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
    - Callback to Big-O discussion
  - Generating sequences
    - *cough* Assignment 3 *cough*

Next time:

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

Binary Search Refresher

(RECALL FROM WEDNESDAY'S LECTURE)

Does this list of numbers contain X?

Context: we have a collection of numbers in a Vector, in sorted order.

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

- **Efficiency Hack: Jump to the middle of the Vector and look there to find:**
  - X (answer Yes)
  - A number greater than X (rule out entire second half of Vector)

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

  - A number less than X (rule out entire first half of Vector)

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

- Key observation: with *one* comparison, you ruled out *N/2* of the N cells in the Vector!

Does this list of numbers contain X?

Context: we have a collection of numbers in a Vector, in sorted order.

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

- **Extreme Efficiency Hack: Keep jumping to the middle!**
  - Let's say our first jump to the middle found a number less than X, so we ruled out the whole first half:

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

  - Now jump to the middle of the remaining second half:

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

- Key observation: we do one piece of work, then delegate the rest. **Recursion!!**

Binary Search Implementation

NOW WE UNDERSTAND THE APPROACH.

WHAT DOES THE CODE LOOK LIKE?

The recursive function pattern

From previous lecture

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 (1) bite off just one piece, and (2) 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!
        return n * factorial(n - 1);
    }
}
```

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);
    }
}
```

Base case: we shrank the search problem so tiny it no longer exists!

Recursive case:

Do one piece of work (comparison)

Delegate the rest of the work

```
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, _________, _________);
    }
}
```

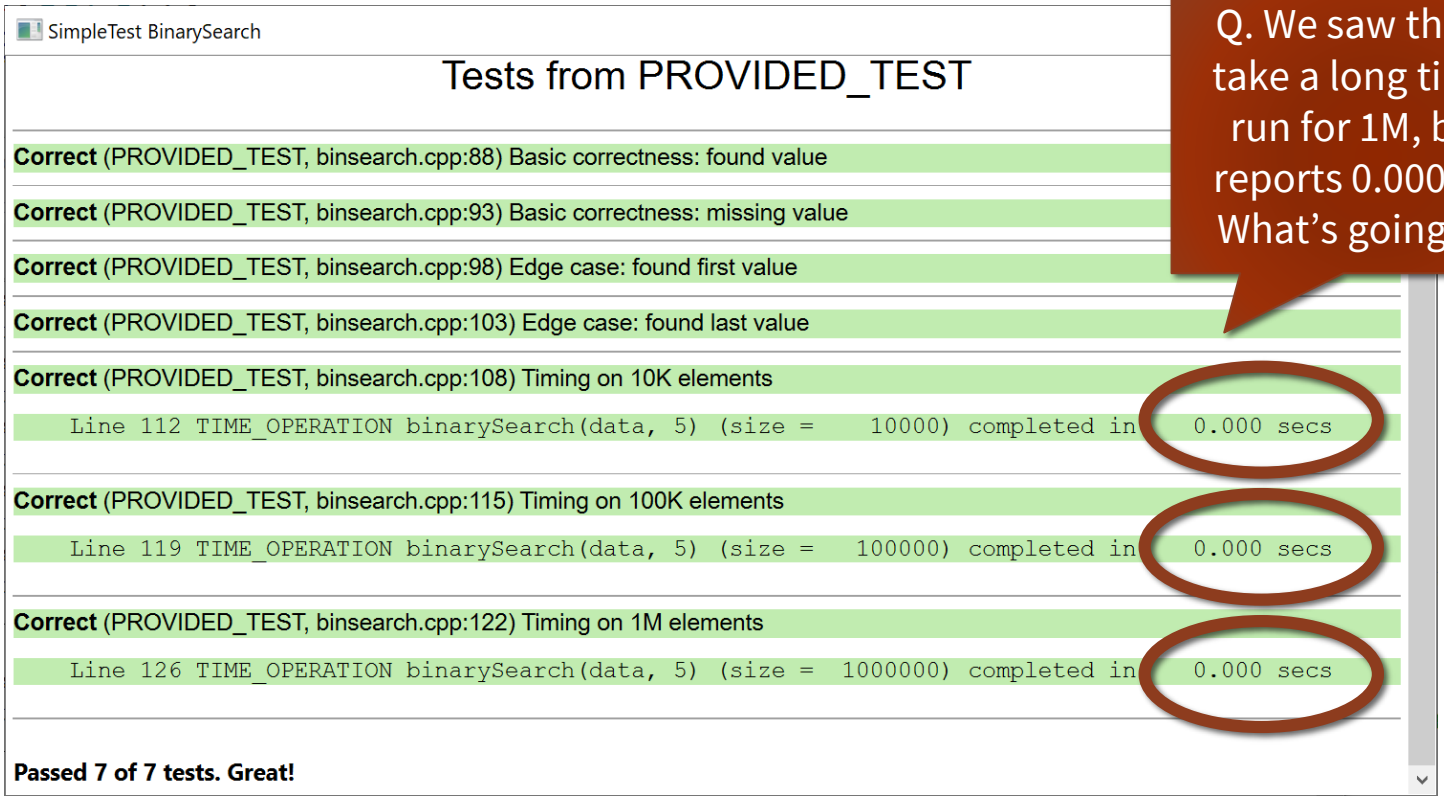
```
bool binarySearch(Vector<int>& data, int key) {
  // want to keep passing same data by reference for efficiency,
  // but then how do we cut in half?
  return binarySearch(data, key, 0, data.size() - 1); // 2 new params
}

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?

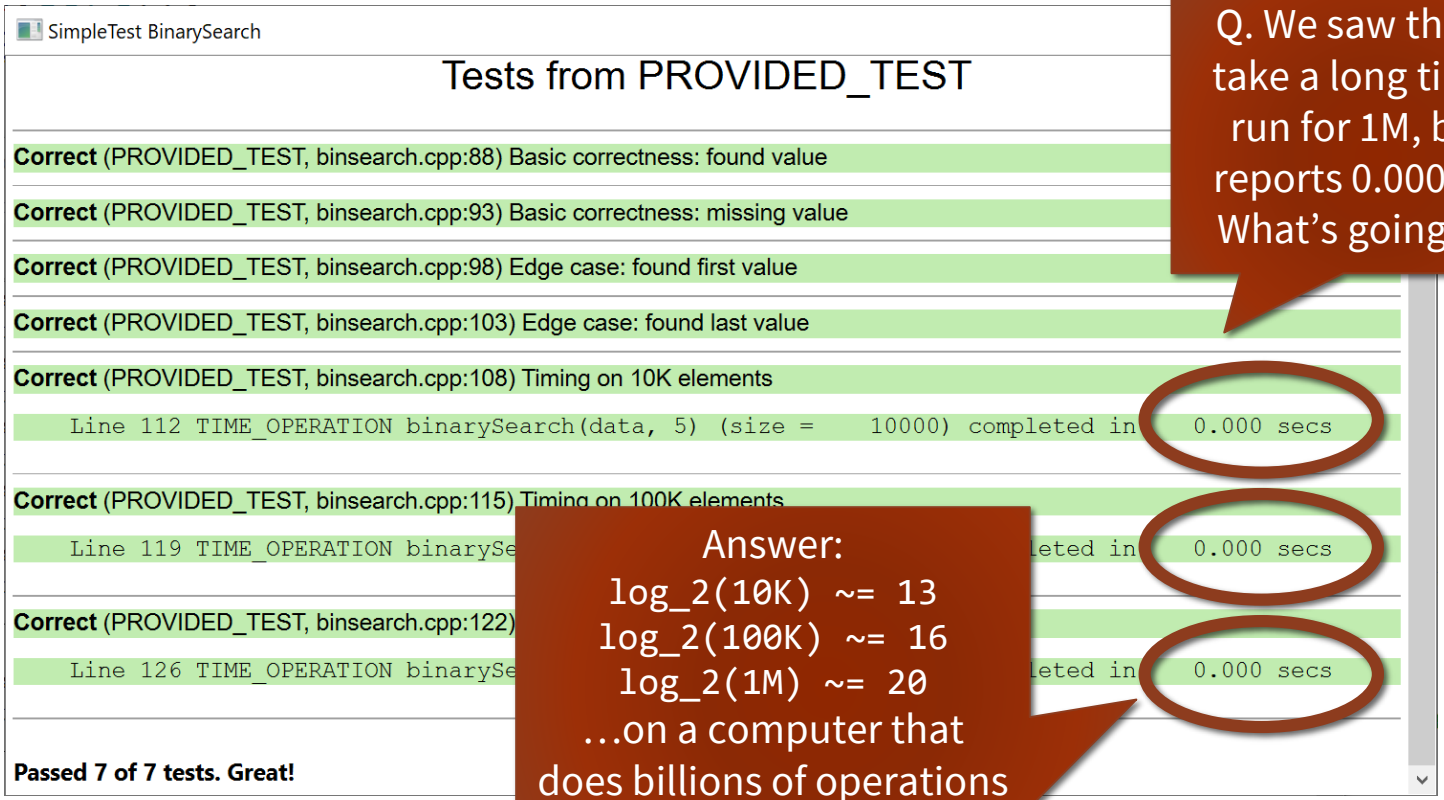
```
bool binarySearch(const Vector<int>& data, int key) {
  // want to keep passing same data by reference for efficiency,
  // but then how do we cut in half?
  return binarySearch(data, key, 0, data.size() - 1); // 2 new params
}

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



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



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) ~= 13`
`log_2(100K) ~= 16`
`log_2(1M) ~= 20`
…on a computer that does billions of operations per second!

$\log_2 n$	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	1.84×10^{19}
7	**128**	896	16,384	3.40×10^{38}
8	**256**	2,048	65,536	1.16×10^{77}
9	**512**	4,608	262,144	1.34×10^{154}
		40 s)	1,048,576 (.0003s)	1.80×10^{308}
30	**2,700,000,000**	84,591,843,105 (28s)	7,290,000,000,000,000, 000 (77 years)	$1.962227 \times 10^{812,780,998}$

2^n is clearly infeasible, but **look at $\log_2 n$—only a tiny fraction of a second!**

Big-O Key Take-Aways:

- **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)

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