Programming Abstractions

CS106B

Cynthia Lee

Stanford University

Today’s Topics

Recursion Week continues!

- Today, two applications of recursion:
  - Fractals (will help us visualize the order of operations in recursion)
  - Binary Search (one of the fundamental algorithms of CS)

Next time:

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

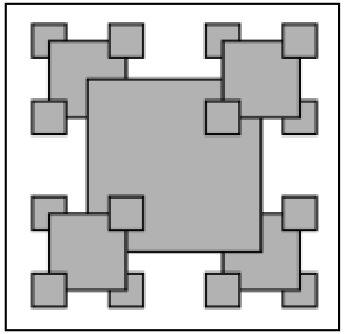
Fractals: Boxy Snowflake Fractal

Stanford University

Boxy Snowflake example

Where should this line of code be inserted to produce the pattern shown on the right?

```
drawFilledBox(w, cx, cy, dim, "Gray", "Black");
```



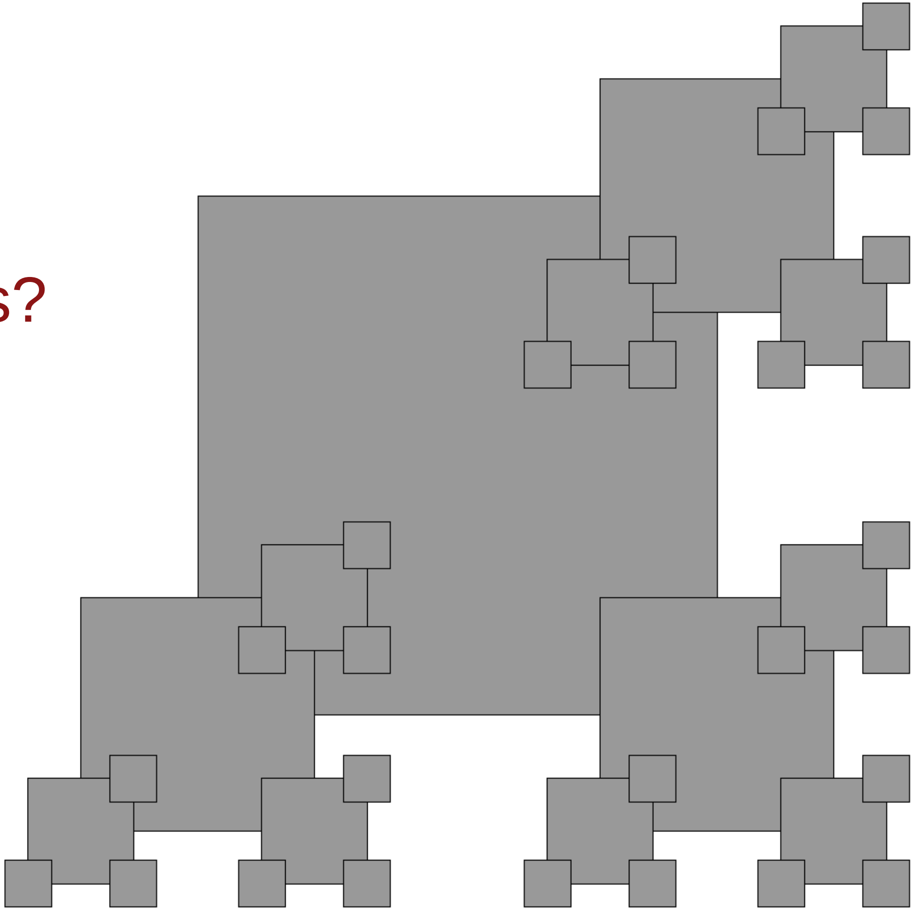
```
const double SCALE = 0.45;

void drawFractal(GWindow& w, double cx, double cy, double dim, int order) {
   if (order == 0) return;
   drawFractal(window, cx-dim/2, cy-dim/2, SCALE*dim, order-1);
                       (A) Insert code here
   drawFractal(window, cx+dim/2, cy+dim/2, SCALE*dim, order-1);
                       (B) Insert code here
   drawFractal(w, cx-dim/2, cy+dim/2, SCALE*dim, order-1);
                       (C) Insert code here
   drawFractal(window, cx+dim/2, cy-dim/2, SCALE*dim, order-1);
                       (D) Insert code here

}
```

(E) None of the above

Stanford University

Variants:

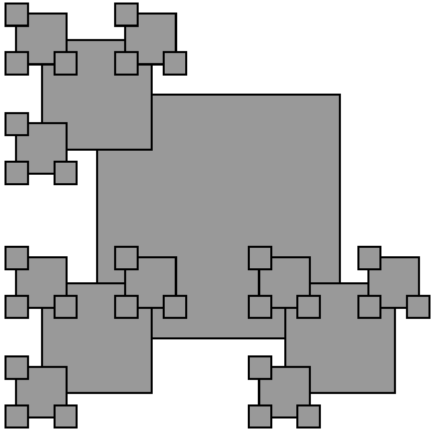
How can we code this?



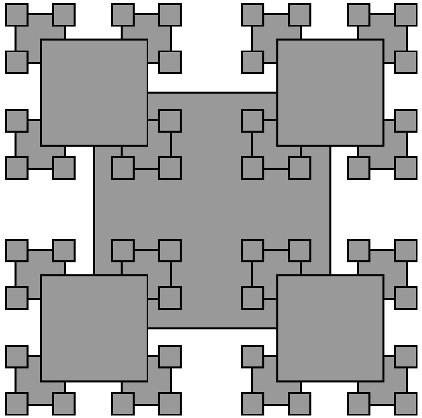
Real or Photoshop?

Can these be made by changing the order of lines and/or deleting lines in the draw() function?

(1)



(2)



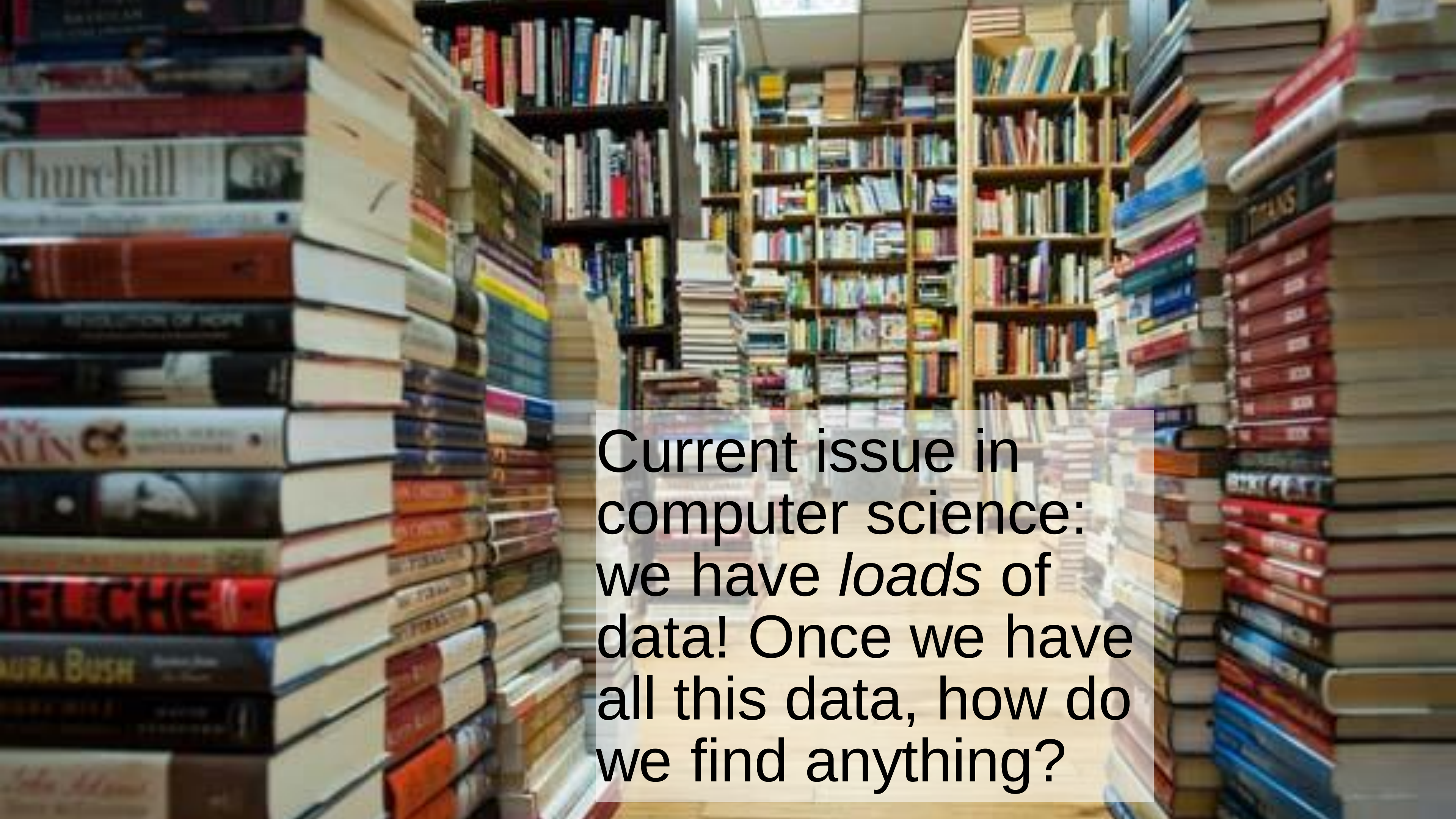
(A) Only #1 is real

(B) Only #2 is real

(C) Both are ‘shopped

(D) Both are real

Classic and important CS problem: ***searching***



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

Imagine storing **sorted** data in an array

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

Imagine storing **sorted** data in an array

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

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

Imagine storing **sorted** data in an array

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

Imagine storing **sorted** data in an array

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 en e fi half, and now only have the second half ea
- We co second half and proceed t at when we know we have a r

RECURSION!!

Jump rig u of the region to search

Stanford University

Designing a recursive algorithm

- 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 *our self* (the function we are in now) on the smaller bits to find out the answer to the problem we face

Binary Search

```
bool binarySearch(const Vector<int>& data, int key){
  return binarySearch(data, key, 0, data.size() - 1);
}

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, ______, ______);
    } else {
        return binarySearch(data, key, ______, ______);
    }
}
```