Programming Abstractions

CS106B

Cynthia Lee
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

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);
Stanford graphics lib

#include "gwindow.h"

gw.drawLine(x1, y1, x2, y2); draws a line between the given two points

gw.drawPolarLine(x, y, r, t); draws line from (x,y) at angle t of length r; returns the line's end point as a GPoint

gw.getPixel(x, y) returns an RGB int for a single pixel

gw.setColor(color); sets color with a color name string like "red", or #RRGGBB string like "#ff00cc", or RGB int

gw.setPixel(x, y, rgb); sets a single RGB pixel on the window

gw.drawOval(x, y, w, h); other shape and line drawing functions

gw.fillRect(x, y, w, h); ... (see online docs for complete member list)

GWindow gw(300, 200);
gw.setTitle("CS 106B Fractals");
gw.drawLine(20, 20, 100, 100);
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.
void cantorSet(GWindow& window, int x, int y, int width, int levels) {
    if (levels > 0) {
        // recursive case: draw line, then repeat by thirds
        window.drawLine(x, y, x + width, y);
        cantorSet(window, x, y + 20, width/3, levels-1);
        cantorSet(window, x + 2*width/3, y + 20, width/3, levels-1);
    }
    // else, base case: 0 levels, do nothing
}
void cantorSet(GWindow& window, int x, int y, int width, int levels) {
    if (levels > 0) {
        // recursive case: draw line, then repeat by thirds
        window.drawLine(x, y, x + width, y);
        cantorSet(window, x, y + 20, width/3, levels-1);
        cantorSet(window, x + 2*width/3, y + 20, width/3, levels-1);
    }
    // else, base case: 0 levels, do nothing
}
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?

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>7</td>
<td>8</td>
<td>13</td>
<td>25</td>
<td>29</td>
<td>33</td>
<td>51</td>
<td>89</td>
<td>90</td>
<td>95</td>
</tr>
</tbody>
</table>
Imagine storing **sorted** data in an array

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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>7</td>
<td>8</td>
<td>13</td>
<td>25</td>
<td>29</td>
<td>33</td>
<td>51</td>
<td>89</td>
<td>90</td>
<td>95</td>
</tr>
</tbody>
</table>

If you start at the front and proceed forward, each item you examine rules out 1 item.
Imagine storing sorted data in an array

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

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>8</td>
<td>13</td>
<td>25</td>
<td>29</td>
<td>33</td>
<td>51</td>
<td>89</td>
<td>90</td>
<td>95</td>
</tr>
</tbody>
</table>

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

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

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

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>7</td>
<td>8</td>
<td>13</td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>8</td>
<td>13</td>
<td>25</td>
<td>29</td>
<td>33</td>
<td>51</td>
<td>89</td>
<td>90</td>
<td>95</td>
</tr>
</tbody>
</table>

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 go to the front of the second half and proceed forward checking one item at a time… but why do that when we have a better way?

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