Recursion
Chapter 8

CS 3358
Spring 2015
Jill Seaman

Sections 8.1-8.4, (8.5 if you can)

What is recursion?

- Generally, when something contains a reference to itself
- Math: defining a function in terms of itself
- Computer science: when a function calls itself

How can a function call itself?

- What happens when this function is called?

```cpp
void message() {
    cout << "This is a recursive function.\n";
    message();
}

int main() {
    message();
}
```

How can a function call itself?

- Infinite Recursion:

```
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
...
Recursive message() modified

• How about this one?

```c++
void message(int n) {
    if (n > 0) {
        cout << "This is a recursive function.\n";
        message(n-1);
    }
}
int main() {
    message(5);
}
```

Tracing the calls

• 6 nested calls to message:

```c++
message(5):
    outputs "This is a recursive function"
    calls message(4):
        outputs "This is a recursive function"
        calls message(3):
            outputs "This is a recursive function"
            calls message(2):
                outputs "This is a recursive function"
                calls message(1):
                    outputs "This is a recursive function"
                    calls message(0):
                        does nothing, just returns
```

• depth of recursion (#times it calls itself) = 5

Why use recursion?

• It is true that recursion is never **required** to solve a problem
  - Any problem that can be solved with recursion can also be solved using iteration.
• Recursion requires extra overhead: function call + return mechanism uses extra resources

However:
• Some repetitive problems are more easily and naturally solved with recursion
  - Iterative solution may be unreadable to humans

Why use recursion?

• Recursion is the primary method of performing repetition in most **functional** languages.
  - Implementations of functional languages are designed to process recursion efficiently
  - Iterative constructs that are added to many functional languages often don’t fit well in the functional context.

• Once programmers adapt to solving problems using recursion, the code produced is generally shorter, more elegant, easier to read and debug.
How to write recursive functions

- Branching is required!! (If or switch)
- Find a base case
  - one (or more) values for which the result of the function is **known** (no repetition required to solve it)
  - no recursive call is allowed here
- Develop the recursive case
  - For a given argument (say n), assume the function works for a smaller value (n-1).
  - Use the result of calling the function on n-1 to form a solution for n

Recursive function example

**factorial**

- Mathematical definition of n! (factorial of n)
  
  - if n=0 then n! = 1
  - if n>0 then n! = 1 x 2 x 3 x ... x (n-1) x n

- What is the base case?
  - n=0 (result is 1)

- If we assume (n-1)! can be computed, how can we get n! from that?
  - n! = n * (n-1)!

Recursive function example

```cpp
int factorial(int n) {
    if (n==0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
int main() {
    int number;
    cout << "Enter a number ";
    cin >> number;
    cout << "The factorial of " << number << " is " << factorial(number) << endl;
}
```
Tracing the calls

- Calls to factorial:

  \[
  \text{factorial}(4): \quad \text{return } 4 \times \text{factorial}(3); \quad =4 \times 6 = 24 \\
  \text{calls factorial}(3): \quad \text{return } 3 \times \text{factorial}(2); \quad =3 \times 2 = 6 \\
  \text{calls factorial}(2): \quad \text{return } 2 \times \text{factorial}(1); \quad =2 \times 1 = 2 \\
  \text{calls factorial}(1): \quad \text{return } 1 \times \text{factorial}(0); \quad =1 \times 1 = 1 \\
  \text{calls factorial}(0): \quad \text{return } 1; 
  \]

- Every call except the last makes a recursive call
- Each call makes the argument smaller

Recursive functions over ints

- Many recursive functions (over integers) look like this:

```java
type f(int n) {
    if (n==0)
        //do the base case
    else
        // ... f(n-1) ...
}
```

- Note these functions are undefined for \( n < 0 \).

Recursive functions over lists

- You can write recursive functions over lists using the length of the list instead of \( n \)
  - base case: length=0  ==> empty list
  - recursive case: assume \( f \) works for list of length \( n-1 \), what is the answer for a list with one more element?
- We will do examples with:
  - arrays
  - vectors
  - linked lists
  - strings

Recursive function example

sum of the list

- Recursive function to compute sum of a list of numbers
- What is the base case?
  - length=0  (empty list)  sum = 0
- If we assume we can sum the first \( n-1 \) items in the list, how can we get the sum of the whole list from that?
  - \( \text{sum (list)} = \text{sum (list[0..n-2])} + \text{list[n-1]} \)

Assume I am given the answer to this, the sum of the first \( n-1 \) items
Recursive function example
sum of a list: array

```c
int sum(int a[], int size) {  //size is number of elems
    if (size==0)
        return 0;
    else
        return sum(a,size-1) + a[size-1];
}
```

For a list with size = 4:
```
sum(a,4) =
(sum(a,3) + a[3]) =
(sum(a,2) + a[2]) + a[3] =
((sum(a,1) + a[1]) + a[2]) + a[3] =
(((sum(a,0) + a[0]) + a[1]) + a[2]) + a[3] =
((0 + a[0]) + a[1]) + a[2]) + a[3]
```

Recursive function example
sum of a list: vector

```c
int sum(vector<int> v) {
    if (v.size()==0)
        return 0;
    else {
        int x = v.back();
        v.pop_back();
        return x + sum(v);
    }
}
```

```c
int main () {
    vector<int> a;
    a.push_back(10);
    a.push_back(20);
    a.push_back(30);
    cout << "sum " << sum(a) << endl;
    cout << "size " << a.size() << endl;
}
```

```
• v.pop_back() creates the shorter vector
• Aren’t we removing all the elements from a?
  - No (why not?)   Hint: Pass by value
  - But something else bad is happening each time.
    Hint: Pass by value
```

Recursive function example
sum of a list: vector without copying

```
int sumRec(vector<int> & v) {
    if (v.size()==0)
        return 0;
    else {
        int x = v.back();
        v.pop_back();
        return x + sumRec(v);
    }
}
```

```
int sum (const vector<int> & x) {
    return sumRec(x);
}
```

```
• Sometimes an auxiliary or driver function is needed to set things up before starting recursion.
```

Recursive function example
sum of a list: linked list

```
int List_3358::sum() {
    return sumNodes(head);
}
```

```
int List_3358::sumNodes(Node *p) {
    if (p==NULL)
        return 0;
    else {
        int x = p->value;
        return x + sumNodes(p->next);
    }
}
```

```
• Add a sum function to List_3358_LL.h
```

```
// this is the public one
int List_3358::sum() {
    return sumNodes(head);
}
```

```
// this one is private
int List_3358::sumNodes(Node *p) {
    if (p==NULL)
        return 0;
    else {
        int x = p->value;
        return x + sumNodes(p->next);
    }
}
```

```
• sumNodes(p) will sum the Nodes starting with the one p points to until the end of the list (NULL)
  passes address of the next Node, (making the list shorter)
```

17
18
19
20
Summary of the list examples

- How to determine empty list, single element, and the shorter list to perform recursion on.

<table>
<thead>
<tr>
<th>Array</th>
<th>Vector</th>
<th>Linked list</th>
</tr>
</thead>
<tbody>
<tr>
<td>size is a parameter</td>
<td>v.size() == 0</td>
<td>p == NULL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Base case</th>
<th>last(or first) element</th>
<th>shorter list (recursive call)</th>
</tr>
</thead>
<tbody>
<tr>
<td>size == 0</td>
<td>a[size-1]</td>
<td>use size-1 v.pop_back()*</td>
</tr>
<tr>
<td>v.size() == 0</td>
<td>v.back()</td>
<td>p-&gt;value</td>
</tr>
<tr>
<td>p == NULL</td>
<td></td>
<td>p-&gt;next</td>
</tr>
</tbody>
</table>

*may need to copy original vector

The Substring function

- C++ string member function: substr
  - string substr (int pos, int len) const;
  - pos  position of the first character to be copied as a substring. Note: The first character is denoted by a value of 0 (not 1).
  - len  Number of characters to include in the substring. If pos+len is greater than the number of characters in the string, the whole value of the string beginning at start is returned.

string x = “hello there”;
cout << x.substr(3,5) << endl;
cout << x.substr(6,50) << endl;

Recursive function example

- Recursive function to count the number of times a specific character appears in a string

```cpp
int numChars(char target, string str) {
    if (str.empty()) {
        return 0;
    } else {
        int result = numChars(target, str.substr(1,str.size()));
        if (str[0] == target)
            return 1 + result;
        else
            return result;
    }
}
```

int main() {
    string a = "hello";
cout << a << numChars(’l’ , a) << endl;
}
Three required properties of recursive functions

- A Base case
  - a non-recursive branch of the function body.
  - must return the correct result for the base case
- Smaller caller
  - each recursive call must pass a smaller version of the current argument.
- Recursive case
  - assuming the recursive call works correctly, the code must produce the correct answer for the current argument.

Recursive function example: greatest common divisor

- Greatest common divisor of two non-zero ints is the largest positive integer that divides the numbers without a remainder
- This is a variant of Euclid’s algorithm:
  \[
gcd(x, y) = \begin{cases} 
  y & \text{if } y \text{ divides } x \text{ evenly, otherwise;} \\
  gcd(y, \text{remainder of } x/y) & \text{otherwise in c++}
\end{cases}
\]
- It’s a recursive mathematical definition
- If \( x < y \), then \( x \% y \) is \( x \) (so \( gcd(x, y) = gcd(y, x) \))
- This moves the larger number to the first position.

Recursive function example: greatest common divisor

- Code:

  ```cpp
  int gcd(int x, int y) {
    cout << "gcd called with " << x << " and " << y << endl;
    if (x % y == 0) {
      return y;
    } else {
      return gcd(y, x % y);
    }
  }

  int main() {
    cout << "GCD(9,1): " << gcd(9,1) << endl;
    cout << "GCD(1,9): " << gcd(1,9) << endl;
    cout << "GCD(9,2): " << gcd(9,2) << endl;
    cout << "GCD(70,25): " << gcd(70,25) << endl;
    cout << "GCD(25,70): " << gcd(25,70) << endl;
  }
  ```

- Output:

  gcd called with 9 and 1
  GCD(9,1): 1
  gcd called with 1 and 9
  gcd called with 9 and 1
  GCD(1,9): 1
  gcd called with 9 and 2
  gcd called with 2 and 1
  GCD(9,2): 1
  gcd called with 70 and 25
  gcd called with 25 and 20
  gcd called with 20 and 5
  GCD(70,25): 5
  gcd called with 25 and 70
  gcd called with 70 and 25
  gcd called with 25 and 20
  gcd called with 20 and 5
  GCD(25,70): 5
Recursive function example
Fibonacci numbers

• Series of Fibonacci numbers:
  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

• Starts with 0, 1. Then each number is the sum of the two previous numbers
  \[ F_0 = 0 \]
  \[ F_1 = 1 \]
  \[ F_i = F_{i-1} + F_{i-2} \quad (\text{for } i > 1) \]

• It’s a recursive definition

Recursive function example
Fibonacci numbers

• Code:

```cpp
int fib(int x) {
    if (x<=1)
        return x;
    else
        return fib(x-1) + fib(x-2);
}
```

```cpp
int main() {
    cout << "The first 13 fibonacci numbers: " << endl;
    for (int i=0; i<13; i++)
        cout << fib(i) << " ";
    cout << endl;
}
```

The first 13 fibonacci numbers:
0 1 1 2 3 5 8 13 21 34 55 89 144

Recursive function example
Fibonacci numbers

• Modified code to count the number of calls to fib:

```cpp
int fib(int x, int &count) {
    count++;
    if (x<=1)
        return x;
    else
        return fib(x-1, count) + fib(x-2, count);
}
```

```cpp
int main() {
    cout << "The first 40 fibonacci numbers: " << endl;
    for (int i=0; i<40; i++) {
        int count = 0;
        int x = fib(i,count);
        cout << "fib (" << i << ")= " << x << "  # of recursive calls to fib = " << count << endl;
    }
}
```

The first 40 fibonacci numbers:
fib (0)= 0  # of recursive calls to fib = 1
fib (1)= 1  # of recursive calls to fib = 1
fib (2)= 1  # of recursive calls to fib = 3
fib (3)= 2  # of recursive calls to fib = 5
fib (4)= 3  # of recursive calls to fib = 9
fib (5)= 5  # of recursive calls to fib = 15
fib (6)= 8  # of recursive calls to fib = 25
fib (7)= 13 # of recursive calls to fib = 41
fib (8)= 21 # of recursive calls to fib = 67
fib (9)= 34 # of recursive calls to fib = 109
fib (10)= 55 # of recursive calls to fib = 177
fib (11)= 89 # of recursive calls to fib = 287
fib (12)= 144 # of recursive calls to fib = 465
fib (13)= 233 # of recursive calls to fib = 753
...
...
Recursive function example
Fibonacci numbers

- Why are there so many calls to \( \text{fib} \)?
  \( \text{fib}(n) \) calls \( \text{fib}(n-1) \) and \( \text{fib}(n-2) \)

- Say it computes \( \text{fib}(n-2) \) first.
- When it computes \( \text{fib}(n-1) \), it computes \( \text{fib}(n-2) \) again

  \[ \text{fib}(n-1) \text{ calls } \text{fib}((n-1)-1) \text{ and } \text{fib}((n-1)-2) \]
  \[ = \text{fib}(n-2) \text{ and } \text{fib} (n-3) \]

- It’s not just double the work. It’s double the work for each recursive call.
- Each recursive call does more and more redundant work

Recursive function example
Fibonacci numbers

- Trace of the recursive calls for \( \text{fib}(5) \)

Binary Search

- Find an item in a list, return the index or -1
- Works only for SORTED lists
- Compare target value to middle element in list.
  - if equal, then return index
  - if less than middle elem, search in first half
  - if greater than middle elem, search in last half
- If search list is narrowed down to 0 elements, return -1
- Divide and conquer style algorithm
**Binary Search**

**Iterative version**

```c
int binarySearch(int array[], int size, int value) {
    int first = 0,         // index of First array element
    last = size - 1,       // index of Last array element
    middle,                // index of Mid point of search
    position = -1;         // index of search value, when found
    bool found = false;    // Flag
    while (!found && first <= last) {
        middle = (first + last) / 2;     // Calculate mid point
        if (array[middle] == value) {    // If value is found at mid
            found = true;
            position = middle;
        } else if (array[middle] > value)  // If value is in lower half
            last = middle - 1;
        else
            first = middle + 1;           // If value is in upper half
    }
    return position;
}
```

**Example**

The target of your search is 42. Given the following array of integers, record the values stored in the variables named `first`, `last`, and `middle` during each iteration of a binary search.

<table>
<thead>
<tr>
<th>values:</th>
<th>1</th>
<th>7</th>
<th>8</th>
<th>14</th>
<th>20</th>
<th>42</th>
<th>55</th>
<th>67</th>
<th>78</th>
<th>101</th>
<th>112</th>
<th>122</th>
<th>170</th>
<th>179</th>
<th>190</th>
</tr>
</thead>
<tbody>
<tr>
<td>indexes:</td>
<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>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>first</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>last</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>middle</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Repeat the exercise with a target of 82:

<table>
<thead>
<tr>
<th>values:</th>
<th>1</th>
<th>7</th>
<th>8</th>
<th>14</th>
<th>20</th>
<th>42</th>
<th>55</th>
<th>67</th>
<th>78</th>
<th>101</th>
<th>112</th>
<th>122</th>
<th>170</th>
<th>179</th>
<th>190</th>
</tr>
</thead>
<tbody>
<tr>
<td>indexes:</td>
<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>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>first</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>last</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>middle</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Recursive version**

```c
int binarySearchRec(int array[], int first, int last, int value) {
    if (first > last)           // check for empty list (base case)
        return -1;
    int middle = (first + last)/2;  // compute middle index
    if (array[middle] == value) {  // recursion
        return middle;
    } else if (value < array[middle]) // recursion
        return binarySearchRec(array, first, middle-1, value);
    else
        return binarySearchRec(array, middle+1, last, value);
}

int binarySearch(int array[], int size, int value) {
    return binarySearchRec(array, 0, size-1, value);
}
```

- Convert the iterative version to recursive
- What is the base case?
  - empty list: result = -1 (not found)
- What is the recursive case?
  - split list into: middle value, first half, last half
  - if target == middle value, then return its index
  - if target < middle elem, search in first half
  - if target > middle elem, search in last half
- Need to add parameters for first and last index of the current subpart of the list to search.
Binary Search
Running time efficiency

- What is the Big-O analysis of the running time?
- N is the length of the list to search
- Worst case: keep dividing N by 2 until it is less than 1.
- This is equivalent to doubling 1 until it gets to N.
- Example: \( N = 64 \):
  
  \[
  \begin{align*}
  1 \times 2 &= 2 \\
  2 \times 2 &= 4 \\
  4 \times 2 &= 8 \\
  8 \times 2 &= 16 \\
  16 \times 2 &= 32 \\
  32 \times 2 &= 64
  \end{align*}
  \]
  
  After 6 steps we have \( 2^6 \).

  After \( k \) steps we have \( 2^k \).

Binary Search
Running time efficiency

- How many steps does it take to double 1 and get to \( N \)?
  \[
  2^k = N
  \]
- How do we solve that for \( k \)?
- Definition of logarithm (see math textbook):
  \[
  \log_B N = k \text{ if } B^k = N \]
- So solving for \( k \):
  \[
  k = \log_2 N
  \]

Binary Search
Running time efficiency

- How many steps does it take to repeatedly double 1 and get to \( N \)?
  \[
  \log_2 N
  \]
- How many steps does it take to repeatedly divide \( N \) by 2 and get to 1?
  \[
  \log_2 N
  \]
- Since (worst case) binary search repeatedly divides the length of the list by 2, until it gets down to one, its running time is
  \[
  O(\log N)
  \]