Programming Abstractions

CS106B

Cynthia Lee
Today’s Topics

Recursion Week Fortnight continues!

- Today, two applications of recursion:
  - Binary Search (one of the fundamental algorithms of CS)
  - Loops + recursion for generating sequences and combinations

Next week:

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

<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>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

<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>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
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

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

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 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); // 2 new params
}

bool binarySearch(const 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.
```cpp
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, _________, _________);
    } 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, _______, _______);
    } 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);
    }
}
Generating sequences and combinations
Recursion pattern: generating sequences and combinations

• Example problems:
  • Given a deck of cards, output all possible distinct 5-card poker hands*
  • Generate all possible passwords of length N
  • Print all possible 7-digit phone numbers
  • Generate all possible 4-number combination lock combos
  › These are all variants of “generate all strings of length N,” but with different alphabets to choose from

* the exception is poker hands, where we are interested in sets, not sequences—we’ll just use a Set<> instead of a Vector<> for this change
Problem: generating sequences (approach #1: iterative)

• How can we code generating all possible N-digit combination lock combos?
• Let’s try: **loops**, and an example with **N=3**

```c++
void generateAllCombos(Vector<Vector<int>>& allCombos) {
    Vector<int> combo(3);
    for (int num0 = 0; num0 <= MAX_COMBO_NUM; num0++) {
        combo[0] = num0;
        for (int num1 = 0; num1 <= MAX_COMBO_NUM; num1++) {
            combo[1] = num1;
            for (int num2 = 0; num2 <= MAX_COMBO_NUM; num2++) {
                combo[2] = num2;
                allCombos.add(combo);
            }
        }
    }
}
```

**Question:** How do we get this code to work for 2- or 4- (or any N-) digit PINs, not just 3-digit?
Problem: generating sequences (approach #1: iterative)

• How can we code generating all possible N-digit combination lock combos?
• Let’s try: loops, and an example with N=2

```cpp
void generateAllCombos(Vector<Vector<int>> & allCombos) {
    Vector<int> combo(2);
    for (int num0 = 0; num0 <= MAX_COMBO_NUM; num0++) {
        combo[0] = num0;
        for (int num1 = 0; num1 <= MAX_COMBO_NUM; num1++) {
            combo[1] = num1;
            allCombos.add(combo);
        }
    }
}
```

**Question:** How do we get this code to work for 2- or 4- (or any N-) digit PINs, not just 3-digit?

**Answer:** Could make different versions
Problem: generating sequences (approach #1: iterative)

- How can we code generating all possible N-digit combination lock combos?
- Let’s try: loops, and an example with N=4

```cpp
void generateAllCombos(Vector<Vector<int>>& allCombos) {
    Vector<int> combo(4);
    for (int num0 = 0; num0 <= MAX_COMBO_NUM; num0++) {
        combo[0] = num0;
        for (int num1 = 0; num1 <= MAX_COMBO_NUM; num1++) {
            combo[1] = num1;
            for (int num2 = 0; num2 <= MAX_COMBO_NUM; num2++) {
                combo[2] = num2;
                for (int num3 = 0; num3 <= MAX_COMBO_NUM; num3++) {
                    combo[2] = num2;
                    allCombos.add(combo);
                }
            }
        }
    }
}
```

**Question:** How do we get this code to work for 2- or 4- (or any N-) digit PINs, not just 3-digit?

**Answer:** Could make different versions
Problem: generating sequences (approach #1: iterative)

• How can we code generating all possible N-digit combination lock combos?
• Let’s try: loops, and an example with $N=\text{length parameter, up to 3}$

```c
void generateAllCombos(int length Vector<Vector<int>>& allCombos) {
    Vector<int> combo(length);
    for (int num0 = 0; num0 <= MAX_COMBO_NUM; num0++) {
        combo[0] = num0;
        if (length == 1) { allCombos.add(combo); continue; }
        for (int num1 = 0; num1 <= MAX_COMBO_NUM; num1++) {
            combo[1] = num1;
            if (length == 2) { allCombos.add(combo); continue; }
            for (int num2 = 0; num2 <= MAX_COMBO_NUM; num2++) {
                combo[2] = num2;
                allCombos.add(combo);
            }
        }
    }
}
```

**Question:** How do we get this code to work for 2- or 4- (or any N-) digit PINs, not just 3-digit?

**Answer:** Could end sequence generation early...
Problem: generating sequences (approach #1: iterative)

- How can we code generating all possible N-digit combination lock combos?
- Let’s try: **loops**, and an example with **N=length parameter, up to 3**

```cpp
void generateAllCombos(int length Vector<Vector<int>>& allCombos) {
    Vector<int> combo(length);
    for (int num0 = 0; num0 <= MAX_COMBO_NUM; num0++) {
        combo[0] = num0;
        if (length == 1) { allCombos.add(combo); continue; }
        for (int num1 = 0; num1 <= MAX_COMBO_NUM; num1++) {
            combo[1] = num1;
            if (length == 2) { allCombos.add(combo); continue; }
            for (int num2 = 0; num2 <= MAX_COMBO_NUM; num2++) {
                combo[2] = num2;
                allCombos.add(combo);
            }
        }
    }
}
```

**Question:** How do we get this code to work for 2- or 4-(or any N-) digit PINs, not just 3-digit?

**Answer:** Could end sequence generation early...

Horrendously ugly code, and you would need to know max N in advance to code loops to that depth, since you can only end early
Problem: generating sequences (approach #1: iterative)

- How can we code generating all possible N-digit combination lock combos?
- Let’s try: loops, and an example with N=3

```cpp
void generateAllCombos(Vector<Vector<int>>& allCombos) {
    Vector<int> combo(4);
    for (int num0 = 0; num0 <= MAX_COMBO_NUM; num0++) {
        combo[0] = num0;
        for (int num1 = 0; num1 <= MAX_COMBO_NUM; num1++) {
            combo[1] = num1;
            for (int num2 = 0; num2 <= MAX_COMBO_NUM; num2++) {
                combo[2] = num2;
                allCombos.add(combo);
            }
        }
    }
}
```

**Question:** How do we get this code to work for 2- or 4- (or any N-) digit PINs, not just 3-digit?

**Answer:** 😞 Can’t dynamically retype loop code!
void generateAllCombos(int length, Vector<Vector<int>>& allCombos) {
    Vector<int> combo;
    generateAllCombos(length, allCombos, combo);
}

void generateAllCombos(int length, Vector<Vector<int>>& allCombos,
                        Vector<int>& combo) {
    // base case: this combo is full-length and ready to add
    if (combo.size() == length) {
        allCombos.add(combo);
        return;
    }
    // recursive case: add all possible next numbers to combo and continue
    for (int num = 0; num <= MAX_COMBO_NUM; num++) {
        combo.add(num);
        generateAllCombos(length, allCombos, combo);
        combo.removeBack();
    }
}

Problem: generating sequences (approach #2: recursive)

Your Turn: What goes in the blank base case test?
Recursion pattern: designing a “wrapper” function

```cpp
void generateAllCombos(int length, Vector<Vector<int>>& allCombos) {
    Vector<int> combo;
    generateAllCombos(length, allCombos, combo);
}

void generateAllCombos(int length, Vector<Vector<int>>& allCombos, Vector<int>& combo) {
    // base case: this combo is full-length and ready to add
    if (combo.size() == length) {
        allCombos.add(combo);
        return;
    }
    // recursive case: add all possible next numbers to combo and continue
    for (int num = 0; num <= MAX_COMBO_NUM; num++) {
        combo.add(num);
        generateAllCombos(length, allCombos, combo);
        combo.removeBack();
    }
}
```
Recursion pattern: designing a “wrapper” function

```cpp
void generateAllCombos(int length, Vector<Vector<int>>& allCombos) {
    Vector<int> combo;
    generateAllCombos(length, allCombos, combo);
}

void generateAllCombos(int length, Vector<Vector<int>>& allCombos,
                        Vector<int>& combo) {
    // base case: this combo is full-length and ready to add
    if (combo.size() == length) {
        allCombos.add(combo);
        return;
    }

    // recursive case: add all possible next numbers to combo and continue
    for (int num = 0; num <= MAX_COMBO_NUM; num++) {
        combo.add(num);
        generateAllCombos(length, allCombos, combo);
        combo.removeBack();
    }
}
```