all 1-6 die roll sequences!



We still want to detect when our combo is full-length (3), but it may not be the *right* full-length combo, so we need to check it.

```
bool findCombo(Vector<int>& combo)
{
    // base case: this sequence is full-length and ready to try on the lock!
    if (combo.size() == 3) {
        return tryCombo(combo);
    }
    // recursive cases: add 1-6 and continue
    for (int i = 1; i <= 6; i++) {
        sequence += integerToString(i);
        generateAllSequences(length, allSequences, sequence);
        sequence.erase(sequence.size() - 1);
    }
}
```



# Trying all 1-39 combos sounds very similar to generating all 1-6 die roll sequences!



```
bool findCombo(Vector<int>& combo)
{
    // base case: this sequence is f    length and ready
    if (combo.size() == 3) {
        return tryCombo(combo);
    }
    // recursive cases: add 1-6 and continue
    for (int i = 1; i <= 6; i++) {
        sequence += integerToString(i);
        generateAllSequences(length, allSequences, sequence);
        sequence.erase(sequence.size() - 1);
    }
}
```

We still want to loop over numbers (now 1-39).

We still want to choose a number, recursively continue generating the combo, and then “un-choose” that number before moving on to choose other numbers.

But we need to rewrite this for-loop body to take into account that a combo we try might or might not work.

# Generating all possible lock sequences, to find the one successful combo

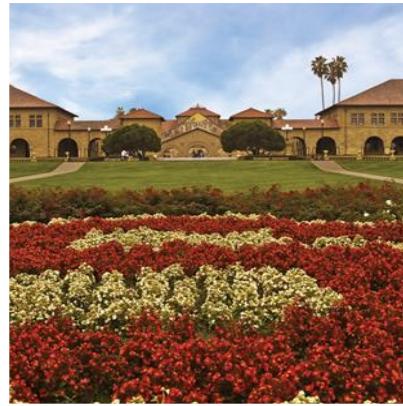


```
bool findCombo(Vector<int>& combo)
{
    // base case: this sequence is full-length and ready to try on the lock!
    if (combo.size() == 3) {
        return tryCombo(combo);
    }

    // recursive cases: add 1-39 and continue
    for (int i = 1; i <= 39; i++) {
        combo += i;
        if (findCombo(combo)) {
            return true;
        }
        combo.remove(combo.size() - 1);
    }
    return false;
}
```

# Choose + Recurse + Un-Choose

A COMMON RECURSIVE DESIGN  
PATTERN



# Generating all possible coin flip sequences



```
// Coin Flipper
// recursive cases: add H or T
sequence += "H";
generateAllSequences(length, allSequences, sequence);
sequence.erase(sequence.size() - 1);
sequence += "T";
generateAllSequences(length, allSequences, sequence);
}
```

1. Choose an option for the next step ("H")

2. Recursion to explore more steps of the sequence

3. Un-choose that option so we can try the other option ("T") for this current step

# A common design pattern in our solution: choose/unchoose



```
// Die Roll
// recursive cases add 1-6 and continue
for (int i = 1; i <= 6; i++) {
    sequence += integerToString(i);           2. Explore
    generateAllSequences(length, allSequences, sequence);
    sequence.erase(sequence.size() - 1);      3. Un-choose
}
```

# A common design pattern in our solution: choose/unchoose



```
// Combo Lock  
// recursive cases: add 1-39 and continue  
for (int i = 1; i <= 39; i++) {  
    combo += i;  
    if (findCombo(combo)) {  
        return true;  
    }  
    sequence.remove(sequence.size() - 1);  
}
```

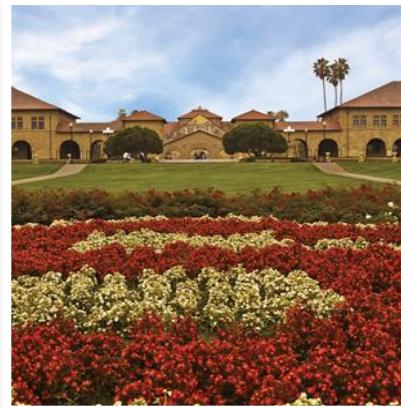
1. Choose

2. Explore

3. Un-choose

# “Backtracking” and Choose + Recurse + Un-Choose

A SPECIAL FLAVOR OF THE  
COMMON RECURSIVE DESIGN  
PATTERN



# Backtracking template

```
bool backtrackingRecursiveFunction(args) {
```

- › Base case test for success: `return true`
- › Base case test for failure: `return false`
- › Loop over several options for “what to do next”:
  1. Tentatively “**choose**” one option
  2. if (“**explore**” with recursive call returns true) `return true`
  3. else That tentative idea didn’t work, so “**un-choose**” that option,  
*but don’t return false yet!--let the loop explore the other options before giving up!*
- › None of the options we tried in the loop worked, so `return false`

```
}
```



# A common design pattern in our solution: Backtracking version of choose/unchoose



```
bool findCombo(Vector<int>& combo)
{
    // base case: this sequence is full-length and ready to try on the lock!
    if (combo.size() == 3) {
        return tryCombo(combo);
    }

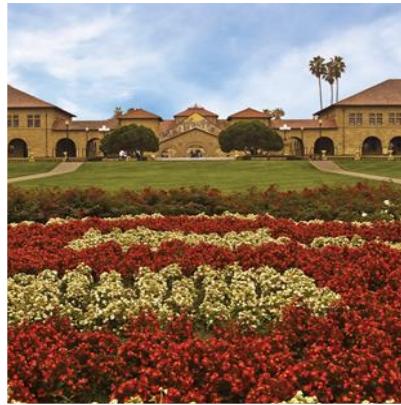
    // recursive cases: add 1-39 and
    for (int i = 1; i <= 39; i++) {
        combo += i;
        if (findCombo(combo)) {
            return true;
        }
        combo.remove(combo.size() - 1);
    }
    return false;
}
```

Annotations pointing to the code:

- Annotation pointing to the first `if` statement: "Base case test for success: `return true`"
- Annotation pointing to the first `if` statement: "Base case test for failure: `return false`"
- Annotation pointing to the `for` loop: "Loop over several options for ‘what to do next’:"
  1. Tentatively “choose” one option
  2. if (“explore” with recursive call returns true) `return true`
  3. else That tentative idea didn’t work, so “un-choose” that option,  
*but don’t return false yet!—let the loop explore the other options before giving up.*
- Annotation pointing to the `return false;` statement: "None of the options we tried in the loop worked, so `return false`"

# Revisiting Big-O

SOME PRACTICAL TIPS



# Big-O Quick Tips

- To examine program runtime, assume:
  - › Single statement = 1
  - › Function call = (sum of statements in function)
  - › A loop of N iterations =  $(N * (\text{body's runtime}))$

# Your Turn: What is the Big-O runtime cost for this function?

```
void myFunction(int N) {  
    statement1;                                // runtime = 1  
  
    for (int i = 1; i <= N; i++) {                // runtime = N^2  
        for (int j = 1; j <= N; j++) {            //     runtime = N  
            statement2;                      //         runtime = 1  
            statement3;                      //         runtime = 1  
        }  
    }  
  
    for (int i = 1; i <= N; i++) {                // runtime = 3N  
        statement4;                      //         runtime = 1  
        statement5;                      //         runtime = 1  
        statement6;                      //         runtime = 1  
    }  
}
```

# Your Turn: What is the Big-O runtime cost for this function?

```
void myFunction(int N) {  
    statement1;                                // runtime = 1  
  
    for (int i = 1; i <= N; i++) {                // runtime = N^2  
        for (int j = 1; j <= N; j++) {            //     runtime = N  
            statement2;                      //         runtime = 1  
            statement3;                      //         runtime = 1  
        }  
    }  
  
    for (int i = 1; i <= N; i++) {                // runtime = 3N  
        statement4;                      //     runtime = 1  
        statement5;                      //     runtime = 1  
        statement6;                      //     runtime = 1  
    }                                              // total = 2N^2 + 3N + 1  
}                                              // total = O(N^2)
```

# Programming Abstractions

## CS106B

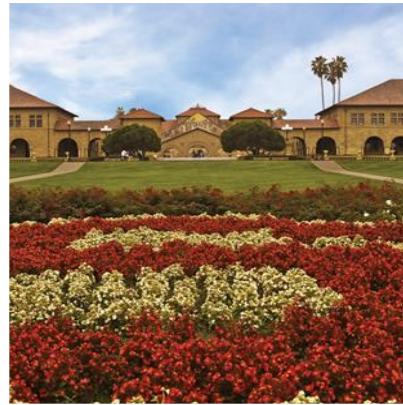
Cynthia Bailey Lee  
Julie Zelenski

# Today's topics:

- Recursion Week Fortnight continues!
- Today:
  - › Loops + recursion for *generating sequences and combinations*
- Upcoming:
  - › Loops + recursion for *recursive backtracking*
- **Assignment 3 “YEAH” Session TONIGHT**
  - › Monday Oct 11, 6:30pm in Durand 410
  - › YEAH = Your Early Assignment Help
  - › Orientation to what the assignment is asking, tips for getting started, common questions, etc.

# Heads or Tails?

GENERATING SEQUENCES



# Heads or Tails?

- You flip a coin 5 times
- What are all the possible heads/tails sequences you could observe?
  - › TTTTT
  - › HHHHH
  - › THTHT
  - › HHHHT
  - › etc...
- We want to write a program to fill a Vector with strings representing each of the possible sequences.



# Generating all possible coin flip sequences



```
void generateAllSequences(int length, Vector<string>& allSequences)
{
    string sequence;
    generateAllSequences(length, allSequences, sequence);
}

void generateAllSequences(int length, Vector<string>& allSequences, string sequence)
{
    // base case: this sequence is full-length and ready to add
    if (sequence.size() == length) {
        allSequences.add(sequence);
        return;
    }
    // recursive cases: add H or T and continue
    sequence += "H";
    generateAllSequences(length, allSequences, sequence);
    sequence.erase(sequence.size() - 1);
    sequence += "T";
    generateAllSequences(length, allSequences, sequence);
}
```

# Your Turn: coin flip sequences



```
void generateAllSequences(int length, Vector<string>& allSequences, string sequence)
{
    // base case: this sequence is full-length and ready to add
    if (sequence.size() == length) {
        allSequences.add(sequence);
        return;
    }
    // recursive cases: add H or T and continue
    sequence += "H";
    generateAllSequences(length, allSequences, sequence);
    sequence.erase(sequence.size() - 1);
    sequence += "T";
    generateAllSequences(length, allSequences, sequence);
}
```

- Q: Of these sequences (all of which should be included in `allSequences`), which sequence appears first in `allSequences`? Last?
  - TTTTT, HHHHH, THTHT, HHHHT

# Your Turn: coin flip sequences

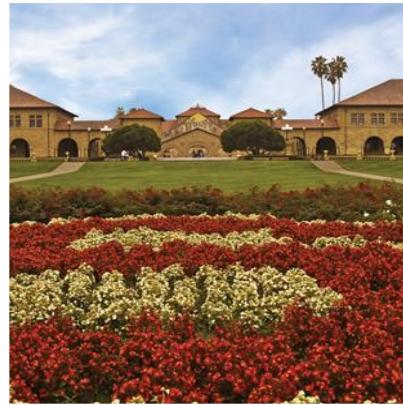


```
void generateAllSequences(int length, Vector<string>& allSequences, string sequence)
{
    // base case: this sequence is full-length and ready to add
    if (sequence.size() == length) {
        allSequences.add(sequence);
        return;
    }
    // recursive cases: add H or T and continue
    sequence += "H";
    generateAllSequences(length, allSequences, sequence);
    sequence.erase(sequence.size() - 1);
    sequence += "T";
    generateAllSequences(length, allSequences, sequence);
}
```

- **Q: What would happen if we didn't do the erase (highlighted above)? Which of the following sequences would we NOT generate? Which additional sequences would we generate (that we shouldn't)?**
  - TTTTT, HHHHH, THTHT, HHHHT

# Roll the Dice!

GENERATING MORE  
SEQUENCES



# Roll the Dice!

- You roll a single die 5 times
- What are all the possible 1/2/3/4/5/6 sequences you could observe?
  - › 11111
  - › 66666
  - › 12345
  - › 21655
  - › etc...
- We want to write a program to fill a Vector with strings representing each of the possible sequences.



# Generating all possible coin flip die roll sequences



```
void generateAllSequences(int length, Vector<string>& allSequences)
{
    string sequence;
    generateAllSequences(length, allSequences, sequence);
}

void generateAllSequences(int length, Vector<string>& allSequences, string sequence)
{
    // base case: this sequence is full-length and ready to add
    if (sequence.size() == length) {
        allSequences.add(sequence);
        return;
    }
    // recursive cases: add H or T and continue
    sequence += "H";
    generateAllSequences(length, allSequences, sequence);
    sequence.erase(sequence.size() - 1);
    sequence += "T";
    generateAllSequences(length, allSequences, sequence);
}
```



To adapt for die rolls,  
we need to change this  
from 2 options (H/T) to  
6 options (1-6).

# Generating all possible ~~coin flip~~ die roll sequences

```
// recursive cases: add 1 or 2 or 3 or 4 or 5 or 6 and continue  
sequence += "1";  
generateAllSequences(length, allSequences, sequence);  
sequence.erase(sequence.size() - 1);  
sequence += "2";  
generateAllSequences(length, allSequences, sequence);  
sequence.erase(sequence.size() - 1);  
sequence += "3";  
generateAllSequences(length, allSequences, sequence);  
sequence.erase(sequence.size() - 1);  
sequence += "4";  
generateAllSequences(length, allSequences, sequence);  
sequence.erase(sequence.size() - 1);  
sequence += "5";  
generateAllSequences(length, allSequences, sequence);  
sequence.erase(sequence.size() - 1);  
sequence += "6";  
generateAllSequences(length, allSequences, sequence);  
}
```



This works, but YIKES!!  
So much copy-paste!!



# Generating all possible coin flip die roll sequences



```
// recursive cases: add 1 or 2 or 3 or 4 or 5 or 6 and continue  
sequence += "1";  
generateAllSequences(length, allSequences, sequence);  
sequence.erase(sequence.size() - 1);  
sequence += "2";  
generateAllSequences(length, allSequences, sequence);  
sequence.erase(sequence.size() - 1);  
sequence += "3";  
generateAllSequences(length, allSequences, sequence);  
sequence.erase(sequence.size() - 1);  
sequence += "4";  
generateAllSequences(length, allSequences, sequence);  
sequence.erase(sequence.size() - 1);  
sequence += "5";  
generateAllSequences(length, allSequences, sequence);  
sequence.erase(sequence.size() - 1);  
sequence += "6";  
generateAllSequences(length, allSequences, sequence);  
}  
}
```

Let's take the repeated actions and put them in a for-loop from 1 to 6.

# Generating all possible coin flip die roll sequences



```
void generateAllSequences(int length, Vector<string>& allSequences)
{
    string sequence;
    generateAllSequences(length, allSequences, sequence);
}

void generateAllSequences(int length, Vector<string>& allSequences, string sequence)
{
    // base case: this sequence is full-length and ready to add
    if (sequence.size() == length) {
        allSequences.add(sequence);
        return;
    }
    // recursive cases: add 1-6 and continue
    for (int i = 1; i <= 6; i++) {
        sequence += integerToString(i);
        generateAllSequences(length, allSequences, sequence);
        sequence.erase(sequence.size() - 1);
    }
}
```



Much nicer!!

# Generating all possible coin flip die roll sequences



```
void generateAllSequences(int length, Vector<string>& allSequences)
{
    string sequence;
    generateAllSequences(length, allSequences, sequence);
}

void generateAllSequences(int length, Vector<string>& allSequences, string sequence)
{
    // base case: this sequence is full-length and ready to add
    if (sequence.size() == length) {
        allSequences.add(sequence);
        return;
    }
    // recursive cases: add 1-6 and continue
    for (int i = 1; i <= 6; i++) {
        sequence += integerToString(i);
        generateAllSequences(length, allSequences, sequence);
        sequence.erase(sequence.size() - 1);
    }
}
```

A small icon of a silver gift box with a red ribbon and bow is positioned next to the recursive call in the first code block.

Notice that this loop **does not replace** the recursion. It just controls how many times the recursion launches.

# Programming Abstractions

CS106B

Cynthia Bailey Lee  
Julie Zelenski

# Today's Topics

Recursion Week continues!

- Today, two applications of recursion:
  - › Binary Search (one of the fundamental algorithms of CS)
    - We saw the idea of this on Wed, but today we'll code it up
  - › Fractals (will help us visualize the order of operations in recursion)

Next time:

- More recursion! It's Recursion Week!
- Like Shark Week, but more nerdy

# Binary Search Refresher

(RECALL FROM WEDNESDAY'S  
LECTURE)



## Binary search (*refresher*)

The question we're trying to answer is, given a list of numbers, does this list contain some particular value, or not? For convenience, we have kept our list sorted.

0	1	2	3	4	5	6	7	8	9	10
2	7	8	13	25	29	33	51	89	90	95

If instead we **jump right to the middle**, one of three things can happen:

1. The middle one happens to be the number we were looking for, yay!
2. We realize we went too far
3. We realize we didn't go far enough

## Binary search (*refresher*)

The question we're trying to answer is, given a list of numbers, does this list contain some particular value, or not? For convenience, we have kept our list sorted.

0	1	2	3	4	5	6	7	8	9	10
2	7	8	13	25	29	33	51	89	90	95

Let's say the answer was case 3, "we didn't go far enough"

- We ruled out the entire first half, and now only have the second half to search
- We could start at the front of the second half and proceed forward checking each item one at a time... **but why do that when we know we have a better way?**

**Jump right to the middle** of the region to search

## Binary search (*refresher*)

The question we're trying to answer is, given a list of numbers, does this list contain some particular value, or not? For convenience, we have kept our list sorted.

0	1	2	3	4	5	6	7	8	9	10
2	7	8	13	25	29	33	51	89	90	95

Let's say the answer was 51. If we said "no", we didn't go far enough"

- We ruled out the first half because we now only have the second half.
- We can repeat the process on the second half and proceed forward, jumping in half at a time... **but why do that when we know we have a better way?**

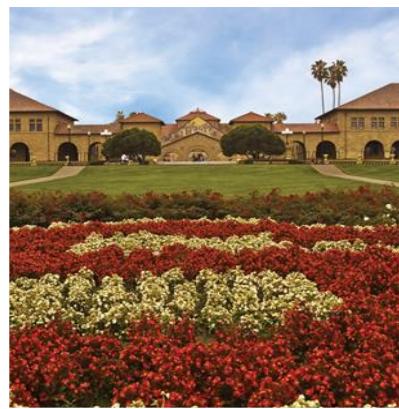
**RECURSION!!**

**Jump right to the middle** of the region to search

# Binary Search Implementation

NOW WE UNDERSTAND THE  
APPROACH.

WHAT DOES THE CODE LOOK  
LIKE?



```
bool binarySearch(Vector<int>& data, int key) {  
    // want to keep passing same data by reference for efficiency,  
    // but then how do we cut in half?  
    return binarySearch(data, key, 0, data.size() - 1); // 2 new params  
}
```

```
bool binarySearch(Vector<int>& data, int key, int start, int end) {
```

# Recursive Function Design Tip: Wrapper function

- When we want to write a recursive function that needs more book-keeping data passed around than an outsider user would want to worry about, do this:
  1. Write the function as you need to for correctness, using any extra book-keeping parameters you like, in whatever way you like.
  2. Make a second function that the outside world sees, using only the minimum number of parameters, and have it do nothing but call the recursive one.
    - Called a “wrapper” function because it’s like pretty outer packaging.



```
bool binarySearch(Vector<int>& data, int key) {  
    // want to keep passing same data by reference for efficiency,  
    // but then how do we cut in half?  
    return binarySearch(data, key, 0, data.size() - 1); // 2 new params  
}
```



```
bool binarySearch(Vector<int>& data, int key, int start, int end) {  
    if (start > end) {  
        return false;  
    }  
    int mid = (start + end) / 2;  
    if (key == data[mid]) {  
        return true;  
    } else if (key < data[mid]) {  
        return binarySearch(data, key, _____, _____);  
    } else {  
        return binarySearch(data, key, _____, _____);  
    }  
}
```

```
bool binarySearch(Vector<int>& data, int key) {  
    // want to keep passing same data by reference for efficiency,  
    // but then how do we cut in half?  
    return binarySearch(data, key, 0, data.size() - 1); // 2 new params  
}
```



```
bool binarySearch(Vector<int>& data, int key, int start, int end) {  
    if (start > end) {  
        return false;  
    }  
    int mid = (start + end) / 2;  
    if (key == data[mid]) {  
        return true;  
    } else if (key < data[mid]) {  
        return binarySearch(data, key, _____, _____);  
    } else {  
        return binarySearch(data, key, _____, _____);  
    }  
}
```

### Your Turn:

What goes on the blanks below, to divide the remaining searchable region of our vector in half?

```
bool binarySearch(const Vector<int>& data, int key) {  
    // want to keep passing same data by reference for efficiency,  
    // but then how do we cut in half?  
    return binarySearch(data, key, 0, data.size() - 1); // 2 new params  
}
```



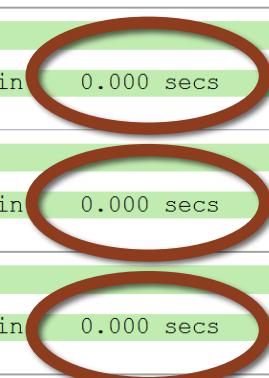
```
bool binarySearch(const Vector<int>& data, int key, int start, int end) {  
    if (start > end) {  
        return false;  
    }  
    int mid = (start + end) / 2;  
    if (key == data[mid]) {  
        return true;  
    } else if (key < data[mid]) {  
        return binarySearch(data, key, start, mid - 1);  
    } else {  
        return binarySearch(data, key, mid + 1, end);  
    }  
}
```

# Binary Search performance

```
SimpleTest BinarySearch
Tests from PROVIDED_TEST

Correct (PROVIDED_TEST, binsearch.cpp:88) Basic correctness: found value
Correct (PROVIDED_TEST, binsearch.cpp:93) Basic correctness: missing value
Correct (PROVIDED_TEST, binsearch.cpp:98) Edge case: found first value
Correct (PROVIDED_TEST, binsearch.cpp:103) Edge case: found last value
Correct (PROVIDED_TEST, binsearch.cpp:108) Timing on 10K elements
    Line 112 TIME_OPERATION binarySearch(data, 5) (size = 10000) completed in 0.000 secs
Correct (PROVIDED_TEST, binsearch.cpp:115) Timing on 100K elements
    Line 119 TIME_OPERATION binarySearch(data, 5) (size = 100000) completed in 0.000 secs
Correct (PROVIDED_TEST, binsearch.cpp:122) Timing on 1M elements
    Line 126 TIME_OPERATION binarySearch(data, 5) (size = 1000000) completed in 0.000 secs
Passed 7 of 7 tests. Great!
```

Q. We saw the test take a long time to run for 1M, but it reports 0.000 secs. What's going on??



# Binary Search performance

SimpleTest BinarySearch

## Tests from PROVIDED\_TEST

Correct (PROVIDED\_TEST, binsearch.cpp:88) Basic correctness: found value

Correct (PROVIDED\_TEST, binsearch.cpp:93) Basic correctness: missing value

Correct (PROVIDED\_TEST, binsearch.cpp:98) Edge case: found first value

Correct (PROVIDED\_TEST, binsearch.cpp:103) Edge case: found last value

Correct (PROVIDED\_TEST, binsearch.cpp:108) Timing on 10K elements

```
Line 112 TIME_OPERATION binarySearch(data, 5) (size = 10000) completed in 0.000 secs
```

Correct (PROVIDED\_TEST, binsearch.cpp:115) Timing on 100K elements

```
Line 119 TIME_OPERATION binarySe
```

Correct (PROVIDED\_TEST, binsearch.cpp:122)

```
Line 126 TIME_OPERATION binarySe
```

Passed 7 of 7 tests. Great!

Q. We saw the test take a long time to run for 1M, but it reports 0.000 secs. What's going on??

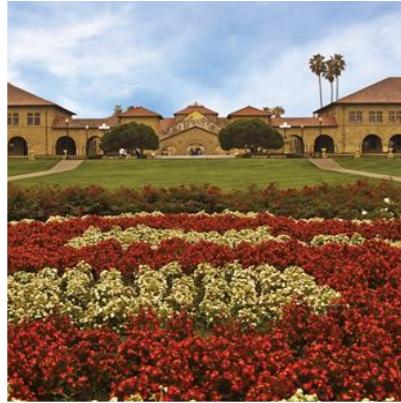
Answer:

$$\log_2(10K) \approx 13$$
$$\log_2(100K) \approx 16$$
$$\log_2(1M) \approx 20$$

...on a computer that does billions of operations per second!

# Fractals

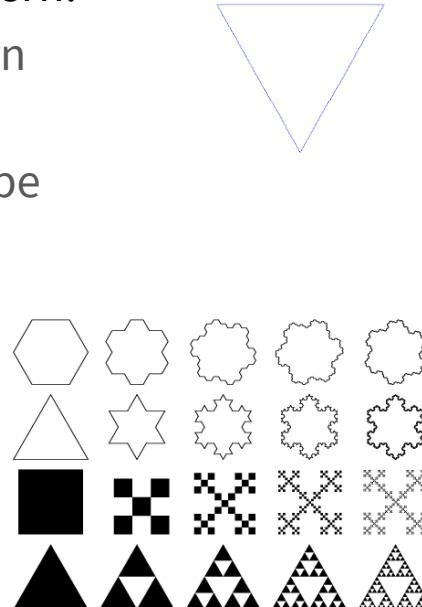
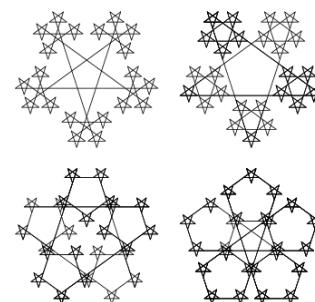
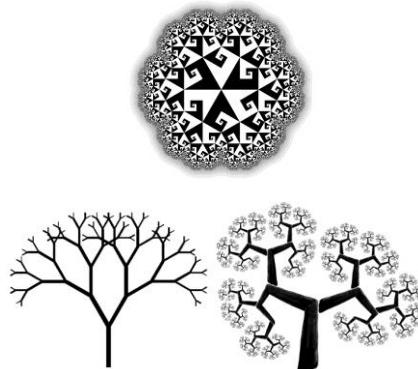
PRETTY!



# Fractals

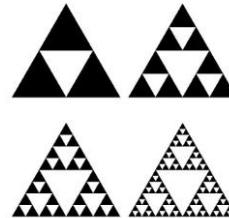
fractal: A self-similar mathematical set that can often be drawn as a recurring graphical pattern.

- Smaller instances of the same shape or pattern occur within the pattern itself.
- When displayed on a computer screen, it can be possible to infinitely zoom in/out of a fractal.

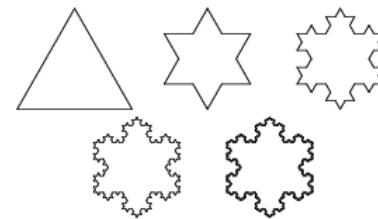


## Example fractals

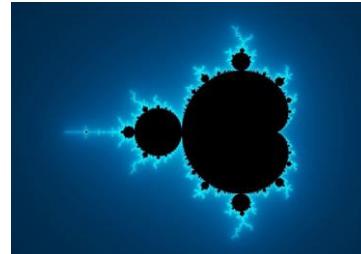
Sierpinski triangle: equilateral triangle contains smaller triangles inside it



Koch snowflake: a triangle with smaller triangles poking out of its sides



Mandelbrot set: circle with smaller circles on its edge



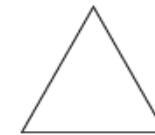
# Coding a fractal

Many fractals are implemented as a function that accepts **x/y coordinates, size,** and a **level** parameter.

- The **level** is the number of recurrences of the pattern to draw.

Example, Koch snowflake:

- `snowflake(window, x, y, size, 1);`



- `snowflake(window, x, y, size, 2);`



- `snowflake(window, x, y, size, 3);`



# Cantor Set

The Cantor Set is a simple fractal that begins with a line segment.

- At each level, the middle third of the segment is removed.
- In the next level, the middle third of each third is removed.



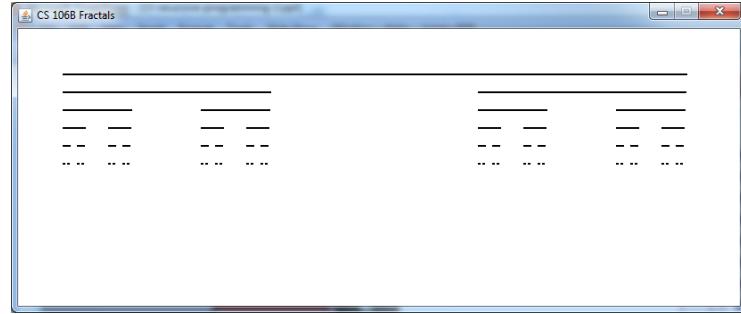
Write a function `cantorSet` that draws a Cantor Set with a given number of levels (lines) at a given position/size.

- Place 20 px of vertical space between levels.

# Cantor Set solution

```
void cantorSet(GWindow& window, int x, int y, int length, int levels)
{
    if (levels > 0) {
        // draw our own line
        drawThickLine(window, x, y, length, levels);

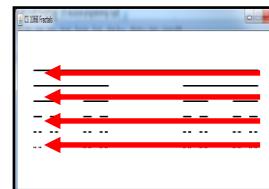
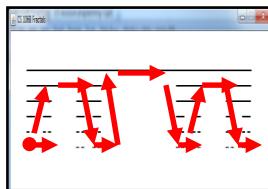
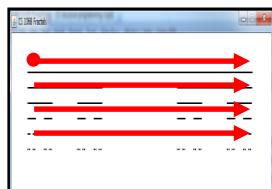
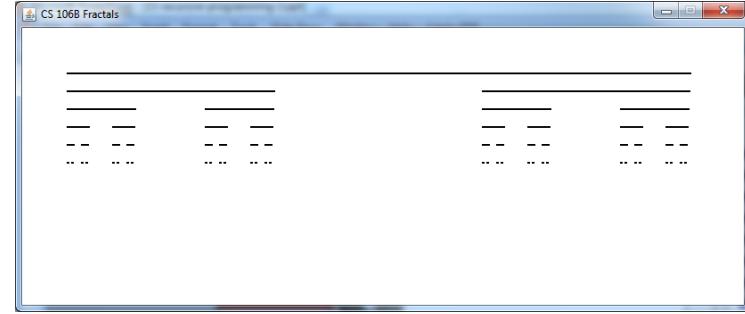
        // recursively draw next lines
        int newY = y + LINE_SPACING;
        int newLength = length / 3;
        int newLevels = levels - 1;
        // left third
        cantorSet(window, x, newY, newLength, newLevels);
        // right third
        cantorSet(window, x + (2 * length / 3), newY, newLength, newLevels);
    }
}
```



# Your Turn: In what order does the recursion draw the lines?

```
void cantorSet(GWindow& window, int x, int y, int length, int levels)
{
    if (levels > 0) {
        // draw our own line
        drawThickLine(window, x, y, length, levels);

        // recursively draw next lines
        int newY = y + LINE_SPACING;
        int newLength = length / 3;
        int newLevels = levels - 1;
        // left third
        cantorSet(window, x, newY, newLength, newLevels);
        // right third
        cantorSet(window, x + (2 * length / 3), newY, newLength, newLevels);
    }
}
```



(A)

(B)

(C)

(D)

(E) other



## Your Turn: die roll sequences

```
void generateAllSequences(int length, Vector<string>& allSequences, string sequence)
{
    // base case: this sequence is full-length and ready to add
    if (sequence.size() == length) {
        allSequences.add(sequence);
        return;
    }
    // recursive cases: add 1-6 and continue
    for (int i = 1; i <= 6; i++) {
        sequence += integerToString(i);
        generateAllSequences(length, allSequences, sequence);
        sequence.erase(sequence.size() - 1);
    }
}
```

- Q: Of these sequences (all of which should be included in `allSequences`), which sequence appears first in `allSequences`? Last?
  - 11111, 66666, 12345, 21655

# Programming Abstractions

## CS106B

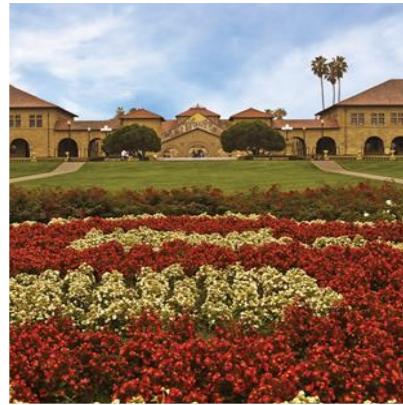
Cynthia Bailey Lee  
Julie Zelenski

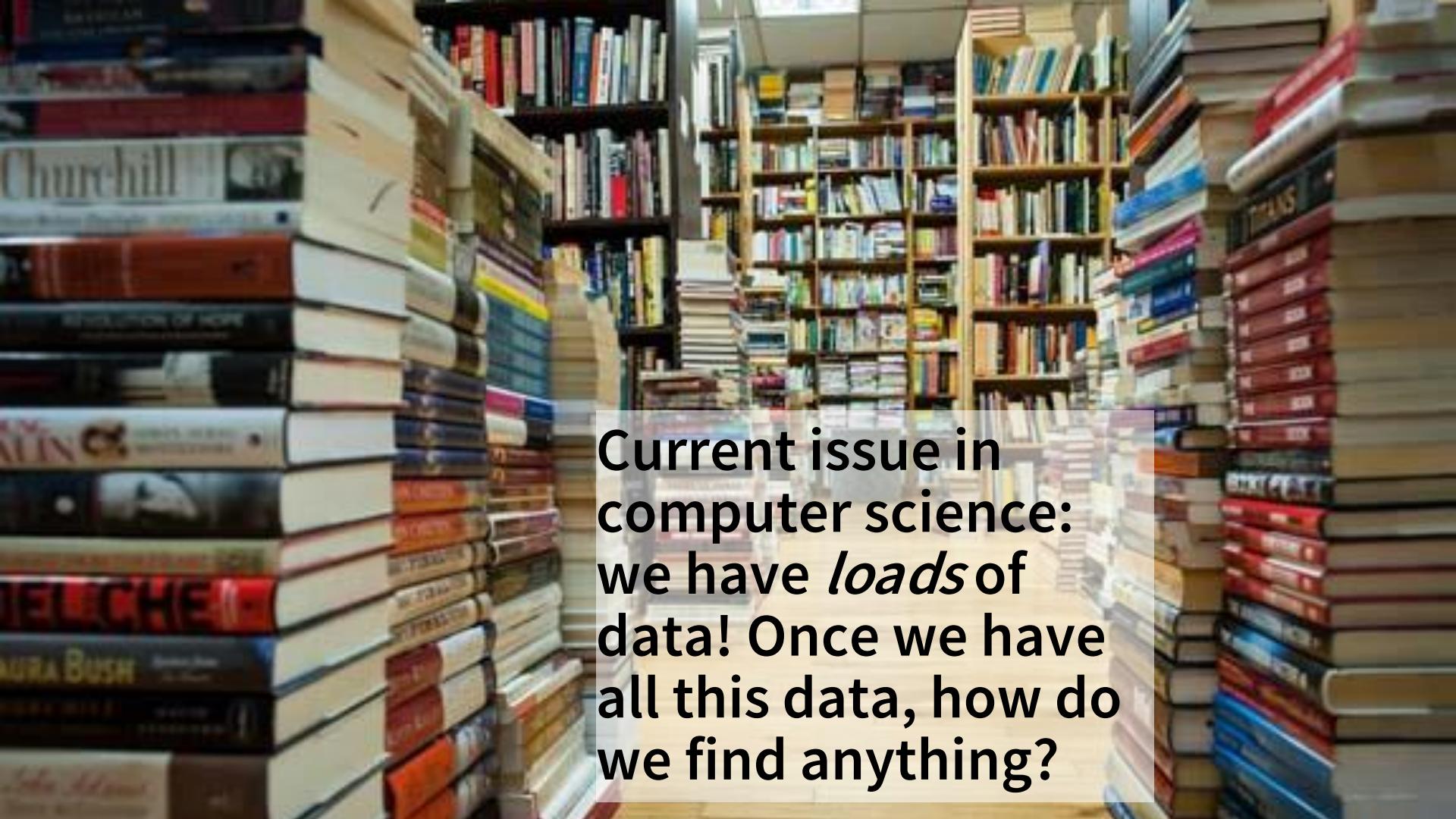
# Today's Topics:

- Contrasting performance of 3 recursive algorithms
- Quantifying algorithm performance with Big-O analysis
- Getting a sense of scale in Big-O analysis

# Binary Search

AN ELEGANT SOLUTION TO  
THE PROBLEM OF TOO MUCH  
DATA





Current issue in  
computer science:  
we have *loads* of  
data! Once we have  
all this data, how do  
we find anything?

# Does this list of numbers contain X?

The question we're trying to answer is, given a list of numbers, does this list contain some particular value, or not? For convenience, we have kept our list sorted.

How long does it take us to find a number we are looking for?

0	1	2	3	4	5	6	7	8	9	10
2	7	8	13	25	29	33	51	89	90	95

# Does this list of numbers contain X?

The question we're trying to answer is, given a list of numbers, does this list contain some particular value, or not? For convenience, we have kept our list sorted.

How long does it take us to find a number we are looking for?

<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>	<b>8</b>	<b>9</b>	<b>10</b>
2	7	8	13	25	29	33	51	89	90	95

If you start at the front and proceed forward, each item you examine rules out 1 item

# Does this list of numbers contain X?

The question we're trying to answer is, given a list of numbers, does this list contain some particular value, or not? For convenience, we have kept our list sorted.

0	1	2	3	4	5	6	7	8	9	10
2	7	8	13	25	29	33	51	89	90	95

If instead we **jump right to the middle**, one of three things can happen:

1. The middle one happens to be the number we were looking for, yay!
2. We realize we went too far
3. We realize we didn't go far enough

# Does this list of numbers contain X?

The question we're trying to answer is, given a list of numbers, does this list contain some particular value, or not? For convenience, we have kept our list sorted.

0	1	2	3	4	5	6	7	8	9	10
2	7	8	13	25	29	33	51	89	90	95

If instead we **jump right to the middle**, one of three things can happen:

1. The middle one happens to be the number we were looking for, yay!
2. We realize we went too far
3. We realize we didn't go far enough

**Ruling out HALF the options in one step is so much faster than only ruling out one!**

## Binary search

0	1	2	3	4	5	6	7	8	9	10
2	7	8	13	25	29	33	51	89	90	95

Let's say the answer was case 3, "we didn't go far enough"

- We ruled out the entire first half, and now only have the second half to search
- We could start at the front of the second half and proceed forward checking each item one at a time...

## Binary search

0	1	2	3	4	5	6	7	8	9	10
2	7	8	13	25	29	33	51	89	90	95

Let's say the answer was case 3, "we didn't go far enough"

- We ruled out the entire first half, and now only have the second half to search
- We could start at the front of the second half and proceed forward checking each item one at a time... **but why do that when we know we have a better way?**

**Jump right to the middle** of the region to search

# Binary search

0	1	2	3	4	5	6	7	8	9	10
2	7	8	13	25	29	33	51	89	90	95

Let's say the answer was case 3, "we didn't go far enough"

- We ruled out the entire first half, and now only have the second half to search
- We could search the second half and proceed forward... but why do that when we can do it **recursively**?

**Jump right** to the middle of the region to search

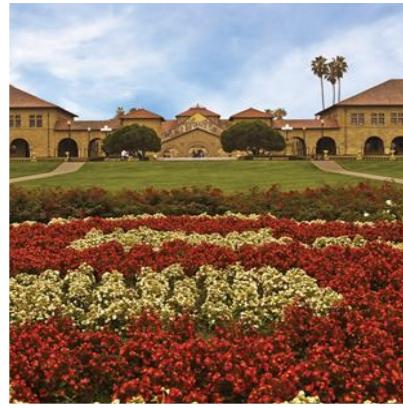
# Binary Search pseudocode

- We'll write the real C++ code together on Friday, but here's the outline/pseudocode of how it works:

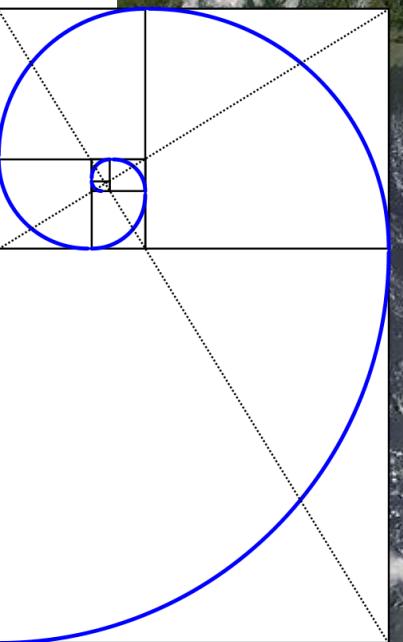
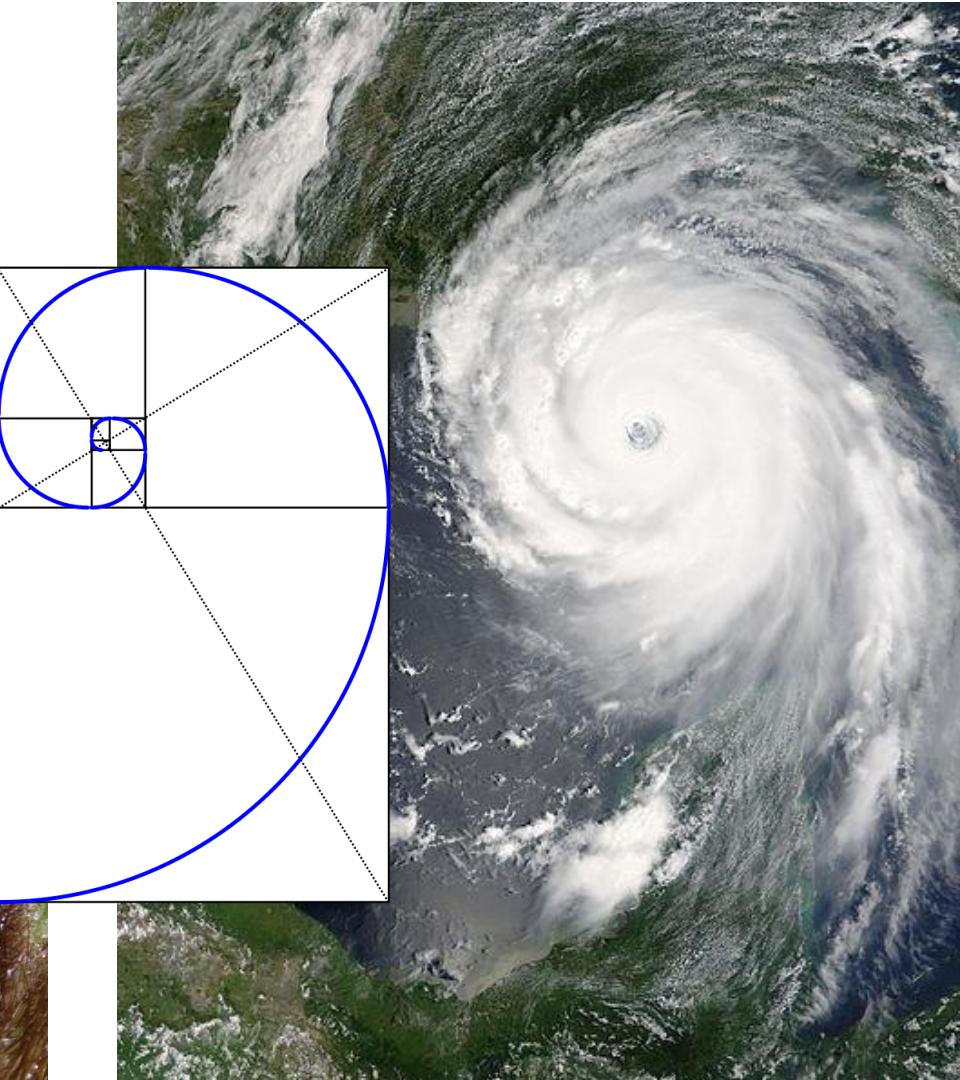
```
bool binarySearch(Vector<int>& data, int key)
{
    if (data.size() == 0) {
        return false;
    }
    if (key == data[midpoint]) {
        return true;
    } else if (key < data[midpoint]) {
        return binarySearch(data[first half only], key);
    } else {
        return binarySearch(data[second half only], key);
    }
}
```

# The Fibonacci Sequence

\*MATH NERD REJOICING  
INTENSIFIES\*



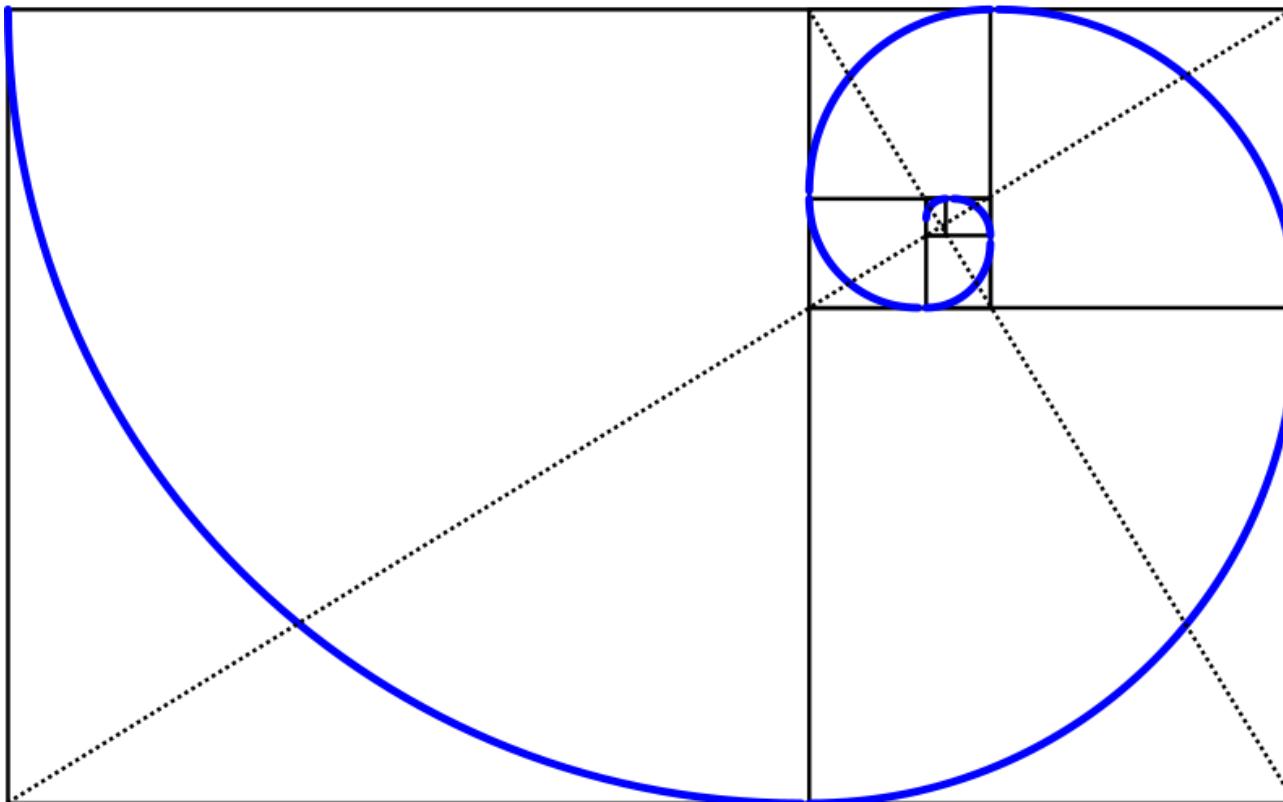
# Fibonacci in nature



These files are, respectively: public domain (hurricane) and licensed under the [Creative Commons Attribution 2.0 Generic license](#) (fibonacci and fern).

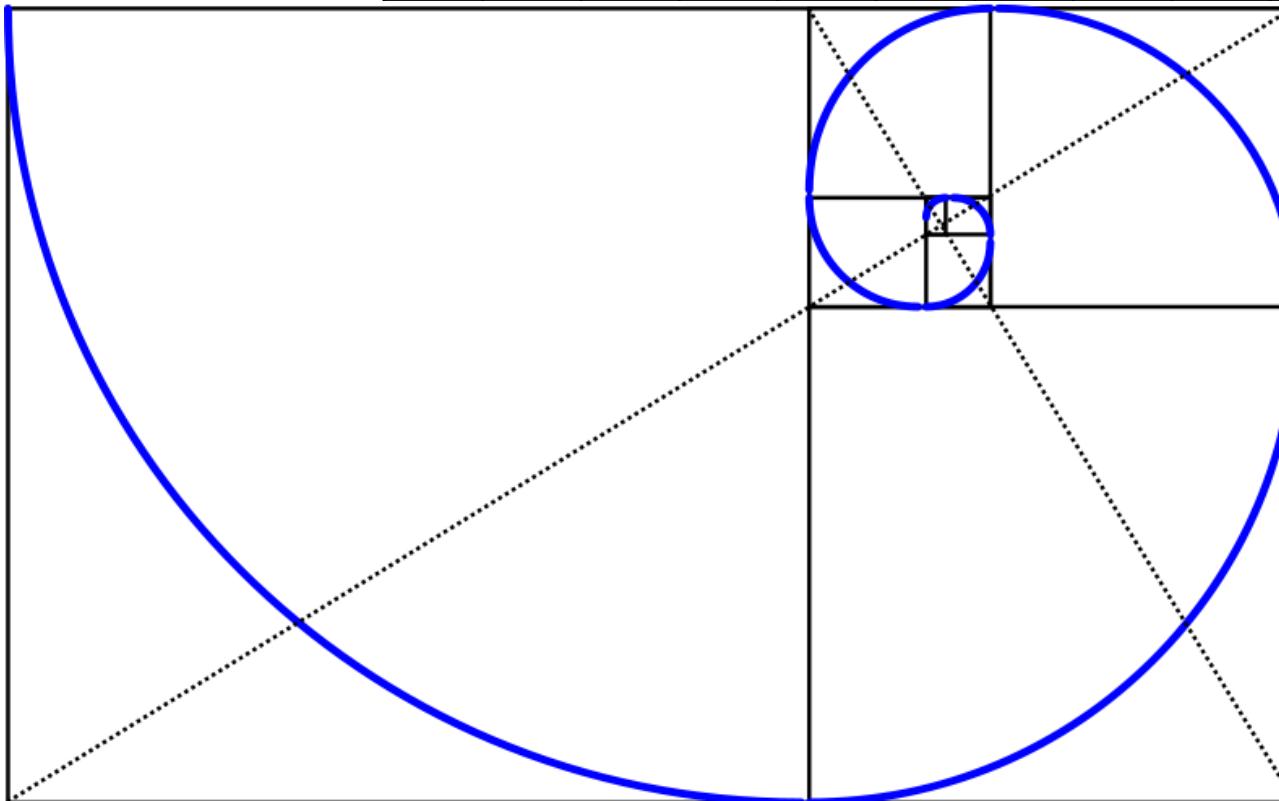
# Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89,



# Fibonacci

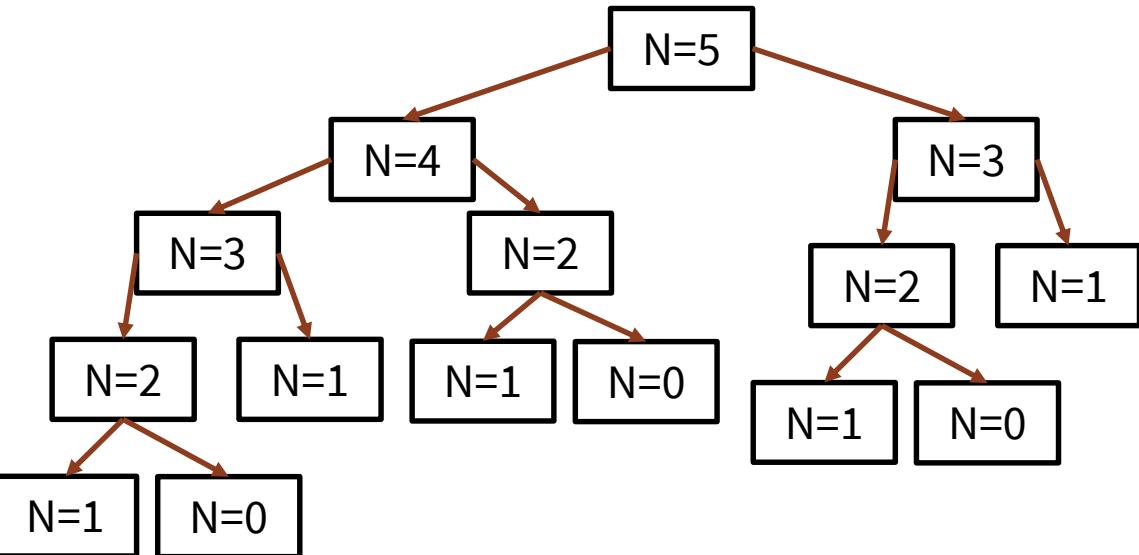
0	1	2	3	4	5	6	7	8	9	10	11
0	1	1	2	3	5	8	13	21	34	55	89



This image is licensed under the [Creative Commons Attribution-Share Alike 3.0 Unported](https://creativecommons.org/licenses/by-sa/3.0/) license. [http://commons.wikimedia.org/wiki/File:Golden\\_spiral\\_in\\_rectangles.png](http://commons.wikimedia.org/wiki/File:Golden_spiral_in_rectangles.png)

# Fibonacci

```
int fib(int n)
{
    if (n == 0) {
        return 0;
    } else if (n == 1)
        return 1;
    } else {
        return fib(n - 1) + fib(n - 2);
    }
}
```

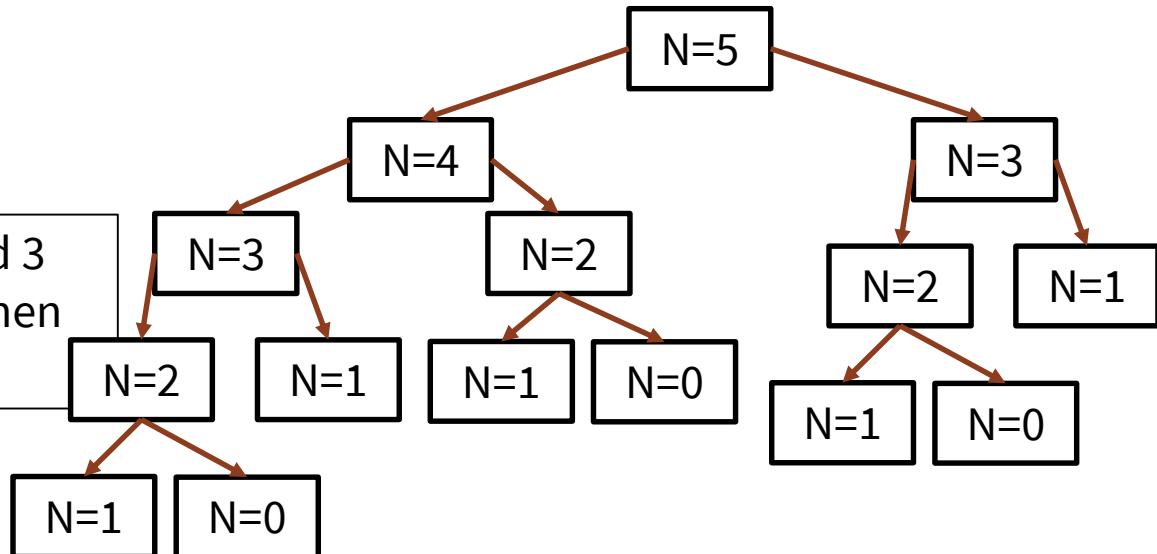


Work is duplicated throughout the call tree

- fib(2) is calculated 3 separate times when calculating fib(5)!
- 15 function calls in total for fib(5)!

## Fibonacci

fib(2) is calculated 3 separate times when calculating fib(5)!



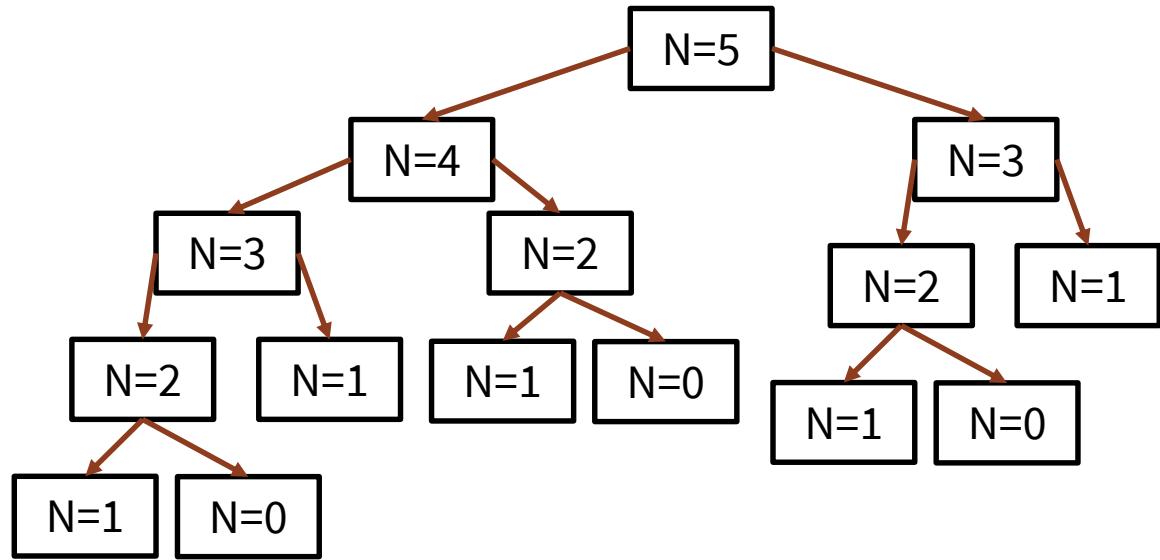
How many times would we calculate  $\text{fib}(2)$  while calculating  $\text{fib}(6)$ ?

***See if you can just “read” it off the chart above.***

- A. 4 times
- B. 5 times
- C. 6 times
- D. Other/none/more

# Fibonacci

N	fib(N)	# of calls to fib(2)
2	1	1
3	2	1
4	3	2
5	5	3
6	8	
7	13	
8	21	
9	34	
10	55	



# Efficiency of naïve Fibonacci implementation

When we **added 1** to the input N, the number of times we had to calculate fib(2) **nearly doubled** ( $\sim 1.6^*$  times)

- Ouch!

\* This number is called the “Golden Ratio” in math—cool!

**Goal: predict how much time it will take to compute for arbitrary input N.**

Calculation: “approximately”  $(1.6)^N$

# Big-O Performance Analysis

A WAY TO COMPARE THE  
NUMBER OF STEPS TO RUN  
THESE FUNCTIONS



# Big-O analysis in computer science

S Vector × + ▾

← → ⌂ web.stanford.edu/dept/cs\_edu/resources/cslib\_docs/Vector.html ⚙ ☆

## The Stanford libcs106 library, Fall Quarter 2021

```
#include "vector.h"

class Vector<ValueType>
```

This class stores an ordered list of values similar to an array. It supports traditional array selection using square brackets, as well as inserting and removing elements. Operations that access elements by index in  $O(1)$  time. Operations, such as insert and remove, that must rearrange elements run in  $O(N)$  time.

**Constructor**

<a href="#"><u>Vector()</u></a>	$O(1)$	Initializes a new empty vector.
<a href="#"><u>Vector(n, value)</u></a>	$O(N)$	Initializes a new vector storing $n$ copies of the given value.

**Methods**

# Big-O analysis in computer science



WIKIPEDIA  
The Free Encyclopedia

Main page  
Contents  
Featured content  
Current events  
Random article  
Donate to Wikipedia  
Wikipedia store

Interaction

Help  
About Wikipedia  
Community portal  
Recent changes  
Contact page

Tools

What links here  
Related changes  
Upload file  
Special pages  
Permanent link

Article Talk

Read Edit View history

Search Wikipedia



## Binary search algorithm

From Wikipedia, the free encyclopedia  
(Redirected from [Binary search](#))

This article is about searching a fi

In computer science, **binary search**, is a search algorithm that finds the position of a target value in a sorted array. It compares the target value to the middle element of the array. If they are unequal, the search continues on the remaining half of the array. If the target is found, the search ends. If the array is empty, the target is not in the array.

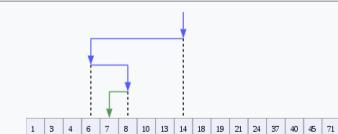
Binary search runs in at worst logarithmic time, making  $O(\log n)$  comparisons, where  $n$  is the number of elements in the array, the  $O$  is Big O notation, and log is the logarithm. Binary search takes constant ( $O(1)$ ) space, meaning that the space taken by the algorithm is the same for any number of elements in the array.<sup>[6]</sup> Although specialized data structures designed for fast searching—such as hash tables—can be searched more efficiently, binary search applies to a wider range of problems.

Although the idea is simple, implementing binary search correctly requires attention to some subtleties about its exit conditions and midpoint calculation.

There are numerous variations of binary search. In particular, fractional cascading speeds up binary searches for the same value in multiple arrays, efficiently solving a series of search problems in computational geometry and numerous other fields. Exponential search extends binary search to unbounded lists. The binary search tree and B-tree data

Worst-case performance	$O(\log n)$
Best-case performance	$O(1)$
Average performance	$O(\log n)$
Worst-case space complexity	$O(1)$

### Binary search algorithm



Visualization of the binary search algorithm where 7 is the target value.

Class	Search algorithm
Data structure	Array
Worst-case performance	$O(\log n)$
Best-case performance	$O(1)$
Average performance	$O(\log n)$
Worst-case space complexity	$O(1)$

## Formal definition of big-O

We say a function  $f(n)$  is “big-O” of another function  $g(n)$   
(written “ $f(n)$  is  $O(g(n))$ ”)  
if and only if

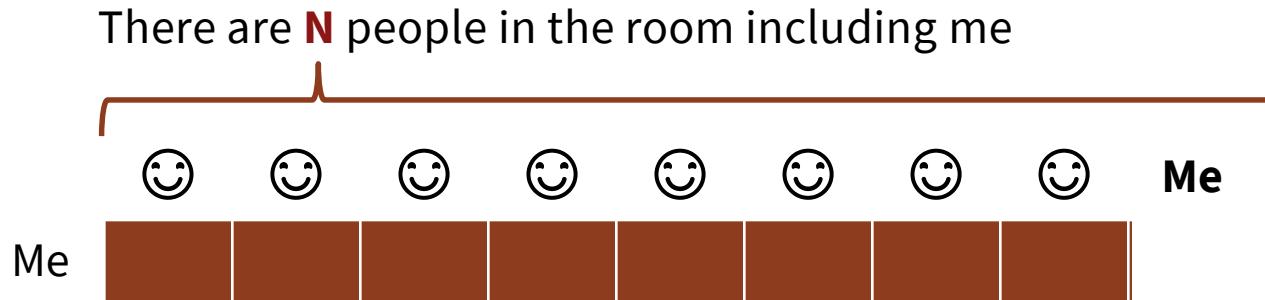
there exist positive constants  $c$  and  $n_0$  such that  
$$f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0.$$

# Before we start, let's get introduced

# Before we start, let's get introduced

Lets say I want to meet each of you today with a handshake and *you tell me* your name...

How many introductions need to happen?



But do I need to shake hands with myself, or tell myself my name?

**N-1 introductions**

## Putting this in Big-O terms

Big-O is a way of categorizing amount of work to be done in general terms, with a focus on:

- ***Rate of growth*** as a function of the problem size N
- What that rate looks like ***on the horizon*** (i.e., for large N)

Therefore, we don't really care about an insignificant  $\pm 1$



## Putting this in Big-O terms

For the first handshake problem, the rate N is important and the -1 constant is not, so **N - 1** introductions becomes:

$$O(N-1)$$

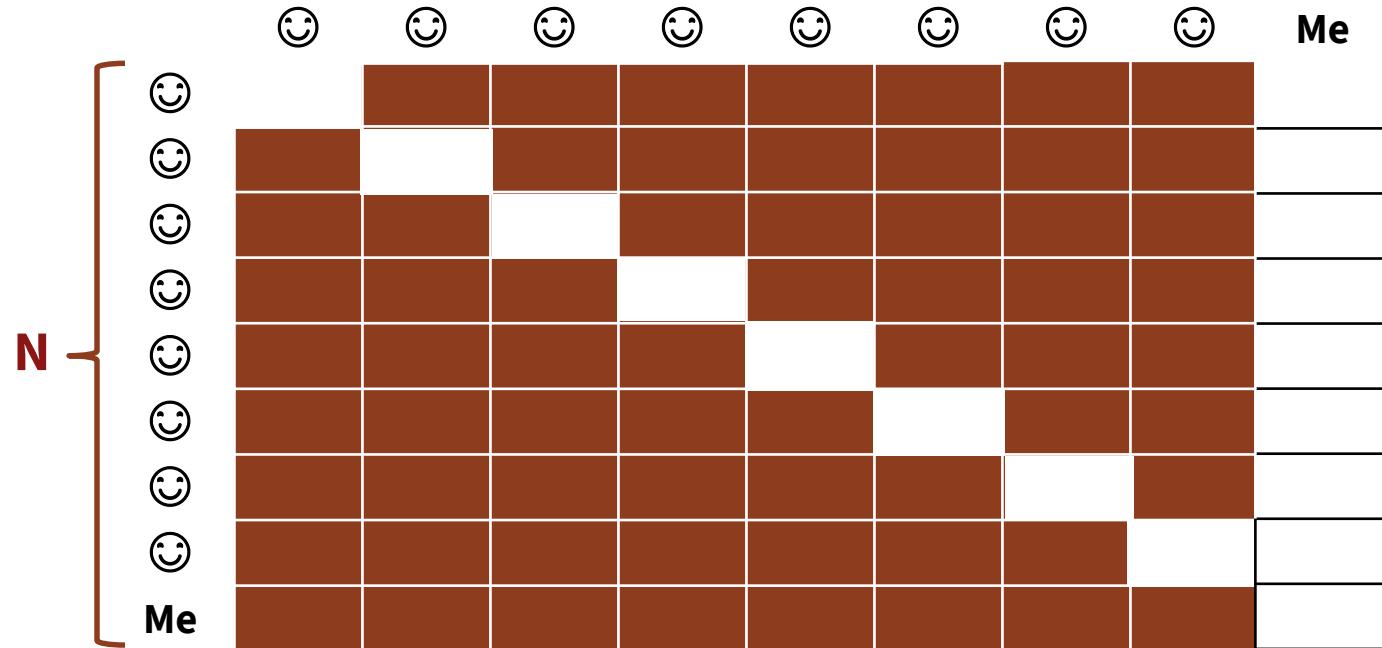
Similarly, if we said that each introduction **takes 3 seconds**, the amount of time is  **$3(N - 1) = 3N - 3$** , but we disregard the constant 3s:

$$O(3N - 3)$$

# Before we start, let's get introduced

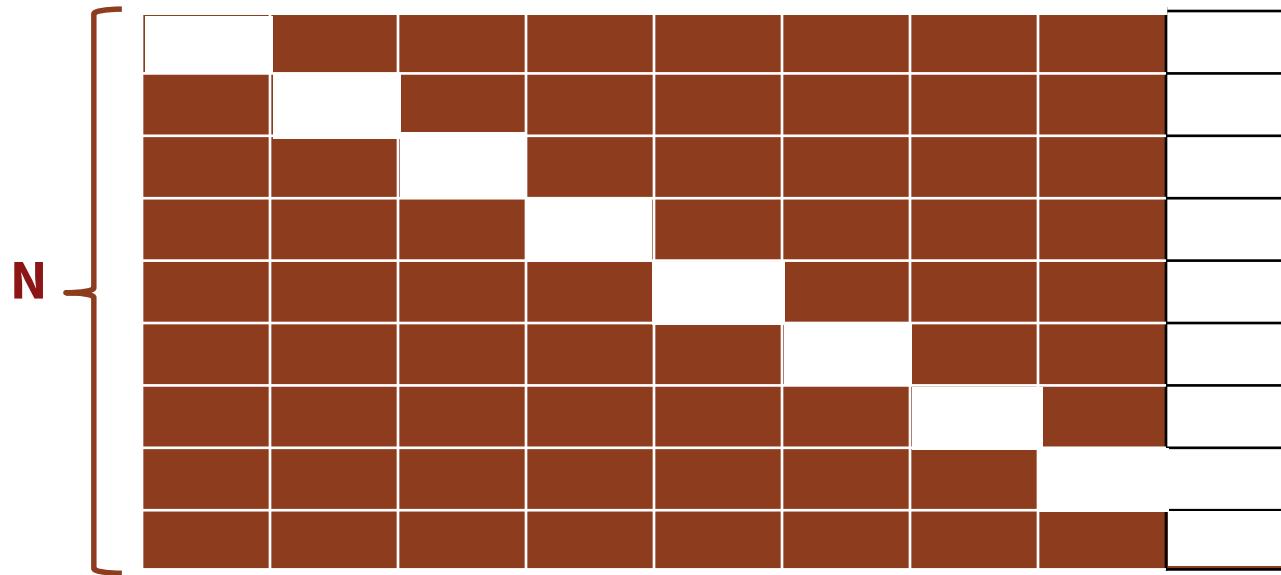
What if I not only want you to be introduced to me, but to each other?

Now how many introductions?  $N^2$



# Before we start, let's get introduced

What if I not only want you to be introduced to me, but to each other?  
Now how many introductions?  $N^2 - 2N + 1$



## Putting this in Big-O terms

For the second handshake problem, the introductions was  $N^2 - N$ :

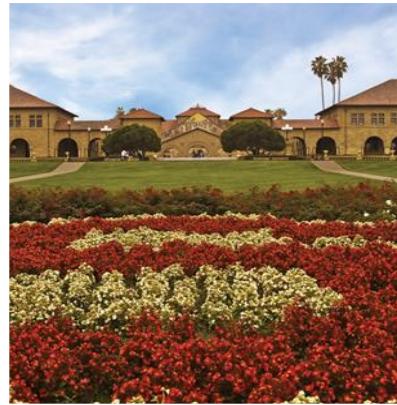
$$O(N^2 - 2N + 1)$$

But wait, didn't we just say that a term of  $+/- N$  was important?

For Big-O, we only care about the **largest term** of the polynomial

# Big-O and Binary Search

SPOILER: FAST!!



## Binary search



**Jump right to the middle** of the region to search, then repeat this process of roughly cutting the array in half again and again until we either find the item or (worst case) cut it down to nothing.

Worst case cost is number of times we can divide length in half:

$$O(\log_2 N)$$

# Putting it all together

Binary search

Handshake #1

Handshake #2

MANY important  
optimization and  
other problems

$\log_2 n$	$n$	$n \log_2 n$	$n^2$	$2^n$
2	4	8	16	
3	8	24	64	256
4	16	64	256	65,536
5	32	160	1,024	4,294,967,296
6	64			
7	128			
8	256			
9	512			
10	1,024			
30	2,700,000,000			

Naïve  
Recursive  
Fibonacci  
( $O(1.6^n)$ )

$\log_2 n$	$n$	$n \log_2 n$	$n^2$	$2^n$
2	4	8	16	16
3	8	24	64	256
4	16	64	256	65,536
5	32	160	1,024	4,294,967,296
6	64			2.4s
7	128			Easy!
8	256			
9	512			
10	1,024			
30	2,700,000,000			

## **Traveling Salesperson Problem:**

We have a bunch of cities to visit. In what order should we visit them to minimize total travel distance?



## Traveling Salesperson Problem:

We have a bunch of cities to visit. In what order should we visit them to minimize total travel distance?



Exhaustively try all orderings:  $O(n!)$

Use current best known algorithm:  $O(n^2n)$

Maybe we could invent an algorithm that fits in our rightmost column:  $O(2^n)$





So let's say we come up with a way to solve  
Traveling Salesperson Problem in  $O(2^n)$ .

It would take **4 days** to solve Traveling  
Salesperson Problem on 50 state capitals.

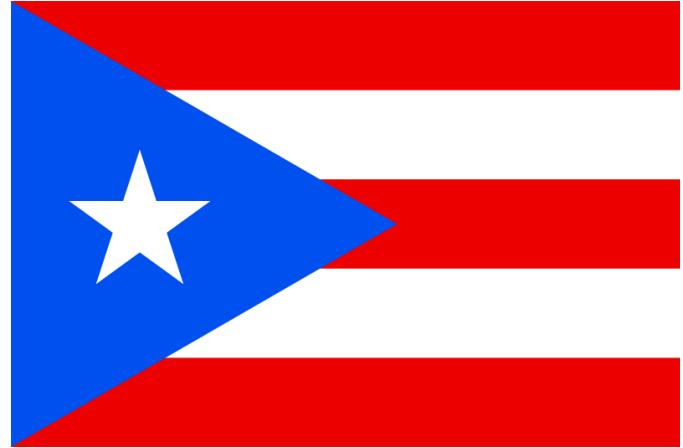


## Two *tiny* little updates

Imagine we approve statehood for US territory Puerto Rico

- Add San Juan, the capital city

Also add Washington, DC



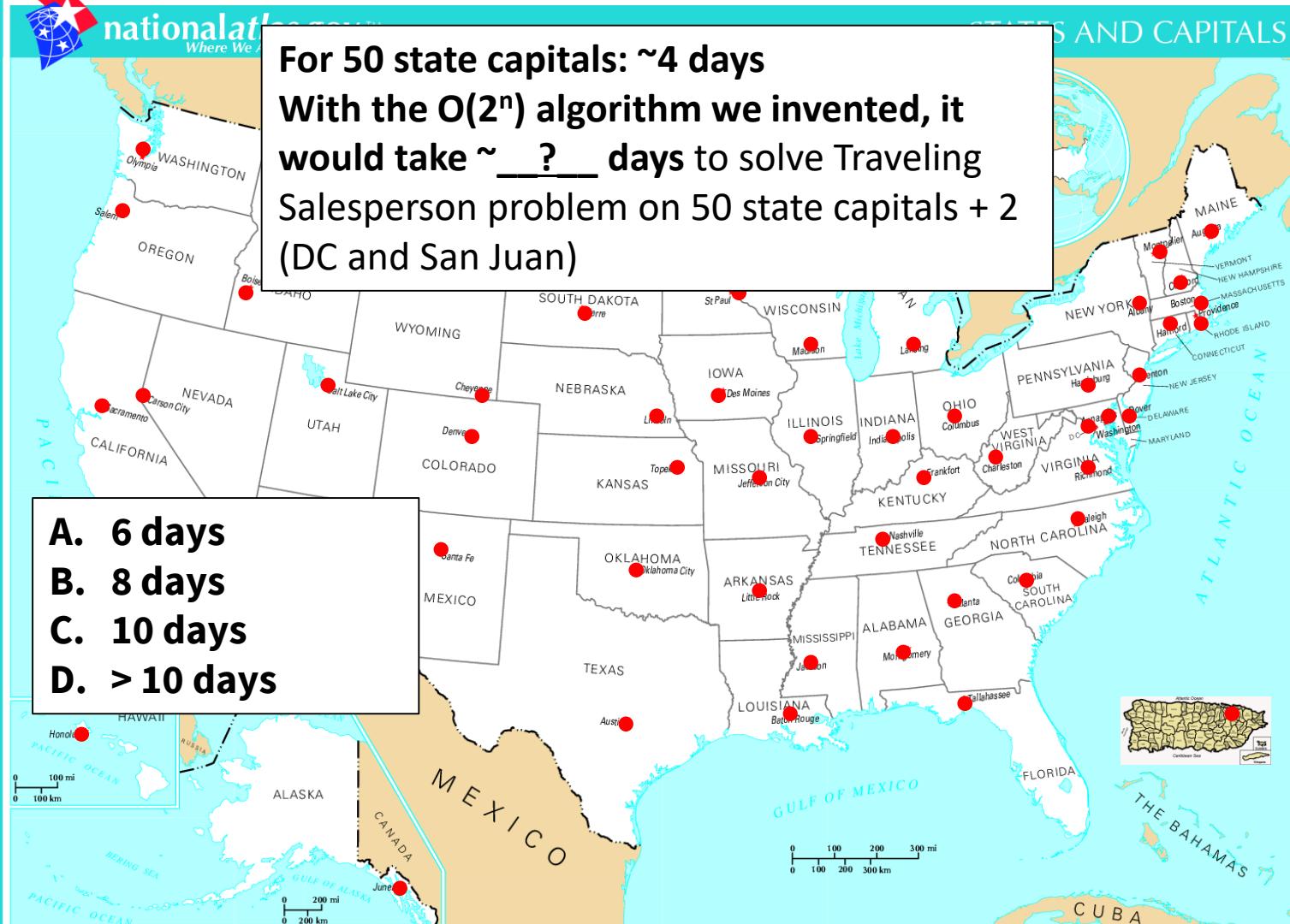
This work has been released into the [public domain](#) by its author, [Madden](#).  
This applies worldwide.

**Now 52 capital cities instead of 50**



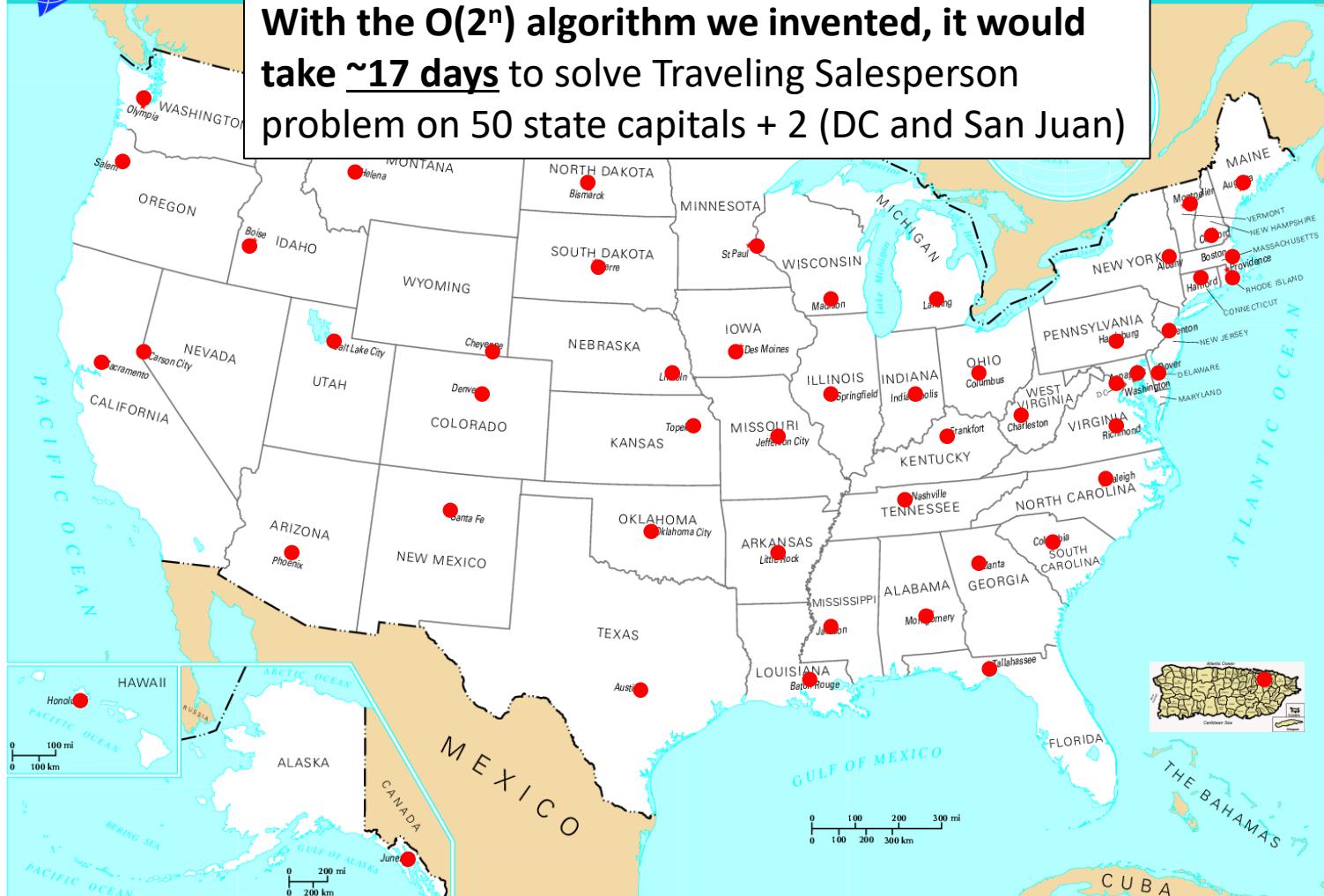
For 50 state capitals: ~4 days  
With the  $O(2^n)$  algorithm we invented, it would take ~   ? days to solve Traveling Salesperson problem on 50 state capitals + 2 (DC and San Juan)

- A. 6 days
- B. 8 days
- C. 10 days
- D. > 10 days





**With the  $O(2^n)$  algorithm we invented, it would take ~17 days to solve Traveling Salesperson problem on 50 state capitals + 2 (DC and San Juan)**





Sacramento is not exactly the most interesting or important city in California (sorry, Sacramento).

**What if we add the 12 biggest non-capital cities in the United States to our map?**



**With the  $O(2^n)$  algorithm we invented,  
It would take **194 YEARS** to solve Traveling  
Salesman problem on 64 cities (state capitals +  
DC + San Juan + 12 biggest non-capital cities)**



$\log_2 n$	$n$	$n \log_2 n$	$n^2$	$2^n$
2	4	8	16	16
3	8	24	64	256
4	16	64	256	65,536
5	32	160	1,024	4,294,967,296
6	64	384	4,096	$1.84 \times 10^{19}$
7	128			<b>194 YEARS</b>
8	256			
9	512			
10	1,024			
30	<b>2,700,000,000</b>			

$\log_2 n$	$n$	$n \log_2 n$	$n^2$	$2^n$
2	4	8	16	16
3	8	24	64	256
4	16	64	256	65,536
5	32	160	1,024	4,294,967,296
6	64	384	4,096	$1.84 \times 10^{19}$
7	128	896	16,384	$3.40 \times 10^{38}$
8	256			<b>3.59E+21 YEARS</b>
9	512			
10	1,024			
30	<b>2,700,000,000</b>			

$\log_2 n$	$n$	$n \log_2 n$	$n^2$	$2^n$
2	4	8	16	16
3	8	24	64	256
4	16	64	256	65,536
5	32	160	1,024	4,294,967,296
6	64	384	4,096	$1.84 \times 10^{19}$
7	128	896	16,384	$3.40 \times 10^{38}$
8	256	<b>3,590,000,000,000,000,000,000 YEARS</b>		
9	512			
10	1,024			
30	<b>2,700,000,000</b>			

$\log_2 n$	$n$	$n \log_2 n$	$n^2$	$2^n$
2	4	8	16	16
3	8	24	64	256
4	16	64	256	65,536
5	32	160	1,024	4,294,967,296
6	64	384	4,096	$1.84 \times 10^{19}$
7	128	896	16,384	$3.40 \times 10^{38}$
8	256	2,048	65,536	$1.16 \times 10^{77}$
9	512			
10	1,024			
30	<b>2,700,000,000</b>			

For comparison: there are about  $10^{80}$  atoms in the universe. No big deal.

$\log_2 n$	$n$	$n \log_2 n$	$n^2$	$2^n$
2	4	8	16	16
3	8	24	64	256
4	16	64	256	65,536
5	32	160	1,024	4,294,967,296
6	64	384	4,096	$1.84 \times 10^{19}$
7	128	896	16,384	$3.40 \times 10^{38}$
8	256	2,048	65,536	$1.16 \times 10^{77}$
9	512	4,608	262,144	$1.34 \times 10^{154}$
10	1,024			<b>1.42E+137 YEARS</b>
30	<b>2,700,000,000</b>			

$\log_2 n$	$n$	$n \log_2 n$	$n^2$	$2^n$
2	4	8	16	16
3	8	24	64	256
4	16	64	256	65,536
5	32	160	1,024	4,294,967,296
6	64	384	4,096	$1.84 \times 10^{19}$
7	128	896	16,384	$3.40 \times 10^{38}$
8	256	2,048	65,536	$1.16 \times 10^{77}$
9	512	4,608	262,144	$1.34 \times 10^{154}$
10	1,024	10,240 (.000003s)	1,048,576 (.0003s)	$1.80 \times 10^{308}$
30	2,700,000,000	84,591,843,105 (28s)	7,290,000,000,000,000 (77 years)	LOL

$\log_2 n$	$n$	$n \log_2 n$	$n^2$	$2^n$
2	4	8	16	16
3	8	24	64	256
4	16	64	256	65,536
5	32	160	1,024	4,294,967,296
6	64	384	4,096	$1.84 \times 10^{19}$
7	128	896	16,384	$3.40 \times 10^{38}$
8	256	2,048	65,536	$1.16 \times 10^{77}$
9	512	4,608	262,144	$1.34 \times 10^{154}$
10	1,024	10,240 (.000003s)	1,048,576 (.0003s)	$1.80 \times 10^{308}$
31	<b>2,700,000,000</b>	84,591,843,105 (28s)	7,290,000,000,000,000 00 (77 years)	$1.962227 \times 10^{812,780,998}$

$\log_2 n$	$n$	$n \log_2 n$	$n^2$	$2^n$
2	4	8	16	16
3	8	24	64	256
4	16	64	256	65,536
5	32	160	1,024	4,294,967,296
6	64	384	4,096	$1.84 \times 10^{19}$
7	128	896	16,384	$3.40 \times 10^{38}$
8	256	2,048	65,536	$1.16 \times 10^{77}$
9	512	4,608	262,144	$1.34 \times 10^{154}$
10	1,024	9,216	1,048,576	$1.80 \times 10^{308}$
11	2,048	18,432	(.0003s)	
12	4,096	36,864		
13	8,192	73,728		
14	16,384	147,456		
15	32,768	294,912		
16	65,536	589,824		
17	131,072	1,179,648		
18	262,144	2,359,296		
19	524,288	4,718,592		
20	1,048,576	9,437,184		
21	2,097,152	18,874,368		
22	4,194,304	37,748,736		
23	8,388,608	75,497,472		
24	16,777,216	150,994,944		
25	33,554,432	301,989,888		
26	67,108,864	603,979,776		
27	134,217,728	1,207,959,552		
28	268,435,456	2,415,918,104		
29	536,870,912	4,831,836,208		
30	1,073,741,824	9,663,672,416		
31	2,147,483,648	19,327,344,832		
32	4,294,967,296	38,654,688,664		
33	8,589,934,592	77,309,377,328		
34	17,179,869,184	154,618,754,656		
35	34,359,738,368	309,237,509,312		
36	68,719,476,736	618,475,018,624		
37	137,438,953,472	1,236,950,037,248		
38	274,877,906,944	2,473,900,074,496		
39	549,755,813,888	4,947,800,148,992		
40	1,099,511,627,776	9,895,600,297,984		
41	2,199,023,255,552	19,791,200,595,968		
42	4,398,046,511,104	39,582,400,191,936		
43	8,796,092,022,208	79,164,800,383,872		
44	17,592,184,044,416	158,329,600,767,744		
45	35,184,368,088,832	316,658,201,535,488		
46	70,368,736,177,664	633,316,403,070,976		
47	140,737,472,355,328	1,266,632,806,141,952		
48	281,474,944,710,656	2,533,265,612,283,904		
49	562,949,889,421,312	5,066,531,224,567,808		
50	1,125,899,778,842,624	10,133,062,449,135,616		
51	2,251,799,557,685,248	20,266,124,898,271,232		
52	4,503,599,115,370,496	40,532,249,796,542,464		
53	9,007,198,230,740,992	81,064,499,593,084,928		
54	18,014,396,461,481,984	162,128,999,186,169,856		
55	36,028,792,922,963,968	324,257,998,372,339,712		
56	72,057,585,845,927,936	648,515,996,744,679,424		
57	144,115,171,691,855,872	1,296,031,993,489,358,848		
58	288,230,343,383,711,744	2,592,063,986,978,717,696		
59	576,460,686,767,423,488	5,184,127,973,957,435,392		
60	1,152,921,373,534,846,976	10,368,255,947,914,870,784		
61	2,305,842,747,069,693,952	20,736,511,895,829,741,568		
62	4,611,685,494,139,387,904	41,473,023,791,659,483,136		
63	9,223,370,988,278,775,808	82,946,047,583,318,966,272		
64	18,446,741,976,557,551,616	165,892,095,166,637,932,544		
65	36,893,483,953,115,103,232	331,784,190,333,275,865,088		
66	73,786,967,906,230,206,464	663,568,380,666,551,730,176		
67	147,573,935,812,460,412,928	1,327,136,761,333,103,460,352		
68	295,147,871,624,920,825,856	2,654,273,522,666,206,920,704		
69	590,295,743,249,841,651,712	5,308,547,045,332,413,841,408		
70	1,180,591,486,499,683,303,424	10,617,094,090,664,827,682,816		
71	2,361,182,972,999,366,606,848	21,234,188,181,329,655,365,632		
72	4,722,365,945,998,733,213,696	42,468,376,362,659,310,731,264		
73	9,444,731,891,997,466,427,392	84,936,752,725,318,621,462,528		
74	18,889,463,783,994,933,854,784	169,873,505,450,637,242,925,056		
75	37,778,927,567,989,867,709,568	339,747,010,901,274,484,850,112		
76	75,557,855,135,979,735,419,136	679,494,021,802,548,969,700,224		
77	151,115,710,271,959,470,838,272	1,358,988,043,605,097,939,400,448		
78	302,231,420,543,918,941,676,544	2,717,976,087,210,195,878,800,896		
79	604,462,841,087,837,883,353,088	5,435,952,174,420,391,757,600,192		
80	1,208,925,682,175,675,766,706,176	10,871,904,348,840,783,515,200,384		
81	2,417,851,364,351,351,533,412,352	21,743,808,697,681,567,030,400,768		
82	4,835,702,728,702,703,066,824,704	43,487,617,395,363,134,060,800,136		
83	9,671,405,457,405,406,133,648,408	86,975,234,790,726,268,121,600,272		
84	19,342,810,914,810,812,267,296,816	173,950,469,581,452,536,243,200,544		
85	38,685,621,829,621,624,534,592,832	347,900,939,162,904,572,486,400,184		
86	77,371,243,659,243,248,068,184,664	695,801,878,325,808,144,972,800,368		
87	154,742,487,318,487,496,336,368,336	1,391,603,756,651,616,289,945,600,736		
88	309,484,974,636,974,992,672,736,672	2,783,207,513,303,232,579,891,200,152		
89	618,969,949,273,949,985,344,473,344	5,566,414,026,606,465,159,782,400,304		
90	1,237,939,898,547,898,970,688,946,946	11,132,828,053,212,930,319,564,800,608		
91	2,475,879,797,095,797,941,377,893,893	22,265,656,106,425,860,638,129,600,128		
92	4,951,759,594,191,595,882,755,787,787	44,531,312,212,851,721,276,259,200,256		
93	9,903,519,188,383,188,765,511,575,575	89,062,624,425,673,442,552,518,400,512		
94	19,807,038,376,766,376,531,022,151,151	178,125,248,851,346,885,104,536,800,800		
95	39,614,076,753,532,753,062,044,302,302	356,250,497,672,693,770,268,073,600,600		
96	79,228,153,507,065,507,124,088,604,604	712,500,995,345,387,540,536,146,400,400		
97	158,456,307,014,131,014,248,176,808,808	1,425,001,990,687,775,080,572,292,800,800		
98	316,912,614,028,262,028,496,353,616,353	2,850,003,981,375,550,160,144,585,600,600		
99	633,825,228,056,524,056,992,707,232,707	5,700,007,962,750,100,320,288,171,200,200		
100	1,267,650,456,112,048,048,985,414,464,414	11,400,015,925,500,200,640,576,342,400,400		
101	2,535,300,912,224,096,096,970,828,928,828	22,800,031,851,000,400,120,153,484,800,800		
102	5,070,601,824,448,192,192,941,657,857,857	45,600,063,702,000,800,240,306,968,000,800		
103	10,141,203,648,896,384,384,923,315,715,715	91,200,127,404,000,160,480,613,936,000,800		
104	20,282,407,297,792,768,768,946,631,431,431	182,400,254,808,000,320,960,127,872,000,800		
105	40,564,814,595,585,536,536,993,262,862,862	364,800,509,616,000,640,920,255,744,000,800		
106	81,129,629,191,171,072,072,986,525,725,725	729,600,109,232,000,128,180,510,488,000,800		
107	162,259,258,382,342,144,144,973,051,451,451	1459,200,218,464,000,256,360,020,976,000,800		
108	324,518,516,764,684,288,288,946,102,902,902	2918,400,436,928,000,512,720,040,952,000,800		
109	648,037,033,529,368,576,576,943,204,804,804	5836,800,873,856,000,102,440,080,904,000,800		
110	1,296,074,067,058,736,152,152,946,408,608,604	11673,600,176,712,000,204,880,160,808,000,800		
111	2,592,148,134,117,472,304,304,943,816,316,308	23347,200,353,424,000,409,760,320,816,000,800		
112	5,184,296,268,234,944,608,608,943,632,632,308	46694,400,706,848,000,819,520,640,632,000,800		
113	10,368,592,536,469,889,216,216,943,264,324,308	93388,800,141,696,000,163,880,128,324,000,800		
114	20,737,185,072,939,778,432,432,943,528,648,308	186777,600,283,392,000,327,760,256,648,000,800		
115	41,474,370,145,879,556,864,864,943,056,396,308	373555,200,566,784,000,655,520,512,396,000,800		
116	82,948,740,291,759,113,728,728,943,112,792,308	747110,400,113,568,000,131,040,104,792,000,800		
117	165,897,480,583,518,227,456,456,943,224,584,308	1494220,800,227,136,000,262,080,208,584,000,800		
118	331,784,960,167,036,454,912,912,943,448,168,308	2988441,600,454,272,000,524,160,448,000,800		
119	663,568,380,334,072,909,825,825,943,896,336,308	5976883,200,909,544,000,104,832,896,000,800		
120	1,327,136,760,668,145,819,651,651,943,792,672,308	11953766,400,183,698,000,209,664,184,672,000,800		
121	2,654,273,520,136,391,603,303,303,943,584,344,308	23907532,800,367,396,000,418,328,568,344,000,800		
122	5,308,547,040,272,783,206,606,606,943,168,688,308	47815065,600,734,788,000,836,656,136,688,000,800		
123	10,617,094,080,545,566,413,213,213,943,336,376,308	95630131,200,148,576,000,167,312,272,376,000,800		
124	21,234,188,161,090,133,832,426,426,943,672,752,308	191260262,400,297,064,000,334,624,544,752,000,800		
125	42,468,376,322,180,267,664,852,852,943,344,504,308	382520524,800,594,128,000,669,248,108,504,000,800		
126	84,936,752,644,360,535,331,711,711,943,688,008,308	765041049,600,118,864,000,133,880,216,008,000,800		
127	169,873,505,288,720,870,662,422,422,943,376,016,308	1530082099,200,237,728,000,267,760,432,016,000,800		
128	339,747,010,577,440,141,334,844,844,943,752,032,308	3060164198,400,475,456,000,535,520,864,032,000,800		
129	679,494,021,154,880,282,669,669,943,504,064,308	6120328397,600,950,912,000,107,040,138,064,000,800		
130	1,358,988,042,309,760,564,339,339,943,032,128,308	1224065675,200,190,184,000,214,080,276,128,000,800		
131	2,717,976,084,619,520,128,678,678,943,064,256,308	2448131350,400,380,368,000,428,160,552,256,000,800		
132	5,435,952,169,238,040,256,356,356,943,128,512,308	4896262700,800,760,736,000,856,320,104,512,000,800		
133	10,871,904,338,476,080,512,712,712,943,256,024,308	9792525401,600,152,144,000,171,640,208,024,000,800		
134	21,743,808,676,952,160,104,424,424,943,512,048,308	19585050803,200,304,288,000,343,280,416,048,000,800		
135	43,487,617,353,904,320,320,848,848,943,024,096,308	39170101606,400,608,576,000,686,560,832,096,000,800		
136	86,975,234,707,808,640,640,169,169,943,048,192,308	78340203212,800,121,152,000,137,112,168,192,000,800		
137	173,950,469,415,616,320,320,338,338,943,096,384,308	156680406425,600,242,304,000,274,224,336,384,000,800		
138	347,900,939,831,232,640,640,676,676,943,192,768,308	313360812851,200,484,608,000,548,448,672,768,000,800		
139	695,801,878,662,464,128,128,135,135,943,384,152,308	626721635702,400,969,216,000,109,888,352,152,0		

# In Conclusion

- **NOT worth doing:** Optimization of your code that **just trims** a bit
  - › Like that +/-1 handshake—we don't need to worry ourselves about it!
  - › Just write clean, easy-to-read code!!!!
- **MAY be worth doing:** Optimization of your code that **changes Big-O**
  - › If performance of a particular function is important, focus on this!
  - › (*but if performance of the function is not very important, for example it will only run on small inputs, focus on just writing clean, easy-to-read code!!*)
- (Also remember that efficiency is not necessarily a virtue—first and foremost focus on correctness, both technical and ethical/moral/societal justice)

# Programming Abstractions

CS106B

Cynthia Bailey Lee  
Julie Zelenski

## Today's Topics

Recursion!

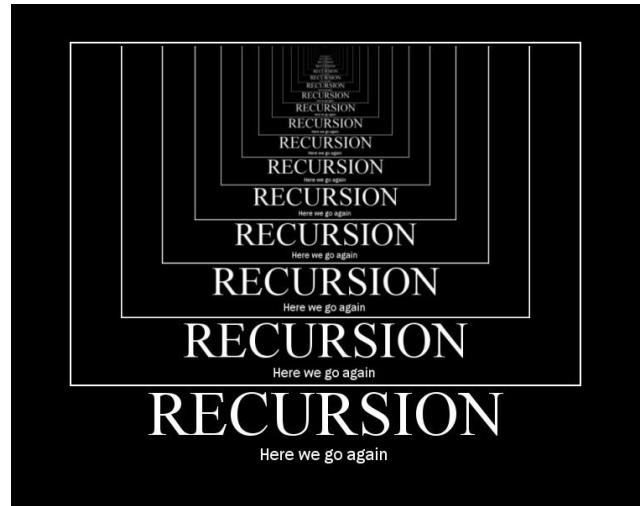
- Functions calling functions

Next time:

- More recursion! It's Recursion Week!
  - › Like Shark Week, but more nerdy

# Recursion!

The exclamation point isn't there only because this is so exciting; it also relates to our first recursion example....



## Factorial!

$$n! = n(n - 1)(n - 2)(n - 3)(n - 4) \dots (3)(2)(1)$$

This could be a really long expression!

**Recursion is a technique for tackling large or complicated problems by just “eating” one “bite” of the problem at a time.**

# Factorial!

$$n! = n(n - 1)(n - 2)(n - 3)(n - 4) \dots (2)(1)$$

An alternate mathematical formulation:

$$n! = \begin{cases} 1 & \text{if } n = 1 \\ n(n - 1)! & \text{otherwise} \end{cases}$$

## Translated to code

```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * someFunctionThatKnowsFactorialOfNMinus1();
    }
}
```

# Factorial!

$$n! = n(n - 1)(n - 2)(n - 3)(n - 4) \dots (2)(1)$$

An alternate mathematical formulation:

$$n! = \begin{cases} 1 & \text{if } n = 1 \\ n(n - 1)! & \text{otherwise} \end{cases}$$

## Translated to code

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

# The recursive function pattern

**Always two parts:**

**Base case:**

- This problem is so tiny, it's hardly a problem anymore! Just give answer.

**Recursive case:**

- This problem is still a bit large, let's bite off just one piece, and delegate the remaining work to recursion.

**Translated to code**

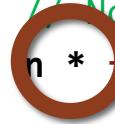
```
int factorial(int n) {  
    if (n == 1) { // Easy! Return trivial answer  
        return 1;  
    } else { // Not easy enough to finish yet! Do 1 piece  
        return n * factorial(n - 1);  
    }  
}
```

# The recursive function pattern

## **Recursive case:**

- This problem is still a bit large, **let's bite off just one piece**, and **delegate the remaining work to recursion**.

```
int factorial(int n) {  
    if (n == 1) { // Easy! Return trivial answer  
        return 1;  
    } else { // Not easy enough to finish yet! Do 1 piece  
        return n * factorial(n - 1);  
    }  
}
```



This is an example of “one piece” of the problem—just doing one of the many, many multiplications required for factorial.

# The recursive function pattern

## Recursive case:

- This problem is still a bit large, let's bite off just one piece, and delegate the remaining work to recursion.

```
int factorial(int n) {  
    if (n == 1) { // Easy! Return trivial answer  
        return 1;  
    } else { // Not easy enough to finish yet! Do 1 piece  
        return n * factorial(n - 1);  
    }  
}
```

This is an example  
“delegating the  
remaining work”—all the  
other multiplications—to  
the recursive call.

# The recursive function pattern

## Recursive case:

- This problem is still a bit large, let's bite off just one piece, and delegate the remaining work to recursion.

```
int factorial(int n) {  
    if (n == 1) { // Easy! Return trivial answer  
        return 1;  
    } else { // Not easy enough to finish yet! Do 1 piece  
        return n * factorial(n - 1);  
    }  
}
```



This is an example of “one piece” of the problem—just doing one of the many, many multiplications required for factorial.

This is an example “delegating the remaining work”—all the other multiplications—to the recursive call.

## Recap: the recursive function pattern

- Recursion is a way of taking a big problem and repeatedly breaking it into smaller and smaller pieces until it is so small that it can be so easily solved that it almost doesn't even need solving.
- There are two parts of a recursive algorithm:
  - › **base case:** where we identify that the problem is so small that we trivially solve it and return that result
  - › **recursive case:** where we see that the problem is still a bit too big for our taste, so we chop it into smaller bits and call **ourselves** (the function we are in now) on the smaller bits to find out the answer to the problem we face

# Digging deeper in the recursion

Looking at how recursion works “under the hood”

## Factorial!

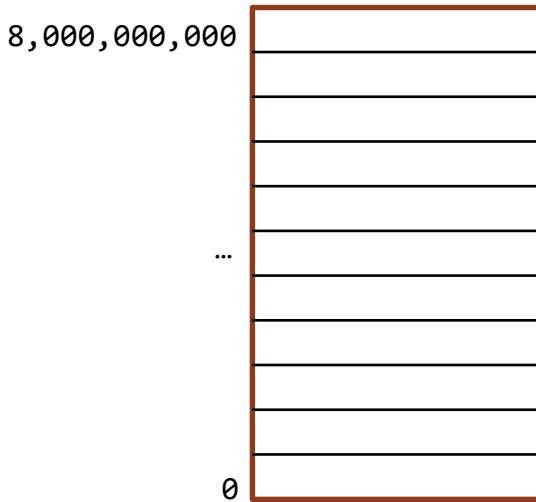
```
int factorial(int n) {  
    cout << n << endl; // **Added for this question**  
    if (n == 1) { // Easy! Return trivial answer  
        return 1;  
    } else { // Not easy enough to finish yet! Do 1 piece  
        return n * factorial(n - 1);  
    }  
}
```

What is the **third** thing **printed** when we call `factorial(4)`?

- A. 1
- B. 2
- C. 3
- D. 4
- E. Other/none/more

# How does this look in memory? A little background...

- A computer's memory is like a giant Vector/array, and like a Vector, we start counting at index 0.
- We typically draw memory vertically (rather than horizontally like a Vector), with index 0 at the bottom.
- A typical laptop's memory has billions of these indexed slots (one byte each)

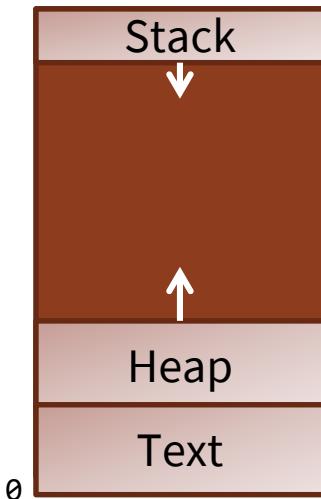


\* Take CS107 to learn much more!!

Stanford University

# How does this look in memory? A little background...

- Broadly speaking, we divide memory into regions:
  - **Text:** the program's own code (needs to be in memory so it can run!)
  - **Heap:** we'll learn about this later in CS106B!
  - **Stack:** this is where local variables for each function are stored.

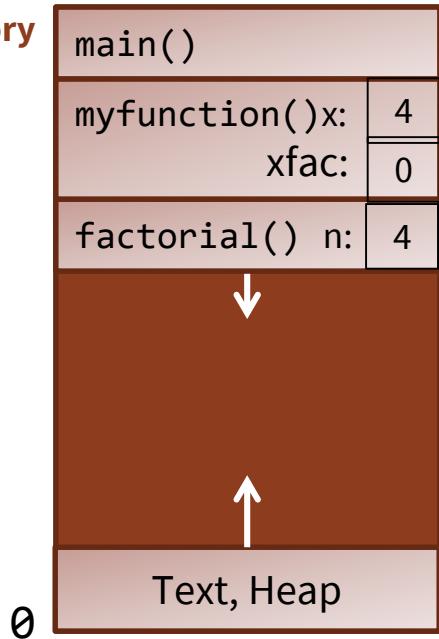


\* Take CS107 to learn much more!!

Stanford University

## How does this look in memory?

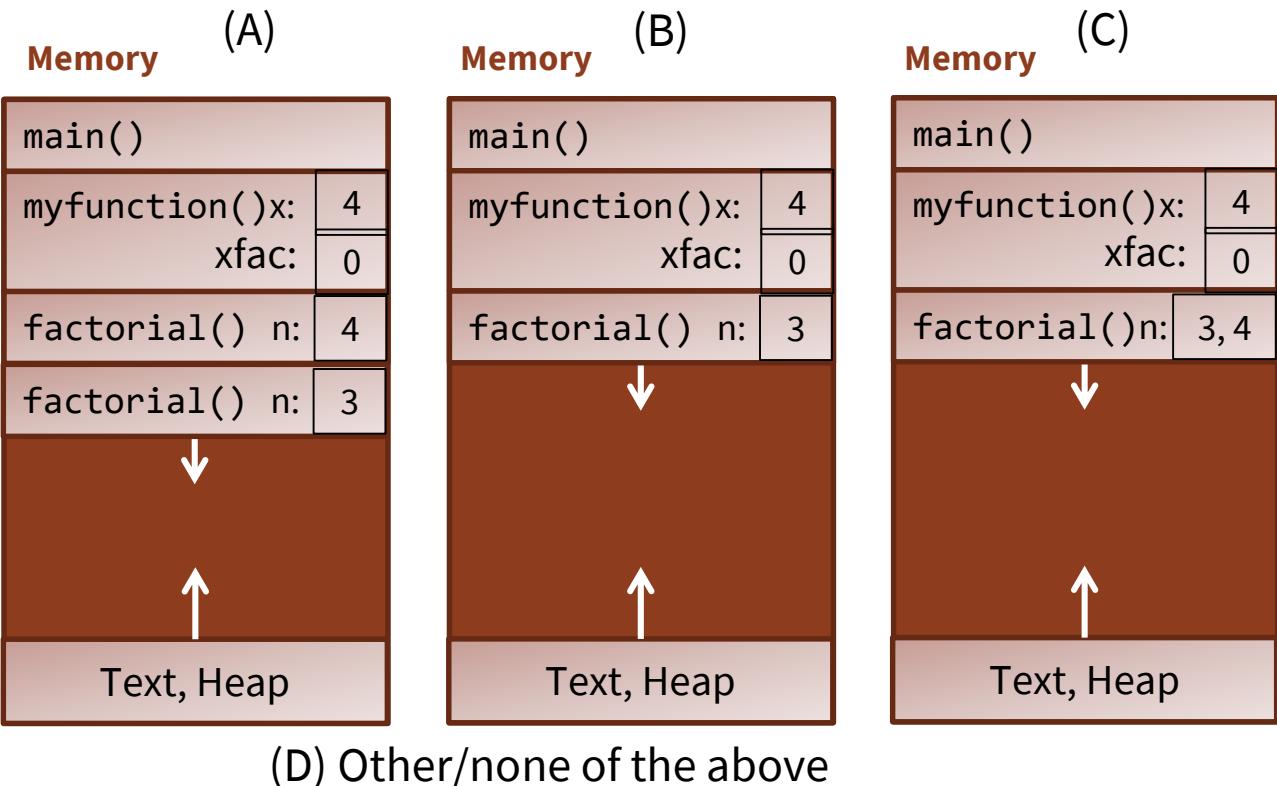
Memory



Recursive code

```
int factorial(int n) {
    cout << n << endl;
    if (n == 1) return 1;
    else return n * factorial(n - 1);
}
```

```
void myfunction(){
    int x = 4;
    int xfac = 0;
    xfac = factorial(x);
}
```



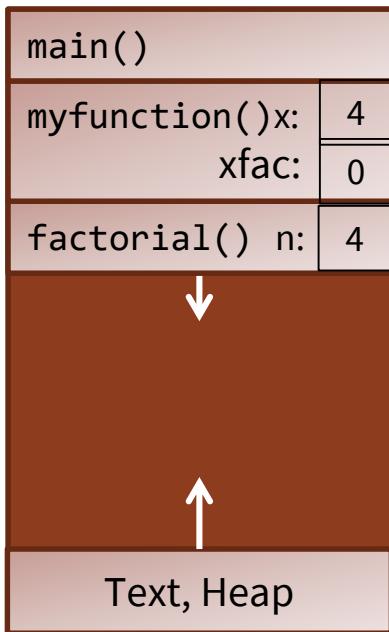
**Fun fact:**  
**The “stack” part of memory is a stack**

Function **call** = **push** a stack frame

Function **return** = **pop** a stack frame

\* Take CS107 to learn much more!!

## The “stack” part of memory is a stack

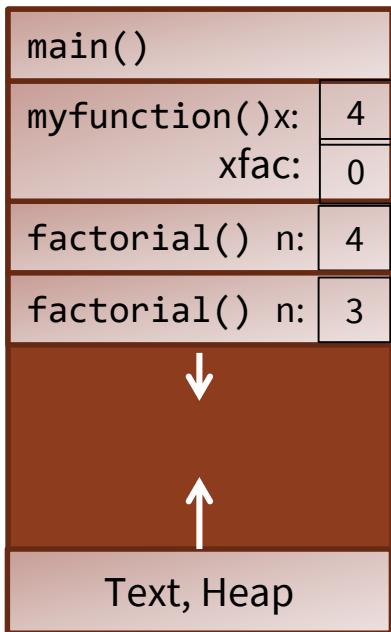


### Recursive code

```
int factorial(int n) {  
    cout << n << endl;  
    if (n == 1) return 1;  
    else return n * factorial(n - 1);  
}
```

```
void myfunction(){  
    int x = 4;  
    int xfac = 0;  
    xfac = factorial(x);  
}
```

## The “stack” part of memory is a stack

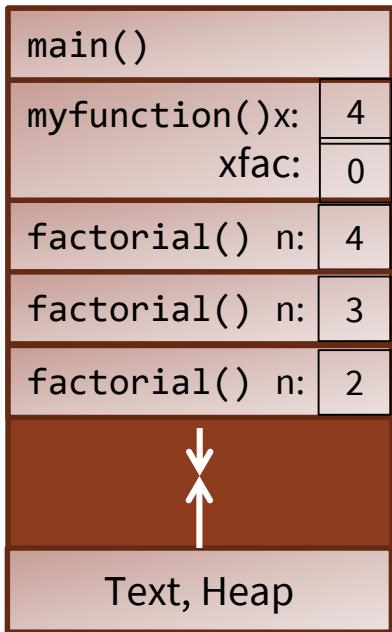


### Recursive code

```
int factorial(int n) {  
    cout << n << endl;  
    if (n == 1) return 1;  
    else return n * factorial(n - 1);  
}
```

```
void myfunction(){  
    int x = 4;  
    int xfac = 0;  
    xfac = factorial(x);  
}
```

## The “stack” part of memory is a stack



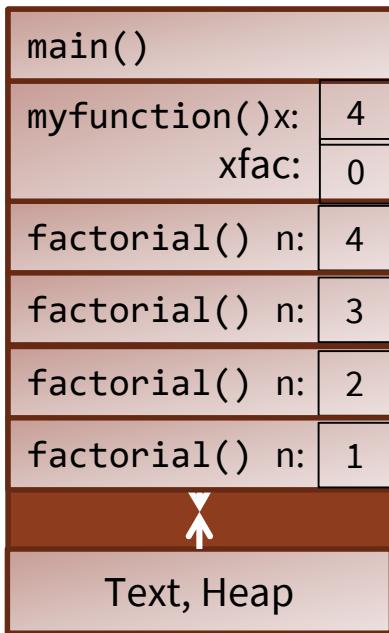
### Recursive code

```
int factorial(int n) {  
    cout << n << endl;  
    if (n == 1) return 1;  
    else return n * factorial(n - 1);  
}
```

```
void myfunction(){  
    int x = 4;  
    int xfac = 0;  
    xfac = factorial(x);  
}
```

Answer: 3<sup>rd</sup>  
thing  
printed is 2

## The “stack” part of memory is a stack



### Recursive code

```
int factorial(int n) {
    cout << n << endl;
    if (n == 1) return 1;
    else return n * factorial(n - 1);
}

void myfunction(){
    int x = 4;
    int xfac = 0;
    xfac = factorial(x);
}
```

# Factorial!

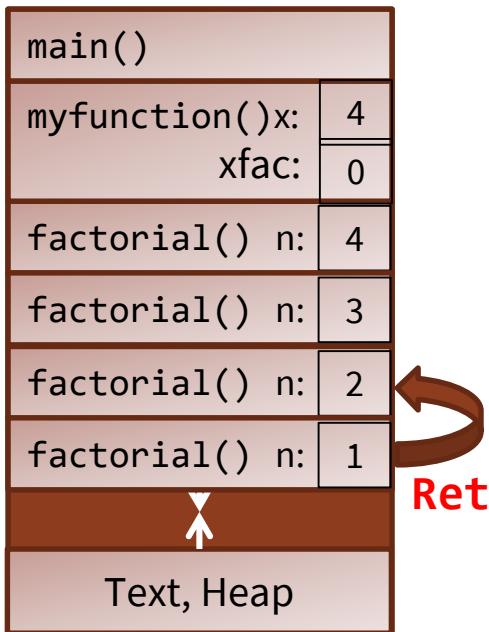
What is the **fourth** value ever **returned** when we call `factorial(4)`?

- A. 4
- B. 6
- C. 10
- D. 24
- E. Other/none/more than one

## Recursive code

```
int factorial(int n) {  
    cout << n << endl;  
    if (n == 1) return 1;  
    else return n * factorial(n - 1);  
}  
  
void myfunction(){  
    int x = 4;  
    int xfac = 0;  
    xfac = factorial(x);  
}
```

## The “stack” part of memory is a stack



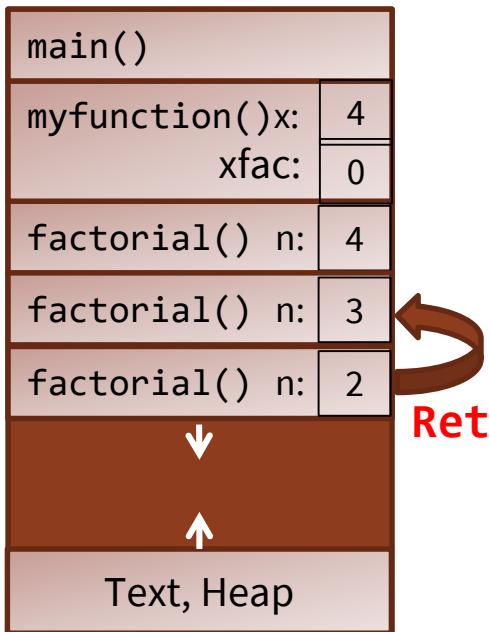
### Recursive code

```
int factorial(int n) {  
    cout << n << endl;  
    if (n == 1) return 1;  
    else return n * factorial(n - 1);  
}
```

```
void myfunction(){  
    int x = 4;  
    int xfac = 0;  
    xfac = factorial(x);  
}
```

**Return 1**

## The “stack” part of memory is a stack

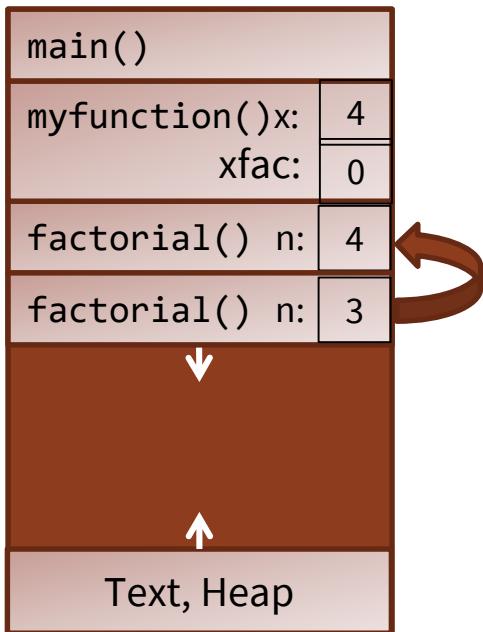


### Recursive code

```
int factorial(int n) {  
    cout << n << endl;  
    if (n == 1) return 1;  
    else return n * factorial(n - 1);  
}
```

```
void myfunction(){  
    int x = 4;  
    int xfac = 0;  
    xfac = factorial(x);  
}
```

## The “stack” part of memory is a stack



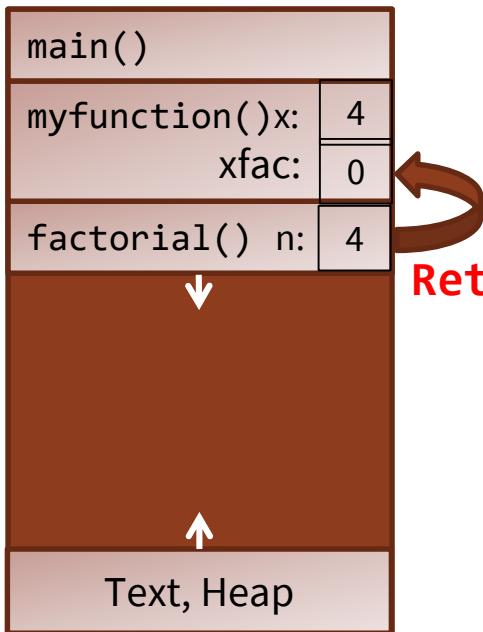
### Recursive code

```
int factorial(int n) {  
    cout << n << endl;  
    if (n == 1) return 1;  
    else return n * factorial(n - 1);  
}
```

**Return 6**

```
void myfunction(){  
    int x = 4;  
    int xfac = 0;  
    xfac = factorial(x);  
}
```

## The “stack” part of memory is a stack



### Recursive code

```
int factorial(int n) {  
    cout << n << endl;  
    if (n == 1) return 1;  
    else return n * factorial(n - 1);
```

Return} 24

```
void myfunction(){  
    int x = 4;  
    int xfac = 0;  
    xfac = factorial(x);  
}
```

Answer: 4<sup>th</sup>  
thing returned  
is 24

# Factorial!

## Iterative version

```
int factorial(int n) {  
    int f = 1;  
    while (n > 1) {  
        f = f * n;  
        n = n - 1;  
    }  
    return f;  
}
```

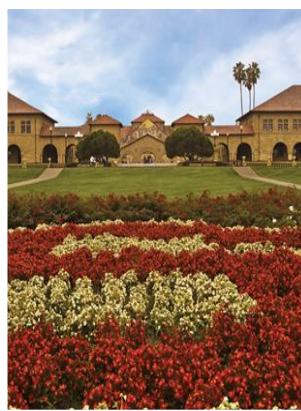
## Recursive version

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

NOTE: sometimes **iterative** can be **much faster** because it doesn't have to push and pop stack frames. Method calls have overhead in terms of space *and* time (to set up and tear down).

## How do we measure “faster” in Computer Science?

NOT AS SIMPLE AS YOU MIGHT  
THINK...



**Recall our discussion of performance  
with the Vector add vs. Insert...**

## Your turn: Vector performance

- **Answer: (D) Something else! (about 50x)**
  - › In addition to analyzing the code and predicting number of writes needed, we can also time the code using our Stanford 106B test system.
  - › **Check the code bundle for class today for runnable version!**

```
void runInsert(int size)
{
    Vector<int> v;
    for (int i = 0; i < size; i++) {
        v.insert(0, i);
    }
}

void runAdd(int size)
{
    Vector<int> v;
    for (int i = 0; i < size; i++) {
        v.add(i);
    }
}
```

```
/* * * * * Test Cases * * * * */
PROVIDED_TEST("Timing comparison")
{
    int size = 500000;
    TIME_OPERATION(size, runInsert(size));
    TIME_OPERATION(size, runAdd(size));
}
```

# Your turn: Vector performance

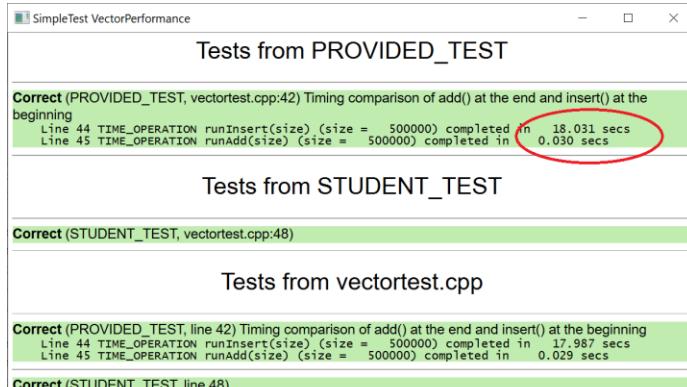
- **Answer: (D) Something else! (about 50x)**

- › In addition to analyzing the code and predicting number of writes needed, we can also time the code using our Stanford 106B test system.
- › **Check the code bundle for class today for runnable version!**

```
void runInsert(int size)
{
    Vector<int> v;
    for (int i = 0; i < size; i++) {
        v.insert(0, i);
    }
}

void runAdd(int size)
{
    Vector<int> v;
    for (int i = 0; i < size; i++) {
        v.add(i);
    }
}
```

```
/* * * * * Test Cases * * * * */
PROVIDED_TEST("Timing comparison")
{
    int size = 500000;
    TIME_OPERATION(size, runInsert(size));
    TIME_OPERATION(size, runAdd(size));
}
```



```
SimpleTest VectorPerformance
Tests from PROVIDED_TEST
Correct (PROVIDED_TEST, vectortest.cpp:42) Timing comparison of add() at the end and insert() at the beginning
Line 44 TIME_OPERATION runInsert(size), (size = 500000) completed in 18.031 secs
Line 45 TIME_OPERATION runAdd(size) (size = 500000) completed in 0.030 secs

Tests from STUDENT_TEST
Correct (STUDENT_TEST, vectortest.cpp:48)

Tests from vectortest.cpp
Correct (PROVIDED_TEST, line 42) Timing comparison of add() at the end and insert() at the beginning
Line 44 TIME_OPERATION runInsert(size), (size = 500000) completed in 17.987 secs
Line 45 TIME_OPERATION runAdd(size) (size = 500000) completed in 0.029 secs

Correct (STUDENT_TEST, line 48)
```

## Your turn: Vector performance

- **Answer: (D) Something else! (about 50x)**

- › Number of times a number is written in a box:

- OPTION 1:

- First loop iteration: 1 write
      - Next loop iteration: 2 writes ... continued...
      - Formula for sum of numbers 1 to N =  $(N * (N + 1)) / 2$
      - *(don't worry if you don't know this formula, we only expected a ballpark estimate)*
      - $100 * (100 + 1) / 2 = 10,100 / 2 = \text{5,050}$

- OPTION 2:

- First loop iteration: 1 write
      - Next loop iteration: 1 write ... continued...
      - **100**

## Big-O

- Big-O analysis in computer science is a way of counting the number of “steps” needed to complete a task
  - › Doesn’t really consider how “big” each step is
  - › Doesn’t consider how fast the computer’s CPU or other hardware components are
  - › Doesn’t involve any actual measurement of the time elapsed for any real code in any way
- But despite all that, really useful for making broad comparisons between different approaches

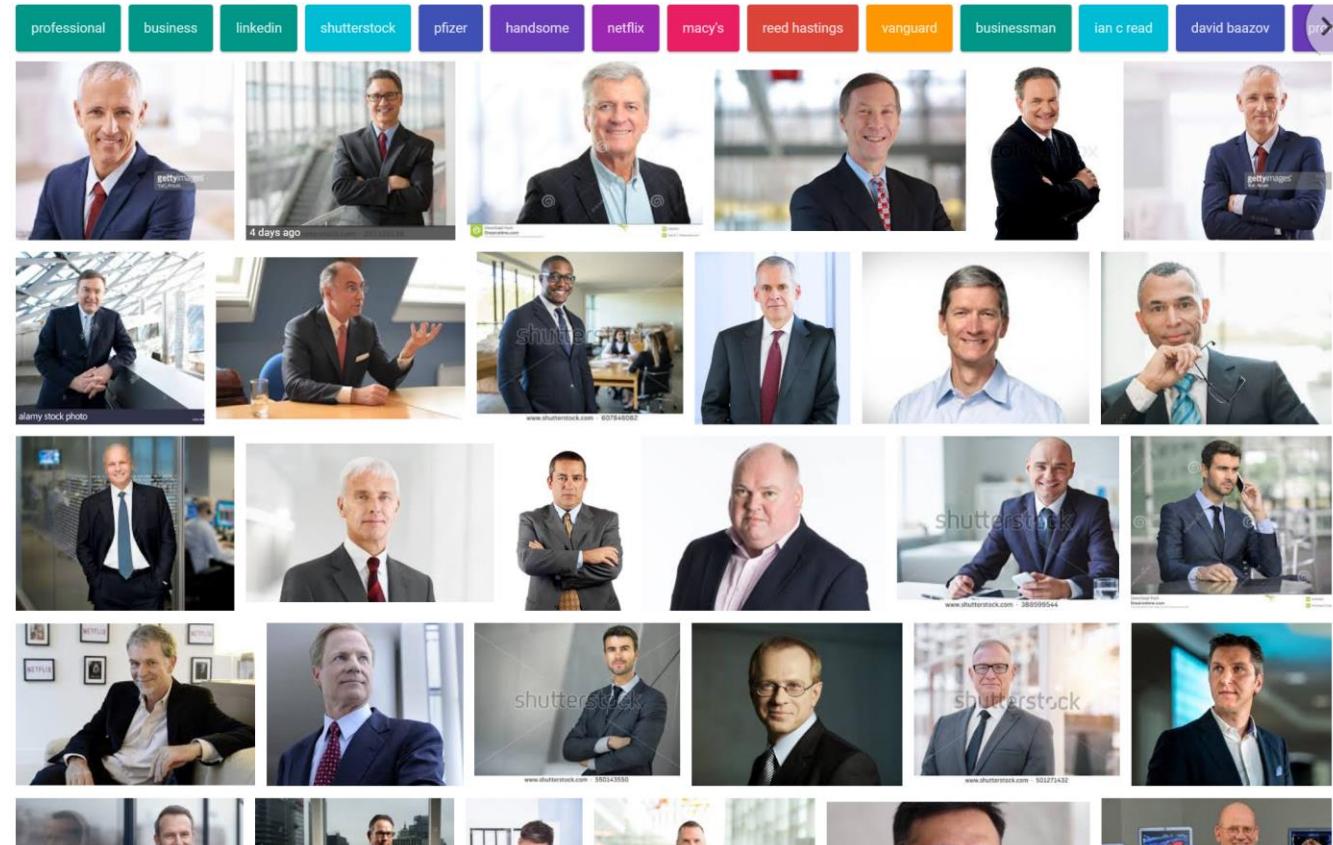
## Efficiency as a virtue?

- In computer science, we tend to obsess about **efficiency**, but it's worth taking a step back and asking ourselves, is efficiency always a virtue?
  - › Racing to be first to the finish line, but with an answer that's wrong, isn't helpful!
  - › That might seem obvious, but it happens \*all the time\* in real tech products

# Google image search



All Images News Videos Shopping More Settings Tools View saved SafeSearch ▾



# Another example...

Google professor

All Images News Videos Books More Settings Tools View saved SafeSearch

hot female android male baby african american indian chinese japanese university college classroom lab concord hospital car

The image grid contains 18 items:

- Row 1: hot female (highlighted), android, male, baby, african american, indian, chinese, japanese, university, college, classroom, lab, concord hospital, car.
- Row 2: Man in suit, Man in suit, Man in blue shirt, Man in suit, Man in suit.
- Row 3: Man in suit, Man in suit, Man with glasses, Man in suit, Man in suit, Man in lab coat, Woman in vest.
- Row 4: Man in suit, Man in tuxedo, Man in suit, Man in suit, Man in suit, Man in suit.

Stanford University

# The danger of a cheap solution: Twitter cropping

- In the summer of 2020, Twitter users noticed something strange about Twitter's new photo cropping algorithm
- Given a too-tall image, it selects which part to show
- It picked the Senator McConnell (the white man), not President Obama



Maybe it just chooses the top of the photo?



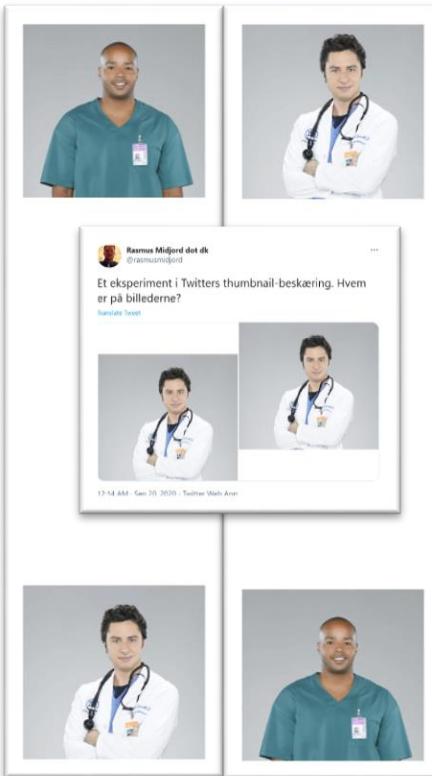
# The danger of a cheap solution: Twitter cropping

- In the summer of 2020, Twitter users noticed something strange about Twitter's new photo cropping algorithm
- Given a too-tall image, it selects which part to show
- It picked the Senator McConnell (the white man), not President Obama



Nope! It still happens when Obama is on top!





Stanford University

## Efficiency as a virtue?

- In each of these cases, companies chose an algorithm that would be most *efficient*, but came up with answers that were “wrong” (problematic) in ways that are significant for society
- How can we balance cost (which is what efficiency is really about in capitalism) with correctness and justice for society?
  - › Reflect on this in your Assignment 2!

# Programming Abstractions

CS106B

Cynthia Bailey Lee  
Julie Zelenski

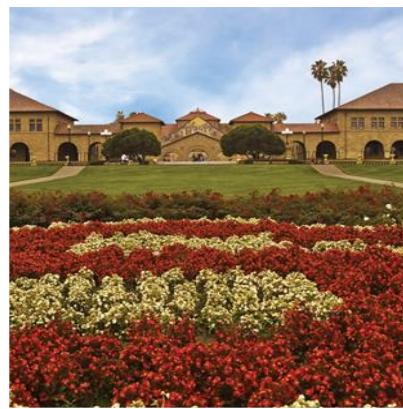
# Today's Topics

More ADTs!

- Map
  - › Code example: counting words in text
- Containers-within-containers
  - › Shallow copy vs. deep copy

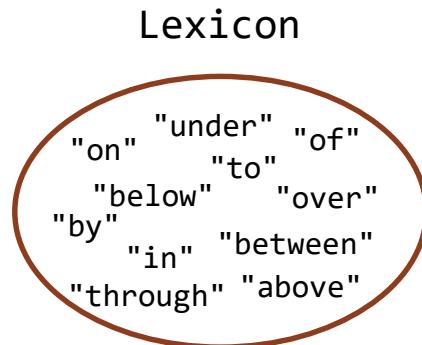
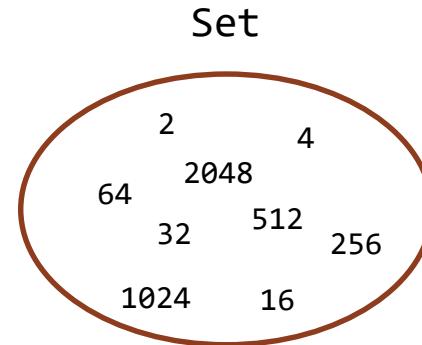
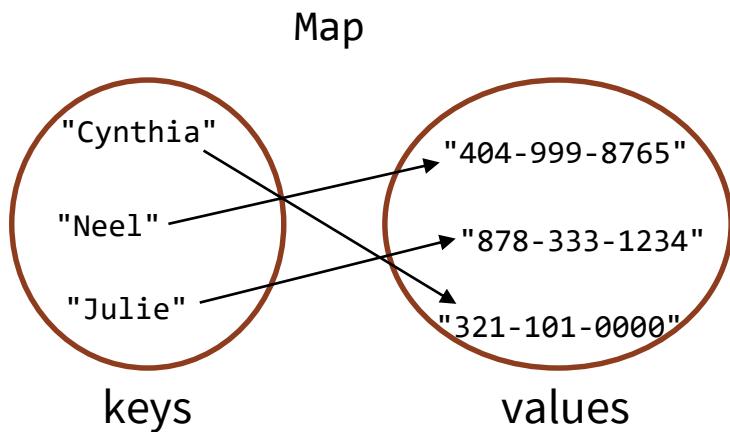
# Maps

(not like the driving  
directions kind of  
maps though)



# Associative containers

- Map
  - Set
  - Lexicon



**Not as concerned with order but with association**

- Map: associates **keys** with **values** (each could be any type)
  - Set: associates **keys** with **membership** (in or out)
    - › Lexicon: a set of strings, *with special internal optimizations for that*

# Stanford library Map (*selected member functions*)

```
void put(KeyType& key, ValueType& value);
bool containsKey(KeyType& key);
ValueType get(KeyType& key);
ValueType operator [](KeyType& key);

#include "map.h"

Map<string, string> phone;                      // Map takes two(!) template parameters

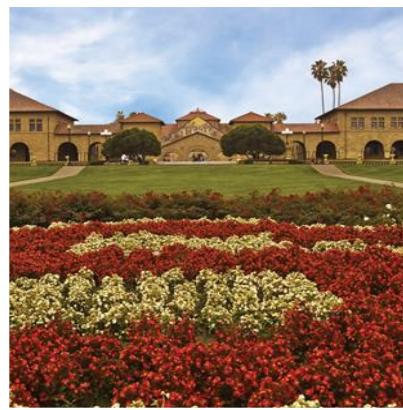
phone["Cynthia"] = "321-101-0000";                // two syntax options for adding new item
phone.put("Julie", "878-333-1234");

if (phone.containsKey("Cynthia") && phone.containsKey("Julie")) {
    cout << phone["Cynthia"] << endl;        // two syntax options for getting item
    cout << phone.get("Julie") << endl;
    cout << phone["MTL"] << endl;            // what would this do??
}
```



## Map Code Example

*Tabulating word  
counts*



# Map programming exercise

Write a program to count the number of occurrences of each unique word in a text file (e.g. *Poker* by Zora Neale Hurston).

- **First do an initial report:**

- › Print all words that appeared in the book at least 100 times, in alphabetical order

- **Then go into interactive query mode:**

- › The user types a word and we report *how many times* that word appeared in the book (repeat in a loop until quit).

# Map programming exercise

Write a program to count the number of occurrences of each unique word in a text file (e.g. *Poker* by Zora Neale Hurston).

- The user types a word and we report *how many times* that word appeared in the book (repeat in a loop until quit).

## What would be a good design for this problem?

- A. `Map<int, string> wordCounts;`
- B. `Map<Vector<string>, Vector<int>> wordCounts;`
- C. `Map<Vector<int>, Vector<string>> wordCounts;`
- D. `Map<string, int> wordCounts;`
- E. `Map<string, Vector<int>> wordCounts;`
- F. Other/none/more

Write a program to count the number of occurrences of each unique word in a text file (e.g. *Poker* by Zora Neale Hurston).

## How can we record the count?

(In other words, what goes in the place marked “record count here” in the code at right?)

- A. `wordCounts[word] += word;`
- B. `wordCounts[word] += 1;`
- C. `wordCounts[word]++;`
- D. B and C are good, but you need to first detect new (never seen before) words so you can start at zero before you start adding +1
- E. Other/none/more

```
// We are given a vector that is just the
// the book, broken into pieces based on
// spaces between words. The type is:
// Vector<string> words;

Map<string, int> wordCounts;
for (string word : words) {
    // record count here
}
```

Write a program to count the number of occurrences of each unique word in a text file (e.g. *Poker* by Zora Neale Hurston).

- The user types a word and we report *how many times* that word appeared in the book (repeat in a loop until quit).

```
// userWord is a word the user typed into the console  
cout << userWord << " appears " << wordCounts[userWord] << " times" << endl;
```

### What happens if queryWord is not a word in the book?

- Will the program crash?
- What other issue(s) besides crash do you foresee?



Write a program to count the number of occurrences of each unique word in a text file (e.g. *Poker* by Zora Neale Hurston).

- Report all words that appeared in the book at least 100 times, in alphabetical order

```
for (string word : wordCounts) {  
    if (wordCounts[word] >= FREQUENCY_THRESHOLD) {  
        cout << word << "\t" << wordCounts[word] << endl;  
    }  
}
```

**Does this work for our alphabetical order requirement?**

- Yes!
- Stanford library Map returns its keys in sorted order



## Compound Containers

*Containers within containers  
within containers!  
It's turtles all the way  
down...*



Stanford University

# Can we add the number 4 to a Vector? Let's see...

```
Vector<int> numbers;  
numbers.add(1);  
numbers.add(2);  
numbers.add(3);  
Map<string, Vector<int>> mymap;  
mymap["abc"] = numbers;  
// Now we want to add 4 to the Vector inside the Map, how can we do it?
```

```
numbers.add(4);
```

```
mymap["abc"].add(4);
```

```
Vector<int> test = mymap["abc"];  
test.add(4);
```

Would any of these three options work if inserted here? Which one(s)? Why or why not?

```
// GOAL: we want this to print 4 (indicating the add(4) worked)  
cout << "New size: " << mymap["abc"].size() << endl;
```

# Can we add the number 4 to a Vector? Let's see...

You don't need to worry too much about the details of how the cases differ in terms of behind-the-scenes mechanism—I just wanted to flag it as a potential issue in case you accidentally encounter this in your code!

```
> mymap;
```

to the Vector inside the Map, how can we do it?

```
numbers.add(4);
```

```
mymap["abc"].add(4);
```

```
Vector<int> test = mymap["abc"];
test.add(4);
```

Would any of these three options work if inserted here? Which one(s)? Why or why not?

```
// GOAL: we want this to print 4 (indicating the add(4) worked)
cout << "New size: " << mymap["abc"].size() << endl;
```

# Programming Abstractions

CS106B

Cynthia Bailey Lee  
Julie Zelenski

# Today's Topics

## Abstract Data Types

- Last time: What is an ADT? And two ADTs: Vector, Grid
- This time: More ADTs!
  - › Stack
  - › Queue
  - › Application of Stack
- Announcements:
  - › Assignment 1 due Friday
  - › *Do not use STL classes like vector, map, etc. for assignment 1 (or ever in this class). We also strongly recommend against using any Stanford library ADTs (except Vector, as directed). We carefully design these assignments to exercise certain skills learned in the class up to the date the assignment is released, so trust us, (a) you don't need them, (b) if you think you need them then you are missing a nice clean solution that takes a different approach (think topics we did cover in week 1, such as strings).*

CS106B COURSE ADMIN RESOURCES LECTURES ASSIGNMENTS SECTIONS ASSESSMENT

 CS106I

Fall Quarter 2022  
Lecture MWF 11

ANNOUNCEMENTS

What's happening this week  
Last updated yesterday by Neal

Lectures

- Monday, September 27th: Vecto

LaIR  
Ed Discussion Forum  
Paperless  
Qt Installation Guide  
**C++ Reference**  
**Stanford Library Documentation**  
Style Guide  
Testing Guide  
Submission Checklist  
Textbook

## Abstractions

# Stacks

**PRO TIP:** TO VIEW DETAILS OF ANY OF OUR STANFORD LIBRARY IMPLEMENTATIONS OF ADTS, GO TO THE COURSE WEBSITE, RESOURCES TAB, STANFORD LIBRARY REFERENCE



# New ADT: Stack

```
#include "stack.h"

Stack<string> recentCalls;
recentCalls.push("Neel");

recentCalls.push("Julie");

recentCalls.push("Esteban");

recentCalls.push("Minh");

while (!recentCalls.isEmpty()) {
    cout << recentCalls.pop() << " ";
}
```



Source: <http://www.flickr.com/photos/35237093334@N01/409465578/>  
Author: <http://www.flickr.com/people/35237093334@N01> Peter Kazanjy



# New ADT: Stack

```
#include "stack.h"
```

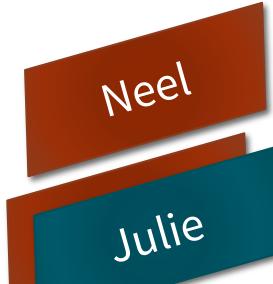
```
Stack<string> recentCalls;  
recentCalls.push("Neel");
```

```
recentCall:
```

“Why do I need Stack??  
I could have done that with a Vector!”  
—ADT skeptic

```
recentCall:
```

```
while (!recentCalls.isEmpty()) {  
    cout << recentCalls.pop() << " ";  
}
```



# Stack and Vector, side-by-side



```
Stack<string> recentCalls;
recentCalls.push("Neel");
recentCalls.push("Julie");
recentCalls.push("Esteban");
recentCalls.push("Minh");

while (!recentCalls.isEmpty()) {
    cout << recentCalls.pop() << " ";
}
```

0	1	2	3
Neel	Julie	Esteban	Minh

```
Vector<string> recentCalls;
recentCalls.add("Neel");
recentCalls.add("Julie");
recentCalls.add("Esteban");
recentCalls.add("Minh");

while (!recentCalls.isEmpty()) {
    string last = recentCalls[recentCalls.size() - 1];
    cout << last << " ";
    recentCalls.remove(recentCalls.size() - 1);
}
```

# Stack and Vector, side-by-side



```
Stack<string> recentCalls;
recentCalls.push("Neel");
recentCalls.push("Julie");
recentCalls.push("Esteban");
recentCalls.push("Minh");
```

```
while (!recentCalls.isEmpty()) {
    cout << recentCalls.pop() << " ";
}
```

```
Vector<string> recentCalls;
recentCalls.add("Neel");
recentCalls.add("Julie");
recentCalls.add("Esteban");
recentCalls.add("Minh");
```

```
while (!recentCalls.isEmpty()) {
    string last = recentCalls[recentCalls.size() - 1];
    cout << last << " ";
    recentCalls.remove(recentCalls.size() - 1);
```

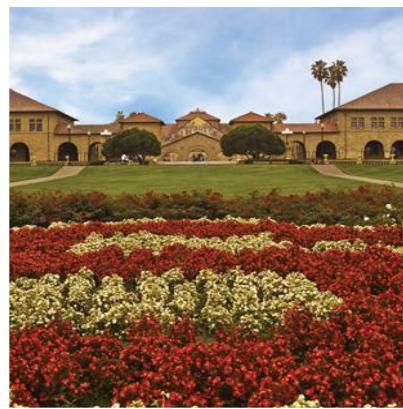
This Vector code isn't terrible, but it is harder to read quickly, and is probably more error prone.

- You need to think carefully about which end of the Vector to use as the top of the stack ( $0^{\text{th}}$  or  $\text{size}() - 1^{\text{th}}$ ), and performance impacts
- It would be easy to forget the “ $-1$ ” when you print/remove  $\text{size}() - 1^{\text{th}}$

0	1	2	3
Neel	Julie	Esteban	Minh

# Queues

FIFO – FIRST IN, FIRST OUT  
(OR “FIRST COME, FIRST SERVE”)



# Queues

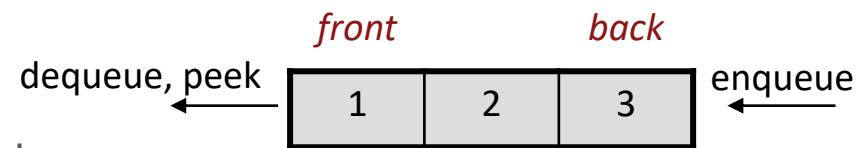
queue: First-In, First-Out ("FIFO")

- Elements stored in order they were added
- Can add only to the back, can only examine/remove frontmost element



queue operations

- enqueue: Add an element to the back
- dequeue: Remove the front element
- peek: Examine the front element



# The Queue class

```
#include "queue.h"
```

<code>q.dequeue()</code>	removes <b>front</b> value and returns it; throws an error if queue is empty
<code>q.enqueue(value)</code>	places given value at <b>back</b> of queue
<code>q.isEmpty()</code>	returns true if queue has no elements
<code>q.peek()</code>	returns <b>front</b> value without removing; throws an error if queue is empty
<code>q.size()</code>	returns number of elements in queue

- `Queue<int> q;` // {} front -> back
- `q.enqueue(42);` // {42}
- `q.enqueue(-3);` // {42, -3}
- `q.enqueue(17);` // {42, -3, 17}
- `cout << q.dequeue() << endl;` // 42 (q is {-3, 17})
- `cout << q.peek() << endl;` // -3 (q is {-3, 17})
- `cout << q.dequeue() << endl;` // -3 (q is {17})

## Application of Stacks

WE'VE SEEN ONE (BUFFERING INPUT AND GIVING IT BACK IN REVERSE—LIFO—ORDER). WHAT ELSE ARE STACKS GOOD FOR?



# Operator Precedence and Syntax Trees

Ignoring operator precedence rules, how many distinct results are there to the following arithmetic expression?

- $3 * 3 + 3 * 3$

# Reverse Polish Notation

Ambiguities don't exist in RPN

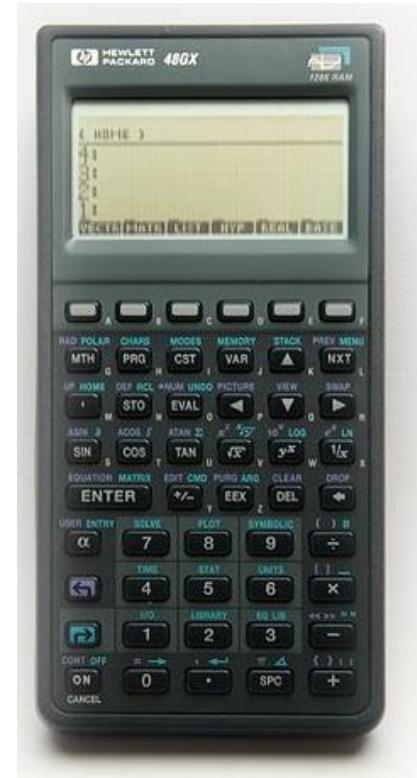
Also called “postfix” because the operator goes after the operands

Postfix (RPN):

- $4 \ 3 \ * \ 4 \ 3 \ * \ +$

Equivalent Infix:

- $(4 * 3) + (4 * 3)$

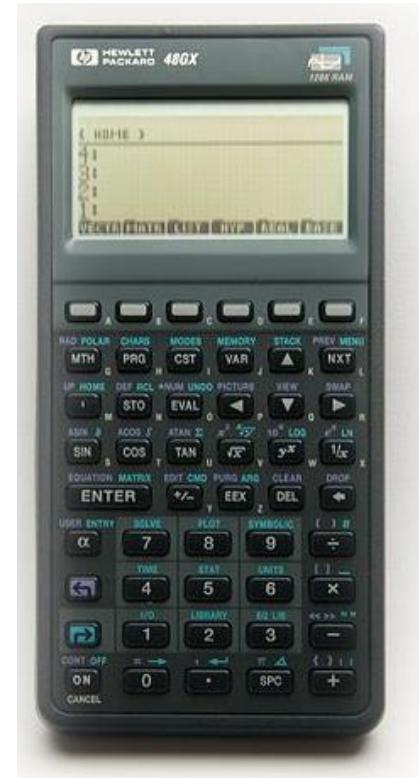


[http://commons.wikimedia.org/wiki/File:Hewlett-Packard\\_48GX\\_Scientific\\_Graphing\\_Calculator.jpg](http://commons.wikimedia.org/wiki/File:Hewlett-Packard_48GX_Scientific_Graphing_Calculator.jpg)

# Reverse Polish Notation



#TBT: Me in 1991, I was 12 years old



[https://commons.wikimedia.org/wiki/File:Hewlett-Packard\\_48GX\\_Scientific\\_Graphing\\_Calculator.jpg](https://commons.wikimedia.org/wiki/File:Hewlett-Packard_48GX_Scientific_Graphing_Calculator.jpg)

Stanford University

# Reverse Polish Notation

This postfix expression:

- $4 \ 3 \ * \ 7 \ 2 \ 5 \ * \ + \ +$

Is equivalent to this infix expression:

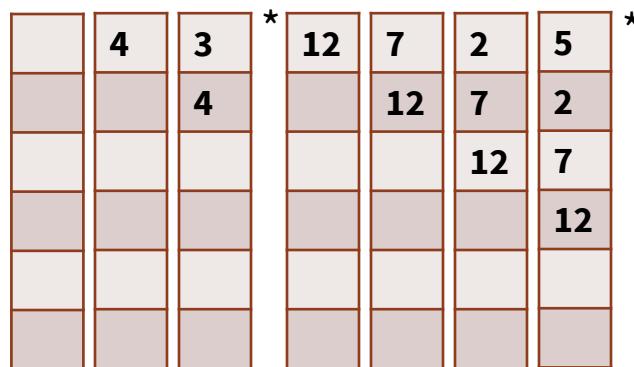
- $((4*3) + (7*2)) + 5$
- $(4*3) + ((7+2) + 5)$
- $(4*3) + (7 + (2*5))$
- Other/none/more than one



[http://commons.wikimedia.org/wiki/File:Hewlett-Packard\\_48GX\\_Scientific\\_Graphing\\_Calculator.jpg](http://commons.wikimedia.org/wiki/File:Hewlett-Packard_48GX_Scientific_Graphing_Calculator.jpg)

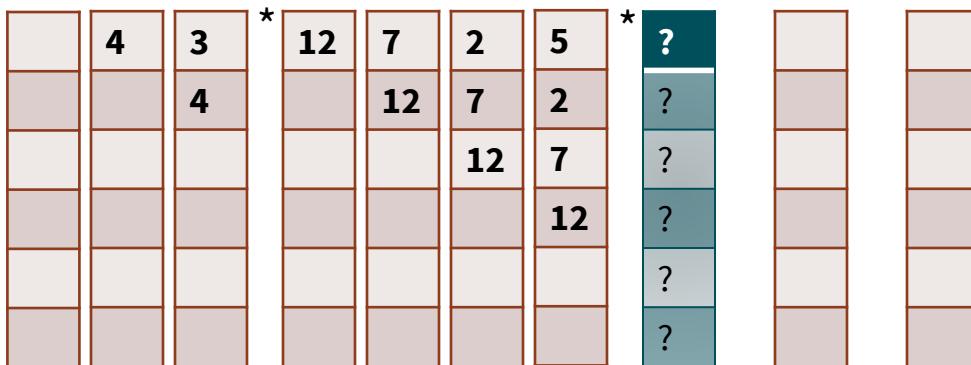
# Stacks and RPN

- Evaluate this expression with the help of a stack
  - Encounter a **number**? **PUSH** it
  - Encounter an **operator**? **POP** two numbers and **PUSH** result
- 4 3 \* 7 2 5 \* ++



# Stacks and RPN

- Evaluate this expression with the help of a stack
    - › Encounter a **number**? **PUSH** it
    - › Encounter an **operator**? **POP** two numbers and **PUSH** result
  - 4 3 \* 7 2 5 \* + +

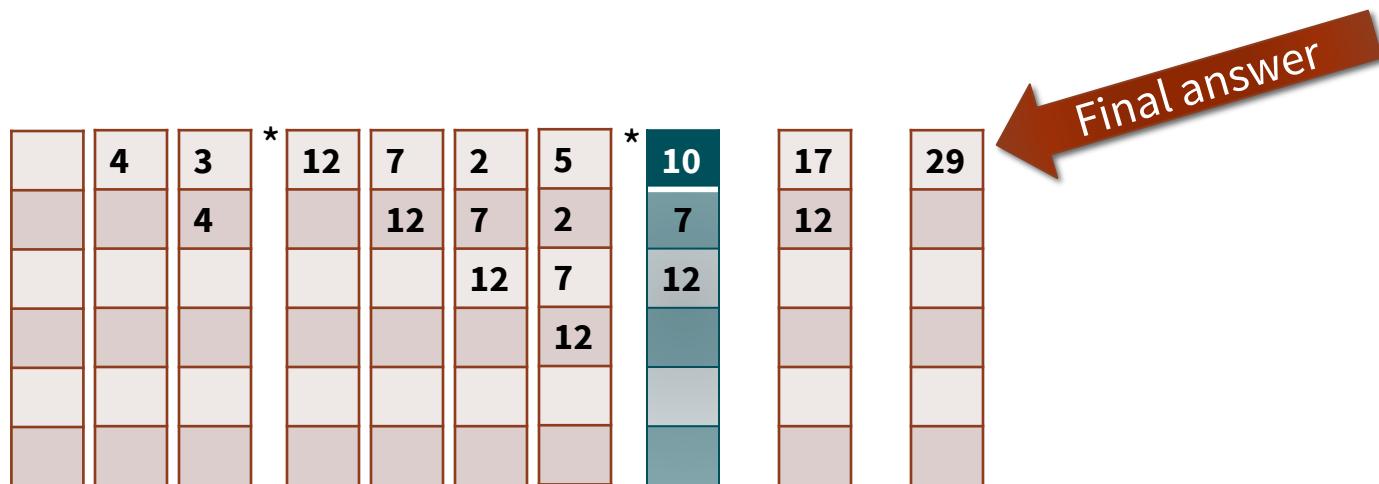


## Contents of the stack, reading from top down:

- (A) 7, 12
  - (B) 10, 7, 12
  - (C) 10, 5, 2, 7, 12
  - (D) Other

# Stacks and RPN

- Evaluate this expression with the help of a stack
  - Encounter a **number**? **PUSH** it
  - Encounter an **operator**? **POP** two numbers and **PUSH** result
- 4 3 \* 7 2 5 \* ++



- Question:** what are some signs that an expression is badly formatted?

# Final code of parser

```
bool calculate(const string& expression, int& result)
{
    Stack<int> memory;
    for (size_t i = 0; i < expression.length(); i++) {
        if (isdigit(expression[i])) {
            int value = charToInteger(expression[i]);
            memory.push(value);
        } else if (isSupportedOperator(expression[i]) && memory.size() >= 2) {
            int rhs = memory.pop(); // right-hand-side operand is 1st to pop
            int lhs = memory.pop(); // left-hand-side operand is 2nd to pop
            memory.push(applyOperator(lhs, expression[i], rhs));
        } else {
            // parse error on anything other than digit or operator
            return false;
        }
    }
    // final parse validity check
    if (memory.size() != 1) {
        return false;
    }
    result = memory.pop();
    return true;
}
```

# Programming Abstractions

CS106B

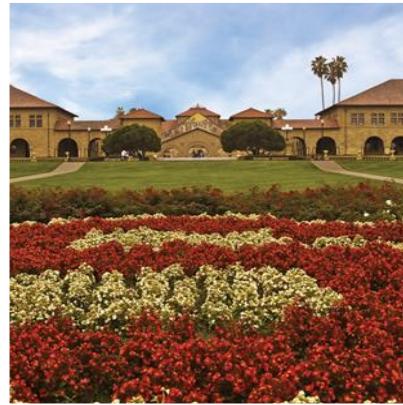
Cynthia Lee

# Today's Topics

## Abstract Data Types

- What is an ADT?
- Vector ADT
- Grid ADT
- *Next time:* Stack, Queue ADTs

**ADTs**



Stanford University

# ADTs = “Abstract Data Types”

- **Language-independent models of common containers**
  - › In other words, we try to focus on the aspects of the ADT that transcend whether we happen to be using it in C++, Java, Python, or some other language
- ADTs encompass both the nature of the data and ways of accessing it
- ADTs form a rich **vocabulary** of **nouns** (nature of the data) and **verbs** (ways of accessing it), often drawing on analogies to make their use intuitive
  - › Skillful ADT use gives code added readability!

# Types of ADTs

- When we say the “nature of the data,” we mean questions like:
  - › Is the data **ordered** in some way?
    - Could/should you be able to say about the data that this element is the “first” one, and this other piece is the “tenth” one?
  - › Is the data **paired or matched** in some way?
    - Could/should you be able to say about the data that this element A goes with element B (not D), and this element C goes with element D (not B)?
- When we say “ways of accessing it,” we mean questions like:
  - › Is it important to be **able to add and remove data** during the course of use, or do we assume we have the “final” version from the beginning?
  - › Is it important to be able to **search for any piece of data** in the collection, or is it enough to always take the first available one?

# Types of ADTs

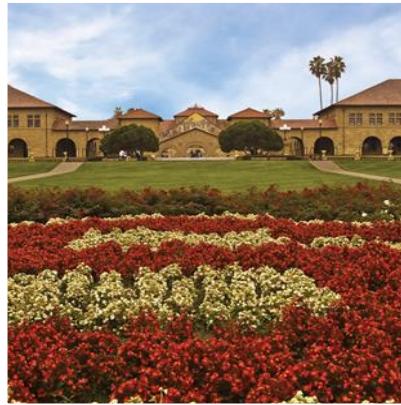
- When we say the “nature of the data,” we mean questions like:
  - › Is the data **ordered** in some way?
    - Could/should you be able to say about the data that this element is the “first” one, and this other piece is the “tenth” one?
  - › Is the data **paired or matched** in some way?
    - Could/should you be able to say about the data that this element A goes with element B (not D), and this element C goes with element D (not B)?
- When we say “ways of accessing it,” we mean questions like:
  - › Is it important to be **able to add and remove data** during the course of use, or do we assume we have the “final” version from the beginning?
  - › Is it important to be able to **search for any piece of data** in the collection, or is it enough to always take the first available one?

We'll talk about ADTs in this category today and Wednesday.

We'll talk about ADTs in this category on Friday.

# Vector

OUR FIRST ADT!



# Vector ADT

- ADT abstraction similar to an array or list
- You're probably thinking, "Hey, there was something like that in the language I studied before!"
  - › This shouldn't be a surprise—remember that ADTs are defined as conceptual abstractions that are language-independent
- We will use **Stanford** library Vector (there is also an C++ STL vector, which will not use—watch out for capitalization!)

# Stanford Library Vector

- We declare one like this:
  - › `#include "vector.h" // note quotes to mean Stanford version`
  - › `Vector<string> lines; // note uppercase V here`
- This syntax is called **template** syntax
  - › In C++, template containers must be **homogenous** (*all items the same type*)
  - › The type goes in the <> after the class name Vector

```
// Example: initialize a vector containing 5 integers
Vector<int> nums {42, 17, -6, 0, 28};
```

<i>index</i>	0	1	2	3	4
<i>value</i>	42	17	-6	0	28

# Vector

- Examples of declaring a Vector:

- › `Vector<int> pset3Scores;`
- › `Vector<double> measurementsData;`
- › `Vector<Vector<int>> allAssignmentScores;`

- Examples of using a Vector:

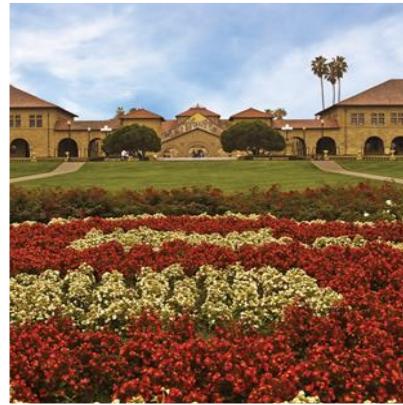
- › `pset3Scores.add(98);`
- › `pset3Scores.add(85);`
- › `pset3Scores.add(92);`
- › `cout << pset3Scores[0] << endl; // prints 98`
- › `cout << pset3Scores[pset3Scores.size() - 1] << endl; // prints 92`
- › `allAssignmentScores.add(pset3Scores);`
- › `cout << allAssignmentScores[0][1] << endl; // prints 85`



More on 2-D Vectors in a moment, with Grid ADT!

# Vector Performance

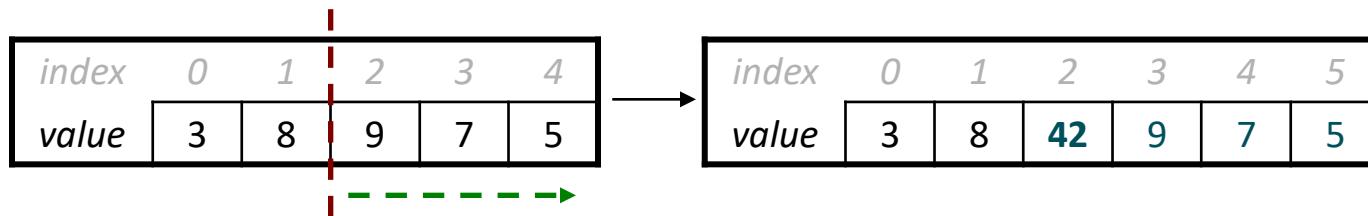
A LITTLE PEEK AT HOW  
VECTORS WORK BEHIND  
THE SCENES



# \*Performance Warning\* Vector insert/remove

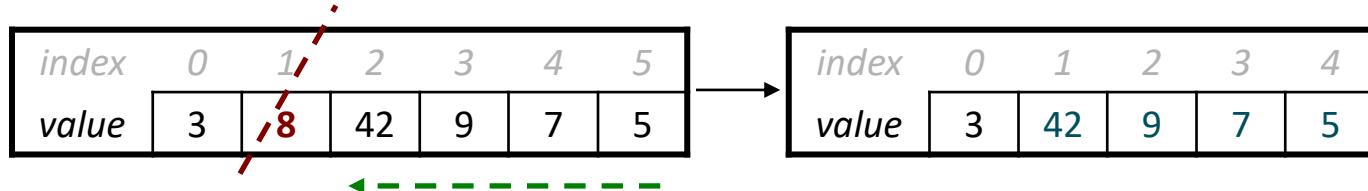
- **v.insert(2, 42)**

- › shift elements right to make room for the new element



- **v.remove(1)**

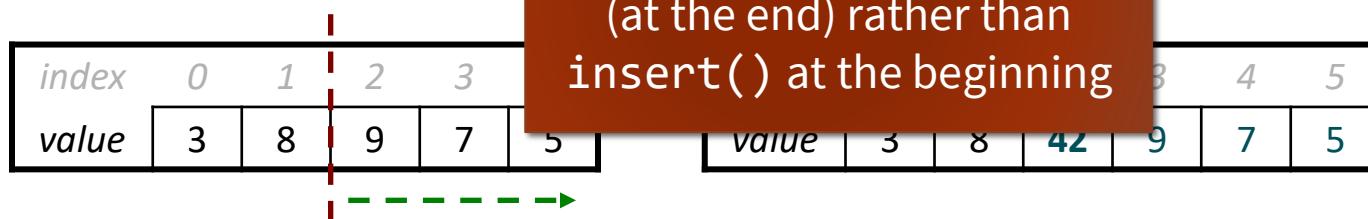
- › shift elements left to cover the space left by the removed element



- These operations are slower the more elements they need to shift

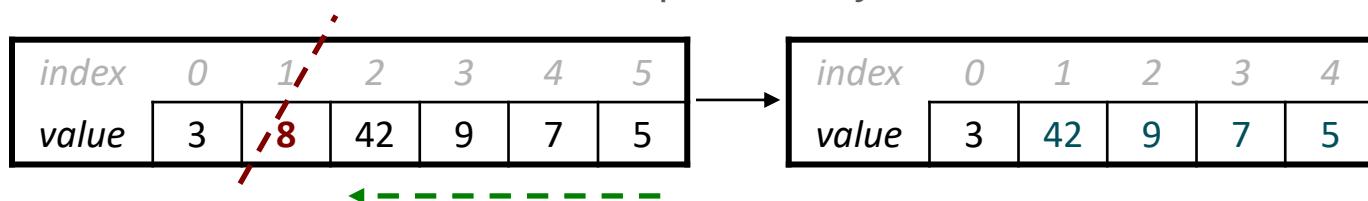
# \*Performance Warning\* Vector insert/remove

- **v.insert(2, 42)**
  - › shift elements right to make space



Pro tip: if possible in your situation, try to use `add()` (at the end) rather than `insert()` at the beginning

- **v.remove(1)**
  - › shift elements left to cover the space left by the removed element



- These operations are **slower** the more elements they need to shift

## Your turn: Vector performance

- **Warm-up question:** tell a neighbor what the contents of the vector look like at the end of each of OPTION 1 and at the end of OPTION 2. (As shown, `v` starts out empty in both cases)

```
Vector<int> v;  
for (int i = 0; i < 100; i++) {  
    v.insert(0, i); // OPTION 1  
}
```

```
Vector<int> v;  
for (int i = 0; i < 100; i++) {  
    v.add(i); // OPTION 2  
}
```

index	0	1	2	3	4	...
value	99	98	97	96	95	...

index	0	1	2	3	4	...
value	0	1	2	3	4	...

## Your turn: Vector performance

- Compare how many times we write a number into one “box” of the Vector, in these two codes. Write can be the original write, or because it had to move over one place. (As shown, v starts out empty in both cases)

```
Vector<int> v;  
for (int i = 0; i < 100; i++) {  
    v.insert(0, i); // OPTION 1  
}
```

```
Vector<int> v;  
for (int i = 0; i < 100; i++) {  
    v.add(i); // OPTION 2  
}
```

- A. They both write in a box about the same number of times
- B. One writes about 2x as many times as the other
- C. One writes about 5x as many times as the other
- D. Something else!

Answer now on [pollev.com/cs106b](http://pollev.com/cs106b) !

Since B and C don't say which option writes more than the other, if you pick one of those, be sure to address that in your group discussion!

# Your turn: Vector performance

- **Answer: (D) Something else! (about 50x)**

- › In addition to analyzing the code and predicting number of writes needed, we can also time the code using our Stanford 106B test system.
- › **Check the code bundle for class today for runnable version!**

```
void runInsert(int size)
{
    Vector<int> v;
    for (int i = 0; i < size; i++) {
        v.insert(0, i);
    }
}

void runAdd(int size)
{
    Vector<int> v;
    for (int i = 0; i < size; i++) {
        v.add(i);
    }
}
```

```
/* * * * * Test Cases * * * * */
PROVIDED_TEST("Timing comparison")
{
    int size = 500000;
    TIME_OPERATION(size, runInsert(size));
    TIME_OPERATION(size, runAdd(size));
}
```

# Your turn: Vector performance

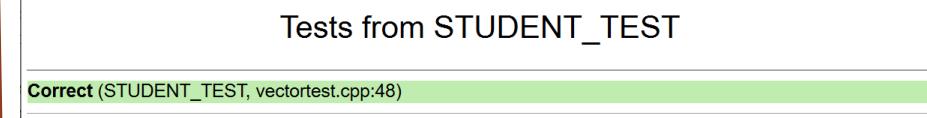
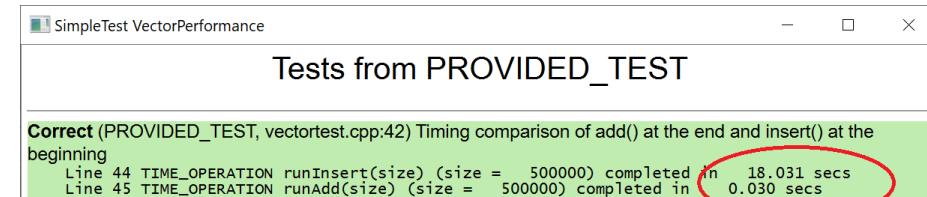
- **Answer: (D) Something else! (about 50x)**

- › In addition to analyzing the code and predicting number of writes needed, we can also time the code using our Stanford 106B test system.
- › **Check the code bundle for class today for runnable version!**

```
void runInsert(int size)
{
    Vector<int> v;
    for (int i = 0; i < size; i++) {
        v.insert(0, i);
    }
}

void runAdd(int size)
{
    Vector<int> v;
    for (int i = 0; i < size; i++) {
        v.add(i);
    }
}
```

```
/* * * * * Test Cases * * * * */
PROVIDED_TEST("Timing comparison")
{
    int size = 500000;
    TIME_OPERATION(size, runInsert(size));
    TIME_OPERATION(size, runAdd(size));
}
```



# Your turn: Vector performance

- **Answer: (D) Something else! (about 50x)**

- › Number of times a number is written in a box:

- OPTION 1:

- First loop iteration: 1 write
      - Next loop iteration: 2 writes ... continued...
      - Formula for sum of numbers 1 to N =  $(N * (N + 1)) / 2$
      - *(don't worry if you don't know this formula, we only expected a ballpark estimate)*
      - $100 * (100 + 1) / 2 = 10,100 / 2 = \text{5,050}$

- OPTION 2:

- First loop iteration: 1 write
      - Next loop iteration: 1 write ... continued...
      - **100**

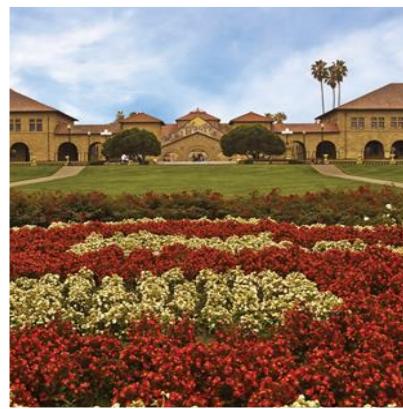
# Vector performance and parameter passing

- **Pro Tip:** always use pass-by-reference for containers like Vector (and Grid, which we'll see next) in this class!
  - › For efficiency reasons—don't want to make a big copy every time with pass-by-value!

```
void printFirst(Vector<int>& input) {  
    cout << input[0] << endl;  
}  
  
void printFirst100Times(Vector<int>& input) {  
    for (int i = 0; i < input.size(); i++) {  
        printFirst(input); // very expensive if not for &  
    }  
}
```

## Grid container

ESSENTIALLY A MATRIX  
(LINEAR ALGEBRA FANS  
CELEBRATE NOW)



# Grid

- ADT abstraction similar to an array of arrays (matrix)
- Many languages have a version of this
  - › (remember, ADTs are conceptual abstractions that are language-independent)
- In C++ we declare one like this:

```
#include "grid.h"

Grid<int> chessboard;
Grid<int> image;
Grid<double> realMatrix;
```

# Code Reading Exercise: Grids and loops and loop

```
void printMe(Grid<int>& grid, int row, int col) {  
    for (int r = row - 1; r <= row + 1; r++) {  
        for (int c = col - 1; c <= col + 1; c++) {  
            if (grid.inBounds(r, c)) {  
                cout << grid[r][c] << " ";  
            }  
        }  
        cout << endl;  
    }  
}
```

- How many 0's does this print with input `row = 2, col = 3?` (*and grid as shown on right*)
- (A) None or 1  
(B) 2 or 3  
(C) 4 or 5  
(D) 6 or 7

2	1	2	0	0
1	0	2	1	2
0	0	0	1	1
2	2	2	2	2
1	1	0	1	1

## Handy loop idiom: iterating over “neighbors” in a Grid

```
void printNeighbors(Grid<int>& grid, int row, int col) {  
    for (int r = row - 1; r <= row + 1; r++) {  
        for (int c = col - 1; c <= col + 1; c++) {  
            if (grid.inBounds(r, c)) {  
                cout << grid[r][c] << " ";  
            }  
        }  
        cout << endl;  
    }  
}
```

row - 1 col - 1	row - 1 col + 0	row - 1 col + 1
row + 0 col - 1	row col	row + 0 col + 1
row + 1 col - 1	row + 1 col + 0	row + 1 col + 1

These nested for loops generate all the pairs in the cross product  $\{-1,0,1\} \times \{-1,0,1\}$ , and we can add these as offsets to a  $(r,c)$  coordinate to generate all the neighbors (note: often want to test for and exclude the  $(0,0)$  offset, which is “myself” not a neighbor)

# Programming Abstractions

CS106B

Cynthia Bailey Lee  
Julie Zelenski

# Today's Topics

## Introducing C++

- Finish in-class string exercise
- Hamilton example (continued)
  - › Style, defining constants
  - › Testing
- Parameter passing in C++
  - › Pass by value semantics
  - › Pass by reference
  - › const
- TODO this week:
  - › Sign ups for section are open at [cs198.stanford.edu](https://cs198.stanford.edu). They will **close on Sunday**, September 26th at 5PM PT. Section meetings start week 2.
  - › **Assignment 0** is **due today, Friday**, September 24th at 11:59PM. There is a 48-hour grace period for assignment 0.
  - › Assignment 1 will go out today and be due in 1 week.

Go to  
pollev.com/cs106b  
to join class practice  
questions

Go to  
edstem.org  
to join live lecture  
Q&A with Julie

## C++ standard `string` object member functions (3.2)

```
#include <string>
```

Member function name	Description
<code>s.append(str)</code>	add text to the end of a string
<code>s.compare(str)</code>	return -1, 0, or 1 depending on relative ordering
<code>s.erase(index, Length)</code>	delete text from a string starting at given index
<code>s.find(str)</code> <code>s.rfind(str)</code>	first or last index where the start of <code>str</code> appears in this string ( <b>returns <code>string::npos</code> if not found</b> )
<code>s.insert(index, str)</code>	add text into a string at a given index
<code>s.length()</code> or <code>s.size()</code>	number of characters in this string
<code>s.replace(index, Len, str)</code>	replaces <code>len</code> chars at given index with new text
<code>s.substr(start, Length)</code> or <code>s.substr(start)</code>	the next <code>length</code> characters beginning at <code>start</code> (inclusive); if <code>length</code> omitted, grabs till end of string

Exercise: Write a line of code that pulls out the part of a string that is inside parentheses, assuming input variable `str` has the form "(blahblah)" where `blahblah` is any pattern of characters.

```
string insidePart = _____;
```

## Exercise solutions:

Exercise: Write a line of code that pulls out the part of a string that is inside parentheses, assuming variable `str` has the form "(blahblah)" where `blahblah` is any pattern of characters.

```
string insidePart = _____;
```

Go to  
[pollev.com/cs106b](http://pollev.com/cs106b)  
to join class practice  
questions

# Stanford library helpful string processing (*read3.7*)

```
#include "strlib.h"
```

- Unlike the previous ones, these take the string as a parameter.
  - › C++ string class example: str.substr(0, 2);
  - › Stanford string library example: endsWith(".jpg");
- That's because we here at Stanford wrote these functions, and they are not official C++ string class methods.

Function name	Description
endsWith( <i>str, suffix</i> ) startsWith( <i>str, prefix</i> )	returns true if the given string begins or ends with the given prefix/suffix text
integerToString( <i>int</i> ) realToString( <i>double</i> ) stringToInteger( <i>str</i> ) stringToReal( <i>str</i> )	returns a conversion between numbers and strings
equalsIgnoreCase( <i>s1, s2</i> )	true if s1 and s2 have same chars, ignoring casing
toLowerCase( <i>str</i> ) toUpperCase( <i>str</i> )	returns an upper/lowercase version of a string
trim( <i>str</i> )	returns string with surrounding whitespace removed

## **Hamilton Code (continued): Style and Testing**

**JUST AS IMPORTANT AS  
WRITING THE CODE IS  
WRITING IT WELL AND  
WRITING GOOD TESTS**



# Hamilton Code Style Notes

- Descriptive function and variable names
  - › Even someone who doesn't know code would have a pretty good idea what a function called "generate lyrics" does!
- Proper indentation
  - › *Even though C++ relies on the {} and not indentation (!)*
  - › Pro tip: in Qt Creator, select all then do CTRL - I (PC) or Cmd - I (Mac)
- One space between operators and variables
  - › Write `i < 3`, not `i<3`
  - › Coders were social distancing before it was cool
  - › Again, we do this even though C++ doesn't rely on it for parsing
- Define constants at the top of your file for any special values
  - › Example: `const int DAT_FREQ = 3;`
  - › Helps the reader understand what the value means or where it comes from
  - › If you use the value in several places, only need to change it in one place

# Writing Good Tests

- “Good” means thorough: covers all code paths and cases
- But don’t just add loads of tests for the sake of having many—each should have a purpose
- Be extra attentive to unusual circumstances
- These will vary, specific to the function you are testing, but common examples include:
  - › Integer inputs: negative numbers, zero, very large numbers
  - › String inputs: very short strings (length 0 or 1), very long strings

# Writing Good Tests



Brenan Keller  
@brenankeller

...

A QA engineer walks into a bar. Orders a beer. Orders 0 beers. Orders 999999999999 beers. Orders a lizard. Orders -1 beers. Orders a ueicbksjdhd.

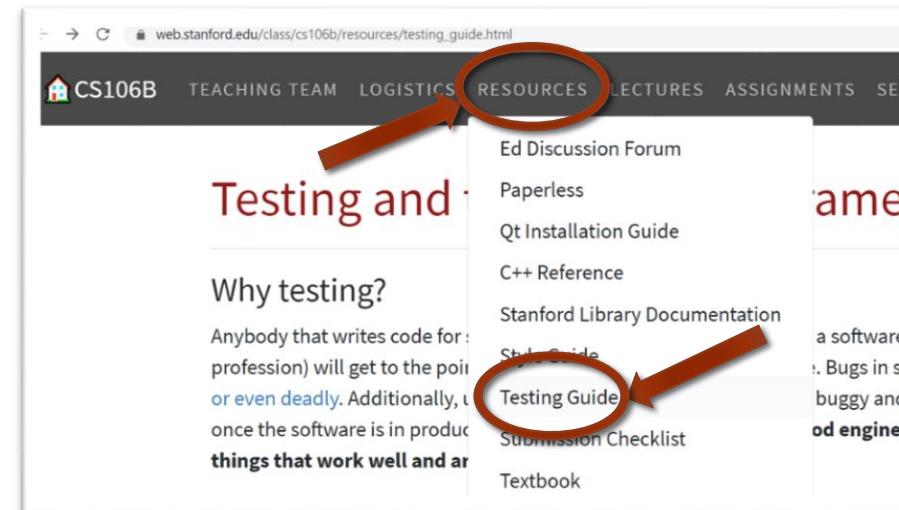
First real customer walks in and asks where the bathroom is. The bar bursts into flames, killing everyone.

1:21 PM · Nov 30, 2018 · Twitter for iPhone

- *A QA engineer is a software developer who specializes in writing tests and finding bugs in other engineers' code*

# CS106B Testing Framework

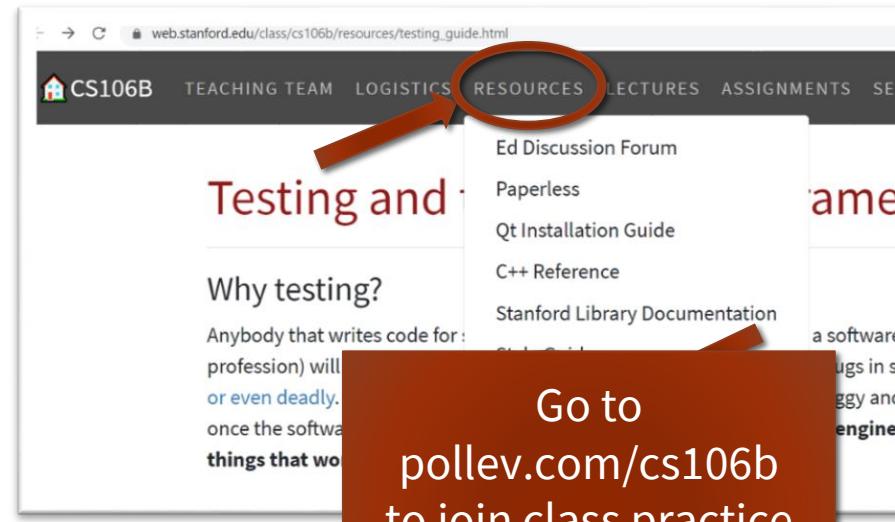
- We provide a framework for testing your code in this class
- More details on the website →



- **Quick version:**
- In `main()`, write:
  - › `runSimpleTests(SELECTED_TESTS);`
- Write tests as:
  - › `EXPECT_EQUAL(functionBeingTested(input), expectedOutput);`
  - › `EXPECT_EQUAL(generateLyrics(2), "Da Da ");`
- **Your Turn: What are some good test cases for our Hamilton code?**

# CS106B Testing Framework

- We provide a framework for testing your code in this class
- More details on the website →



- **Quick version:**
- In `main()`, write:
  - › `runSimpleTests(SELECTED_TESTS);`
- Write tests as:
  - › `EXPECT_EQUAL(functionBeingTested(input), expectedOutput);`
  - › `EXPECT_EQUAL(generateLyrics(2), "Da Da ");`
- **Your Turn: What are some good test cases for our Hamilton code?**

# C++ Parameter Passing

TWO PARADIGMS:  
PASS BY VALUE  
PASS BY REFERENCE



# "Pass by value"

(default behavior of parameters)

```
#include <iostream>
void foo(int n);

int main(){
    int n = 5;
    foo(n);
    cout << n << endl;
    return 0;
}

void foo(int n) {
    n++;
}
```

What is printed?

- A. 5
- B. 6
- C. Error or something else

# "Pass by reference"

```
#include <iostream>
void foo(int &num);

int main(){
    int n = 5;
    foo(n);
    cout << n << endl;
    return 0;
}

void foo(int &num) {
    num++;
}
```



- This one prints 6!
- I like to think of the & as a rope lasso that grabs the input parameter and drags it into the function call directly, rather than making a copy of its value and then leaving it in place.

# Your turn!

```
void mystery(int c, int& a, int b) {  
    cout << b << " + " << c << " = " << a << endl;  
    a++;  
    b--;  
}  
  
int main() {  
    int a = 4;  
    int b = 7;  
    int c = -2;  
  
    mystery(b, a, c);  
    mystery(c, b, 3);  
    mystery(b, c, b + a);  
    return 0;  
}
```

What does this print?

Go to

<https://pollev.com/cs106b>  
and send your answer!

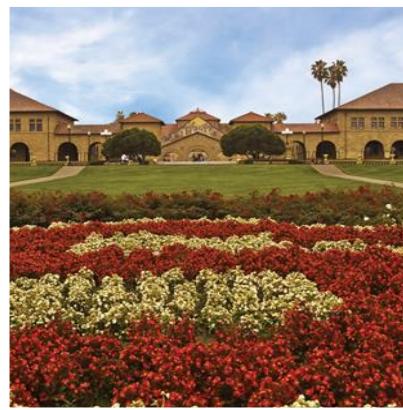
## Why though??

- We've looked at the *how* of pass-by-reference, but we haven't yet discussed the *why*.
- We'll see some examples of when this feature comes especially in handy next week when we learn about containers for data!

# Ethics in CS106B

ETHICAL DECISION-MAKING  
FRAMEWORKS

ETHICS OF STRINGS!



# Ethics in CS106B

- This will be a recurring series throughout the quarter, and will tie in to your homework assignments
- **What to watch for today:**
  - › **Meet your guide, Katie Creel!** Dr. Creel has degrees in computer science, moral philosophy, and history of science in society.
  - › Learn about some philosophical **frameworks for making ethical decisions**, which we will be a formal guide for our thinking throughout the quarter
  - › Get a preview of topics in **Assignment 1**
  - › Consider the **ethical implications of C++ variable types char and string**, which you just learned about
    - *That's right, even something as simple as strings has ethical concerns!*

# Ethics in CS106B

- This will be a recurring series throughout the quarter, and will tie in to your homework assignments
- **Free association question:**
  - › What words do you think of when you hear “tech ethics”?
  - › Write one word per submission (you may submit multiple times)

Go to  
[pollev.com/cs106b](https://pollev.com/cs106b)  
to join class practice  
questions

# Programming Abstractions

CS106B

Cynthia Bailey Lee  
Julie Zelenski

# Today's Topics

## Introducing C++

- Hamilton example
  - › In QT Creator (the IDE for our class)
  - › Function prototypes
  - › <iostream> and cout
  - › C++ characters and strings
  - › Testing
- TODO this week:
  - › Sign ups for section will be **open on Thursday**, September 23rd at 5PM PT at [cs198.stanford.edu](http://cs198.stanford.edu). They will **close on Sunday**, September 26th at 5PM PT. Section meetings start week 2.
  - › **Assignment 0** is **due Friday**, September 24th at 11:59PM.
  - › **Qt Installation Help Session** in Huang Engineering Basement on **Thursday**, September 23rd from 7PM-9PM

Go to  
pollev.com/cs106b  
to join class practice  
questions

Go to  
edstem.org/  
to join live lecture  
Q&A with Julie

# First C++ program (from Monday)

```
/*
 * hello.cpp
 * This program prints a welcome message
 * to the user.
 */
#include <iostream>
#include "console.h"
using namespace std;

int main() {
    cout << "Hello, world!" << endl;
    return 0;
}
```

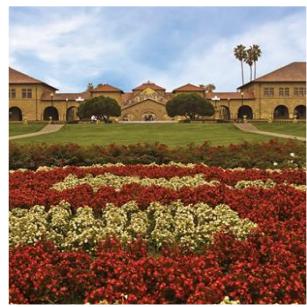
# C++ math functions (2.1)

```
#include <cmath>
```

Function name	Description (returns)
<code>abs(<i>value</i>)</code>	absolute value
<code>ceil(<i>value</i>)</code>	rounds up
<code>floor(<i>value</i>)</code>	rounds down
<code>log10(<i>value</i>)</code>	logarithm, base 10
<code>max(<i>value1</i>, <i>value2</i>)</code>	larger of two values
<code>min(<i>value1</i>, <i>value2</i>)</code>	smaller of two values
<code>pow(<i>base</i>, <i>exp</i>)</code>	<i>base</i> to the <i>exp</i> power
<code>round(<i>value</i>)</code>	nearest whole number
<code>sqrt(<i>value</i>)</code>	square root
<code>sin(<i>value</i>)</code> <code>cos(<i>value</i>)</code> <code>tan(<i>value</i>)</code>	sine/cosine/tangent of an angle in radians

# Live coding in Qt

HAMILTON KING GEORGE  
EXAMPLE



# Hamilton Code Demo: What essential skills did we just see?

- You must use function prototypes for your helper functions (if you want to keep **main** at the top, which is good style)
- You can write input/output with:
  - › **cout** (**<iostream>**)
- cout uses the **<<** operator
  - › Remember: the arrows point in the way the data is “flowing”
  - › These aren’t like HTML tags **<b></b>** or C++ parentheses () or curly braces {} in that they don’t need to “match”
- Good style: **const int** to make int constants
  - › (in demo, not previous slides)
  - › No “magic numbers”!
  - › Works for other types too (**const double**)

# Live Coding concept review

FUNCTION PROTOTYPES



# A simple C++ program (ERROR)

```
simple.cpp #include <iostream>
#include "console.h"
using namespace std;

int main() {
    myFunction(); // compiler is unhappy with this line
    return 0;
}

void myFunction() {
    cout << "myFunction!!" << endl;
}
```

# A simple C++ program (Fix option 1)

```
simple.cpp #include <iostream>
           #include "console.h"
           using namespace std;

           void myFunction() {
               cout << "myFunction!!" << endl;
           }

           int main() {
               myFunction(); // compiler is happy with this line now
               return 0;
           }
```

# A simple C++ program (Fix option 2)

```
simple.cpp #include <iostream>
#include "console.h"
using namespace std;

void myFunction(); // this is called a function prototype

int main() {
    myFunction(); // compiler is happy with this line now
    return 0;
}

void myFunction() {
    cout << "myFunction!!" << endl;
}
```

# A simple C++ program (Fix option 2)

```
simple.cpp      #include <iostream>
                #include "console.h"
                using namespace std;

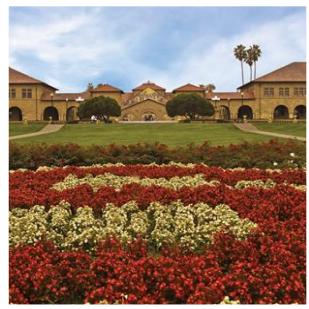
                void myFunction(); // this is called a function prototype

                int main() {
                    myFunction(); // compiler initially ok with this line...
                    return 0;
                }

                // ...but sad when it realizes it was tricked and you
                // never gave a definition of myFunction!!
```

# Live Coding concept review

STRINGS AND  
CHARACTERS IN C++



# Using cout and strings

```
int main(){
    string s = "ab";
    s = s + "cd";
    cout << s << endl;
    return 0;
}
```

```
int main(){
    string s = "ab" + "cd";
    cout << s << endl;
    return 0;
}
```

- This prints “abcd”
- The + operator concatenates strings in the way you’d expect.
- But...SURPRISE!...this one doesn’t work.

# String literals vs. C++ string objects

- In this class, we will interact with two types of strings:
  - › **String literals** are just hard-coded string values:
    - "hello!" "1234" "#nailedit"
    - Even though old C style, we still need to use it to write string literals
    - They have no methods that do things for us
    - (object-oriented programming didn't exist back in the day of C)
  - › **String objects** are objects with lots of helpful methods and operators:
    - `string s;`
    - `string piece = s.substr(0,3);`
    - `s.append(t); //or, equivalently: s += t;`

# C++ standard `string` object member functions (3.2)

```
#include <string>
```

Member function name	Description
<code>s.append(str)</code>	add text to the end of a string
<code>s.compare(str)</code>	return -1, 0, or 1 depending on relative ordering
<code>s.erase(index, length)</code>	delete text from a string starting at given index
<code>s.find(str)</code> <code>s.rfind(str)</code>	first or last index where the start of <code>str</code> appears in this string ( <b>returns <code>string::npos</code> if not found</b> )
<code>s.insert(index, str)</code>	add text into a string at a given index
<code>s.length()</code> or <code>s.size()</code>	number of characters in this string
<code>s.replace(index, Len, str)</code>	replaces <code>len</code> chars at given index with new text
<code>s.substr(start, Length)</code> or <code>s.substr(start)</code>	the next <code>length</code> characters beginning at <code>start</code> (inclusive); if <code>length</code> omitted, grabs till end of string

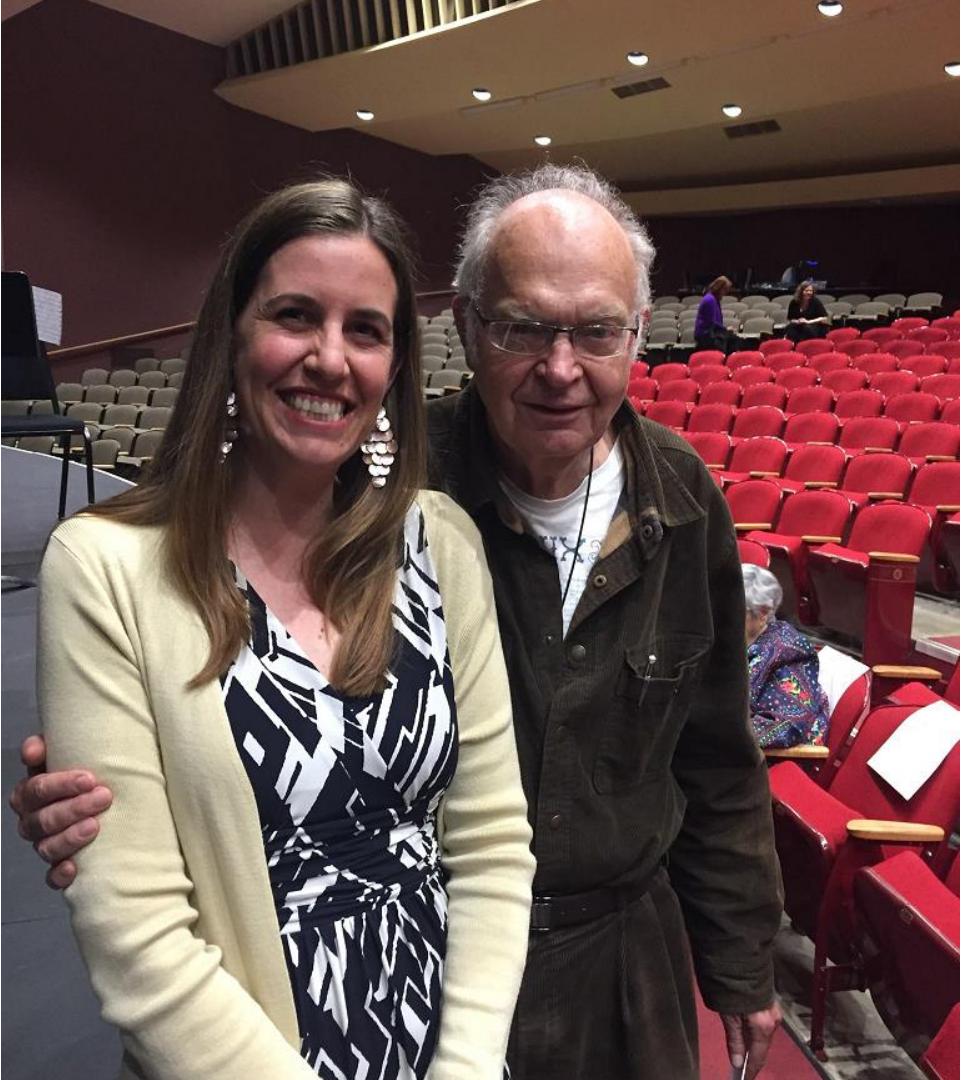
```
string name = "Donald Knuth";
if (name.find("Knu") != string::npos) {
    name.erase(5, 6);
}
```

# “Father of Algorithms” “Yoda of Silicon Valley” Donald Knuth

- Probably the most famous living computer scientist
- Stanford faculty (emeritus)
- Still lives on campus and comes to Gates building about once a week
- You'll see him on his bike



BFFs!



## C++ standard `string` object member functions (3.2)

```
#include <string>
```

Member function name	Description
<code>s.append(str)</code>	add text to the end of a string
<code>s.compare(str)</code>	return -1, 0, or 1 depending on relative ordering
<code>s.erase(index, Length)</code>	delete text from a string starting at given index
<code>s.find(str)</code> <code>s.rfind(str)</code>	first or last index where the start of <code>str</code> appears in this string ( <b>returns <code>string::npos</code> if not found</b> )
<code>s.insert(index, str)</code>	add text into a string at a given index
<code>s.length()</code> or <code>s.size()</code>	number of characters in this string
<code>s.replace(index, Len, str)</code>	replaces <code>len</code> chars at given index with new text
<code>s.substr(start, Length)</code> or <code>s.substr(start)</code>	the next <code>length</code> characters beginning at <code>start</code> (inclusive); if <code>length</code> omitted, grabs till end of string

Exercise: Write a line of code that pulls out the part of a string that is inside parentheses, assuming input variable `str` has the form "(blahblah)" where `blahblah` is any pattern of characters.

```
string insidePart = _____;
```

## Exercise solutions:

Exercise: Write a line of code that pulls out the part of a string that is inside parentheses, assuming variable `str` has the form "(blahblah)" where `blahblah` is any pattern of characters.

```
string insidePart = _____;
```

# Stanford library helpful string processing (*read3.7*)

```
#include "strlib.h"
```

- Unlike the previous ones, these take the string as a parameter.

Function name	Description
endsWith( <i>str, suffix</i> ) startsWith( <i>str, prefix</i> )	returns true if the given string begins or ends with the given prefix/suffix text
integerToString( <i>int</i> ) realToString( <i>double</i> ) stringToInteger( <i>str</i> ) stringToReal( <i>str</i> )	returns a conversion between numbers and strings
equalsIgnoreCase( <i>s1, s2</i> )	true if <i>s1</i> and <i>s2</i> have same chars, ignoring casing
toLowerCase( <i>str</i> ) toUpperCase( <i>str</i> )	returns an upper/lowercase version of a string
trim( <i>str</i> )	returns string with surrounding whitespace removed

# Today's Topics

## Introducing C++

- Hamilton example
  - › In QT Creator (the IDE for our class)
  - › Function prototypes
  - › <iostream> and cout
  - › C++ characters and strings
  - › Testing
- TODO this week:
  - › Sign ups for section will be **open on Thursday**, September 23rd at 5PM PT at [cs198.stanford.edu](http://cs198.stanford.edu). They will **close on Sunday**, September 26th at 5PM PT. Section meetings start week 2.
  - › [Assignment 0](#) is **due Friday**, September 24th at 11:59PM.
  - › **Qt Installation Help Session** in Huang Engineering Basement on **Thursday**, September 23rd from 7PM-9PM

# Programming Abstractions in C++

CS106B

Instructors:

Cynthia Bailey Lee

Julie Zelenski

# Today's Topics

1. Introductions
2. Course structure and procedures
3. What is this class? What do we mean by “abstractions”?
4. Introduce the C++ language
  - › Variables
  - › Functions

Upcoming lectures:

- Strings
- Testing
- Our first abstraction!

# Meet your instructors: Cynthia Bailey Lee

## RESEARCH INTERESTS

- UCSD PhD in large-scale computing
- Recently: computer science education, DEI in tech, justice and social impacts of tech

## TEACHING

- At Stanford since 2013
- CS106B, CS103, CS107, CS109, CS9, SSEA, CS80Q (introsem)

## SOFTWARE ENGINEER

- iPhone educational games
- Document clustering and classification

## AWAY FROM KEYBOARD

- Family, biking, hiking, pet chickens



# Meet your instructors: Julie Zelenski

## **PROUD STANFORD ALUM (UNDERGRAD AND GRAD)**

- FLI from CA Central Valley
- Coming to Stanford changed the arc of my life in every possible way
- Hope your experience is similarly transformative!  
Software engineering

## **NEXT COMPUTER, ACQUIRED BY APPLE**

## **LECTURER AT STANFORD**

- Fantastic colleagues, awesome students
- CS department a research powerhouse AND deeply committed to education

## **AWAY FROM KEYBOARD**

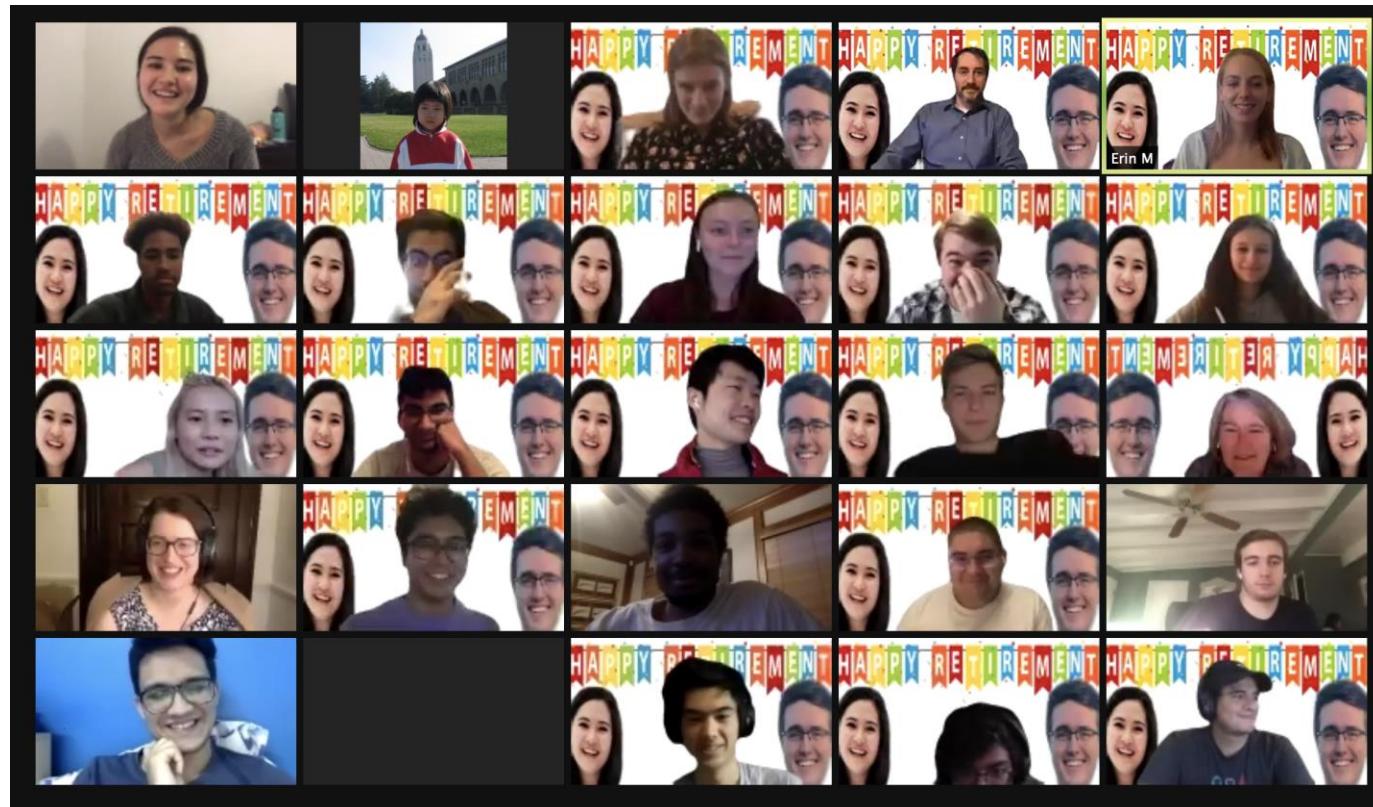
- Outdoors with family as much as possible



# Discussion Section, Section Leaders (“SLs”)

**Section Leaders are helpful undergraduate assistants who will:**

- run your discussion section each week
- help you when you have questions
- grade your work
- ... and much more

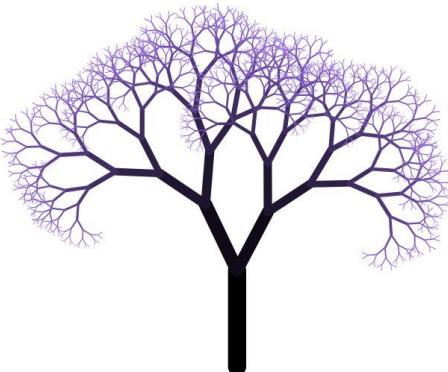


# What is CS 106B?

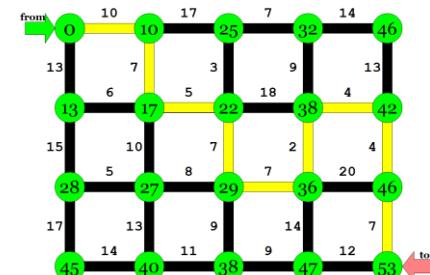
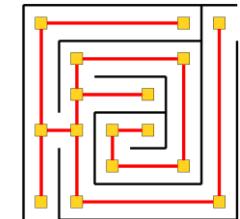
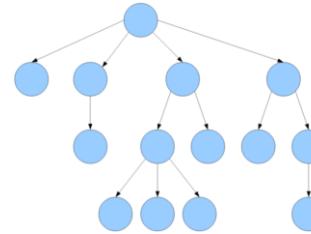
## CS 106B: Programming Abstractions

- solving big(ger) problems and processing big(ger) data
- learning to manage complex data structures
- algorithmic analysis and algorithmic techniques such as recursion
- programming style and software development practices
- familiarity with the C++ programming language

Prerequisite: CS 106A or equivalent



<http://cs106b.stanford.edu/>



Stanford University

# CS 106L

One unit course to learn the  
C++ language in depth.

Lecture:

T/Th 3:15-4:45 in 380-380C

Website:

<http://cs106l.stanford.edu>

Questions?

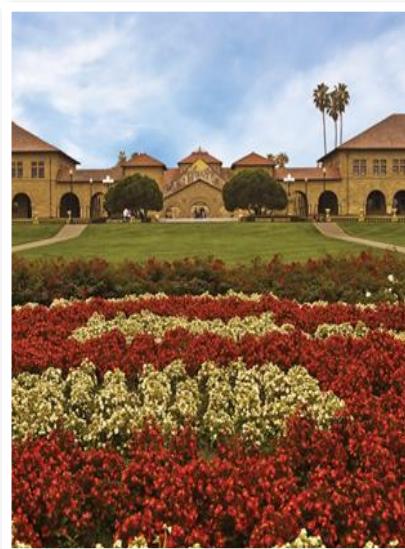
Email us at:

[sath@stanford.edu](mailto:sath@stanford.edu)

[fmcerk@stanford.edu](mailto:fmcerk@stanford.edu)

## Course Logistics

QUICK OVERVIEW OF HOW TO  
EARN THE GRADE YOU WANT  
IN CS106B



# What is this class about?

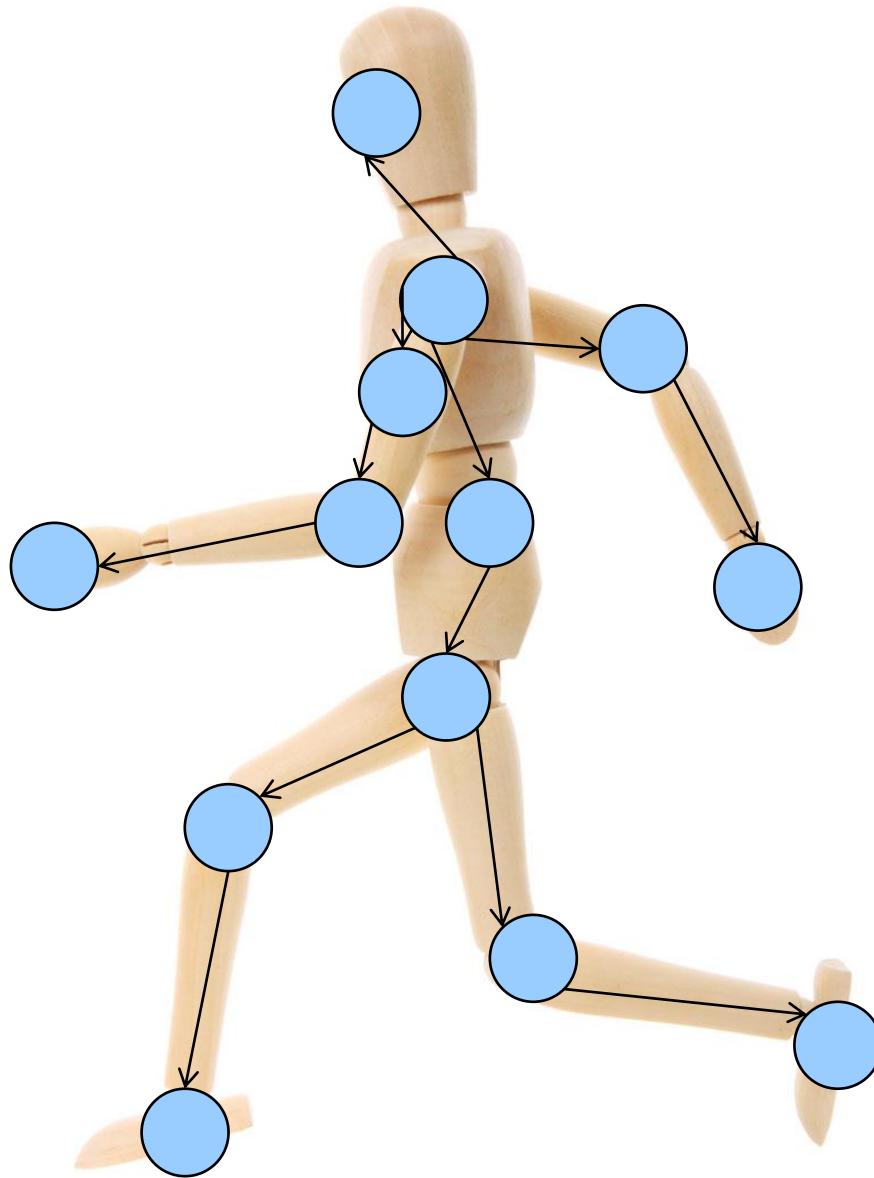
What do we mean by  
“abstractions”?

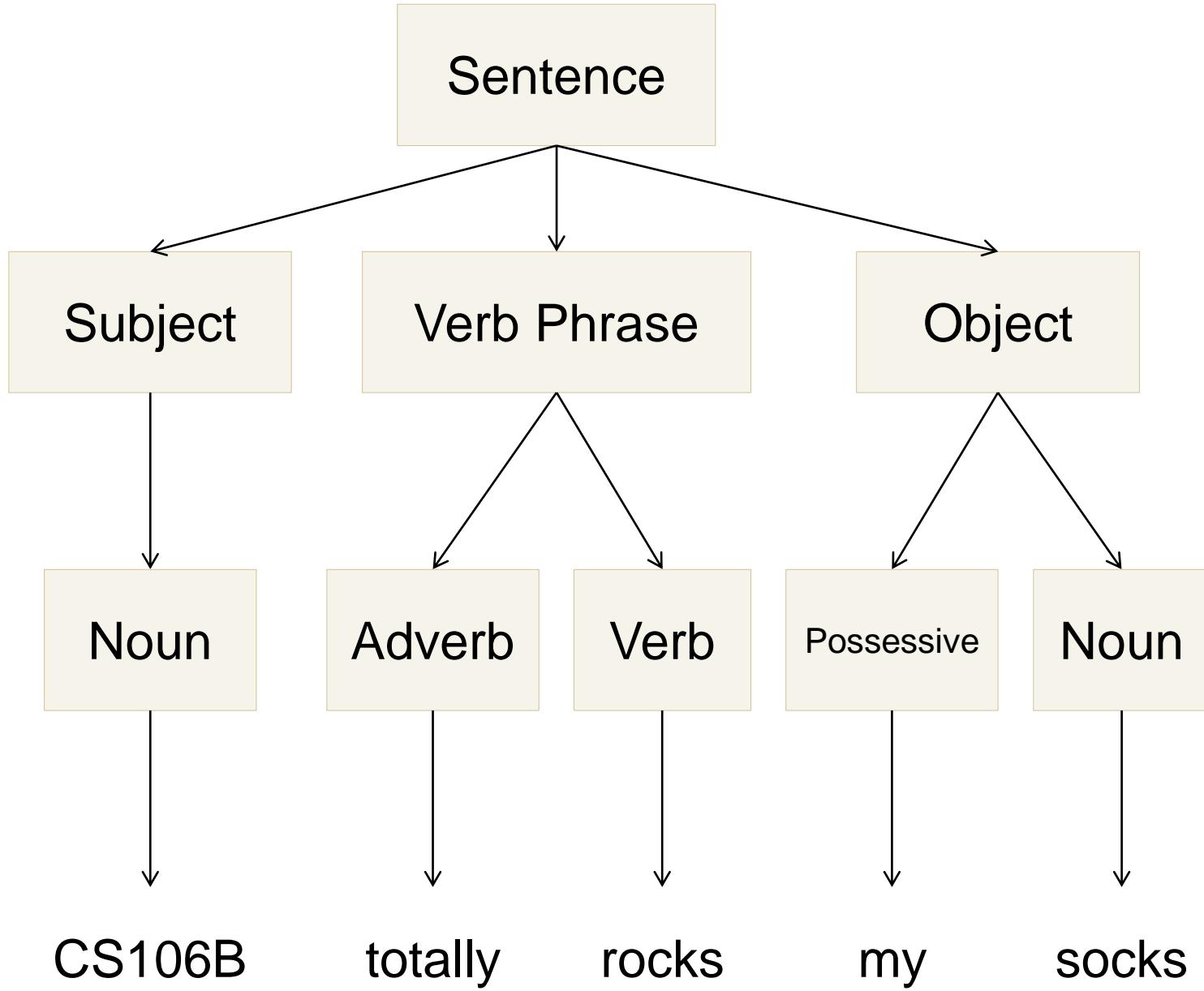


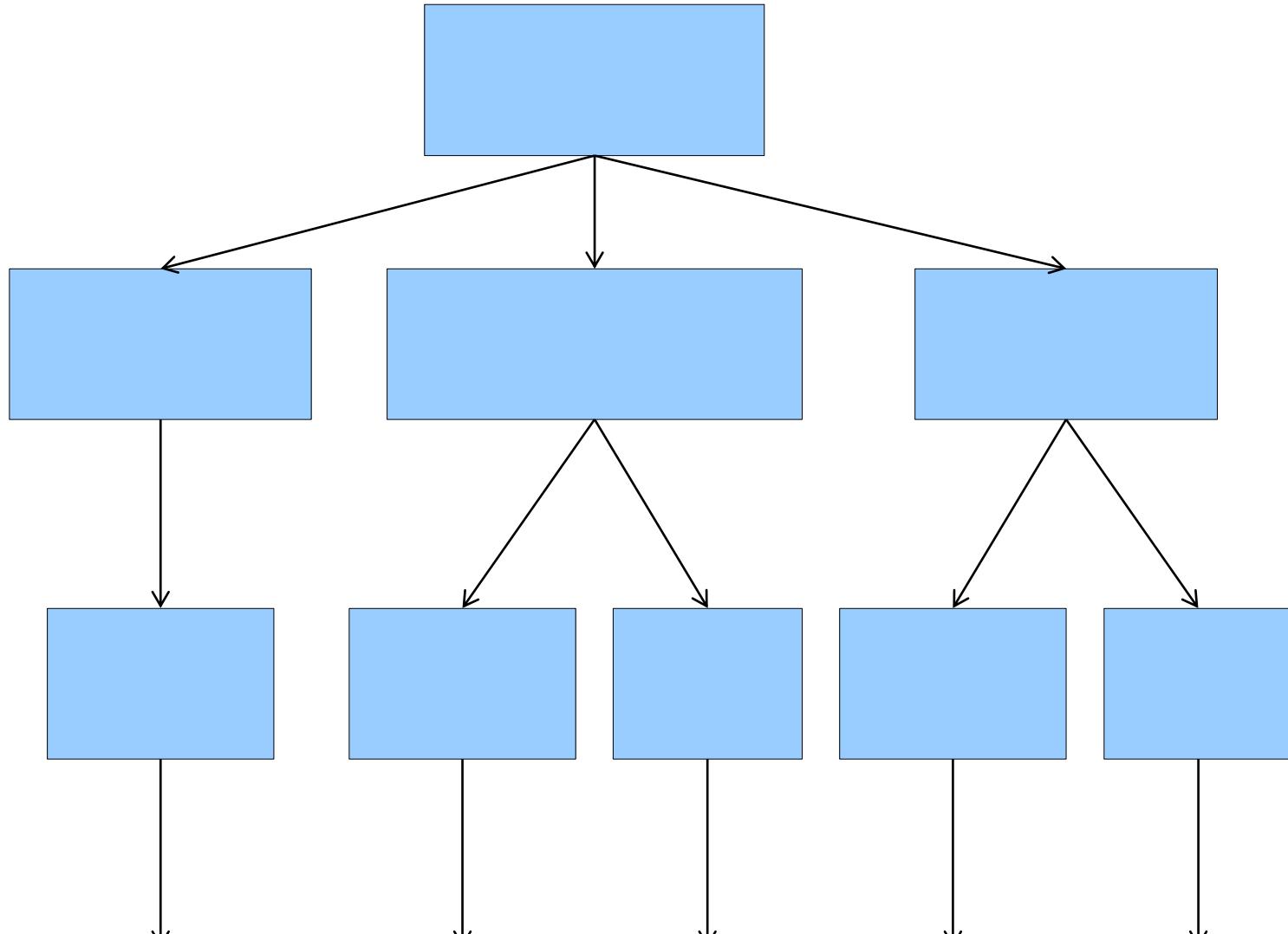
[Colatina, Carlos Nemer](#)

This file is licensed under the [Creative Commons Attribution 3.0 Unported](#) license.









CS106B

totally

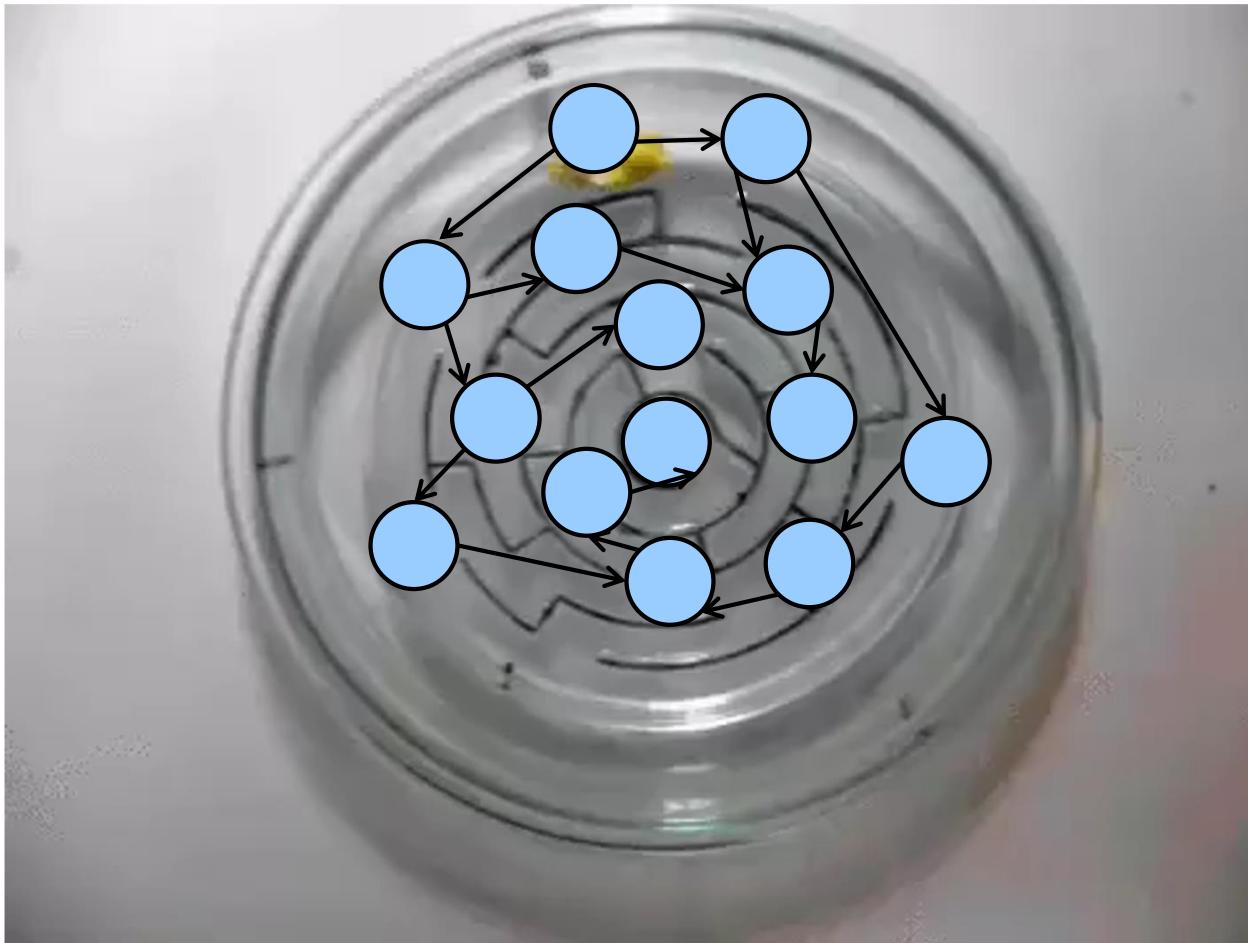
rocks

my

socks

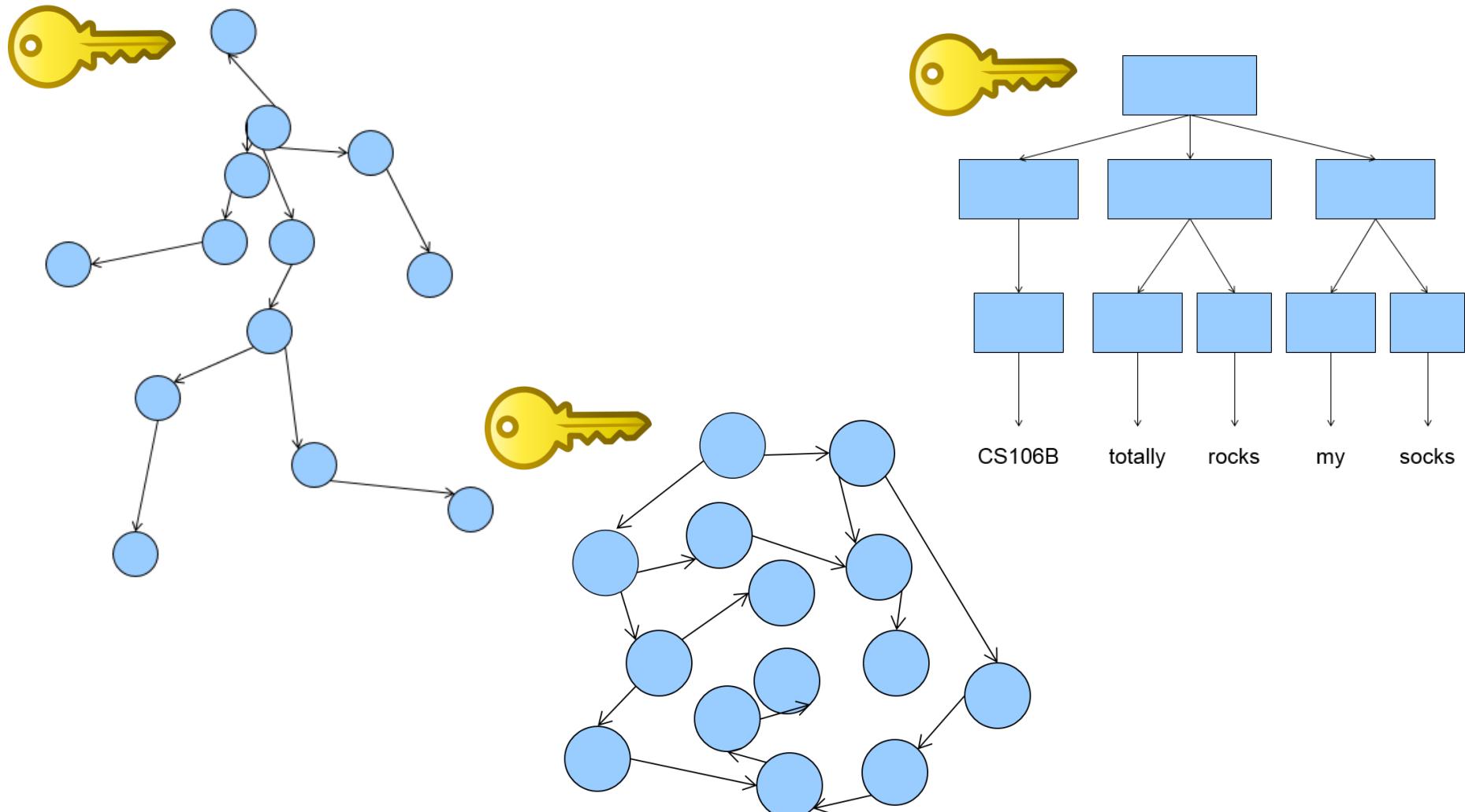


Stanford University



## In CS106B, you'll learn to:

1. Identify common underlying structures
2. Apply known algorithmic tools that solve diverse problems that share that structure



Building a vocabulary of **abstractions**  
makes it possible to represent and solve a huge  
variety of problems using known tools.

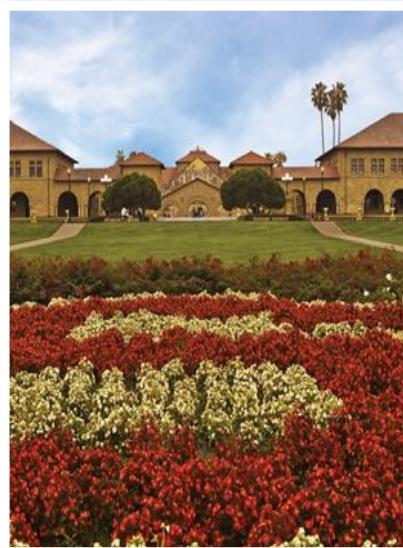
**A little bit of advice for this  
course...**

**You'll have what is effectively  
a superpower**

**Spend some time thinking about how  
you'll use it.**

# Welcome to C++

LET'S START CODING!!



# First C++ program (1.1)

```
/*
 * hello.cpp
 * This program prints a welcome message
 * to the user.
 */
#include <iostream>
#include "console.h"
using namespace std;

int main() {
    cout << "Hello, world!" << endl;
    return 0;
}
```

**Include** statements are like imports in Python. More on this in a moment.

Every C++ program has a **main** function. The program starts at main and executes its statements in sequence.

At program end, **main** returns 0 to indicate successful completion. A non-zero return value is an error code, but we won't use this method of error reporting in this class so we will always return zero.

## C++ variables and types (1.5-1.8)

- The C++ compiler is rather picky about *types* when it comes to variables.
- Types exist in languages like Python (see the two code examples at right), but you don't need to say much about them in the code. They just happen.
- The **first time** you introduce a variable in C++, you need to announce its type to the compiler (what kind of data it will hold).
  - After that, just use the variable name (don't repeat the type).
  - You won't be able to change the type of data later! C++ variables can only do one thing.

C++

```
int x = 42 + 7 * -5;  
double pi = 3.14159;  
char letter = 'Q';  
bool done = true;  
  
x = x - 3;
```

Python

```
x = 42 + 7 * -5  
pi = 3.14159  
letter = 'Q'  
done = True  
  
x = x - 3
```

## More C++ syntax examples (1.5-1.8)

```
for (int i = 0; i < 10; i++) {      // for loops
    if (i % 2 == 0) {                // if statements
        x += i;
    }
}                                     /* two comment styles */

while (letter != 'Q' && !done) {     // while loops, logic
    x = x / 2;
    if (x == 42) { return 0; }
}

binky(pi, 17);                      // function call
winky("this is a string");          // string usage
```

## Some C++ logistical details (2.2)

```
#include <libraryname>      // standard C++ library  
#include "libraryname.h"    // local project library
```

- Attaches a library for use in your program
- Note the differences (common bugs):
  - <> vs "
  - .h vs no .h

`using namespace name;`

- *Mostly, just don't worry about what this actually does/means!* Copy & paste the std line below into the top of your programs.
- Brings a group of features into global scope so your program can directly refer to them
- Many C++ standard library features are in namespace std so we write:
  - › `using namespace std;`
  - › “std” is short for “standard”