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.

<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>

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

- 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.

<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>

**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:

<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>

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

<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>

**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

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

```c
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);
    }
}
```
We’ll write the real C++ code together on Friday, but here’s the outline/pseudocode of how it works:

```cpp
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, _________, _________);
    }
}
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);
    }
}
Q. We saw the test take a long time to run for 1M, but it reports 0.000 secs. What’s going on??

| 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 |
| Correct (PROVIDED_TEST, binsearch.cpp:115) Timing on 100K elements |
| Correct (PROVIDED_TEST, binsearch.cpp:122) Timing on 1M elements |

- Line 112 TIME_OPERATION binarySearch(data, 5) (size = 10000) completed in 0.000 secs
- Line 119 TIME_OPERATION binarySearch(data, 5) (size = 100000) completed in 0.000 secs
- Line 126 TIME_OPERATION binarySearch(data, 5) (size = 1000000) completed in 0.000 secs

Passed 7 of 7 tests. Great!
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:
\[
\log_2(10K) \approx 13
\]
\[
\log_2(100K) \approx 16
\]
\[
\log_2(1M) \approx 20
\]
...on a computer that does billions of operations per second!
<table>
<thead>
<tr>
<th>$\log_2 n$</th>
<th>$n$</th>
<th>$n \log_2 n$</th>
<th>$n^2$</th>
<th>$2^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>24</td>
<td>64</td>
<td>256</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>64</td>
<td>256</td>
<td>65,536</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
<td>160</td>
<td>1,024</td>
<td>4,294,967,296</td>
</tr>
<tr>
<td>6</td>
<td>64</td>
<td>384</td>
<td>4,096</td>
<td>$1.84 \times 10^{19}$</td>
</tr>
<tr>
<td>7</td>
<td>128</td>
<td>896</td>
<td>16,384</td>
<td>$3.40 \times 10^{38}$</td>
</tr>
<tr>
<td>8</td>
<td>256</td>
<td>2,048</td>
<td>65,536</td>
<td>$1.16 \times 10^{77}$</td>
</tr>
<tr>
<td>9</td>
<td>512</td>
<td>4,608</td>
<td>262,144</td>
<td>$1.34 \times 10^{154}$</td>
</tr>
<tr>
<td>30</td>
<td>2,700,000,000</td>
<td>84,591,843,105 ( (28s) )</td>
<td>7,290,000,000,000,000,000 ( (77 \text{ years}) )</td>
<td>$1.962227 \times 10^{812,780,998}$</td>
</tr>
</tbody>
</table>

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

```cpp
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);
}
```

Generating all possible 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

```cpp
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