Recursion

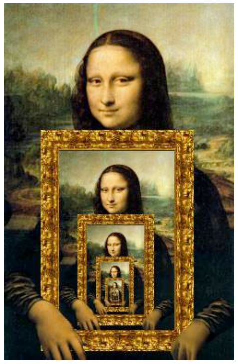
Chapter 8

CS 3358
Summer I 2012

Jill Seaman

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?

```
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.
...
```

Recursive message() modified

- How about this one?

```
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:

```
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
- 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 added to 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
```

- What is the base case for n?
- If we assume (n-1)! can be computed, how can we get n! from that?

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
```

- What is the base case for n?
  - 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

factorial

```
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:

```
factorial(4):
  return 4 * factorial(3);
  calls factorial(3):
    return 3 * factorial(2);
    calls factorial(2):
      return 2 * factorial(1);
      calls factorial(1):
        return 1 * factorial(0);
        calls factorial(0):
          return 1;
```

- each return statement must wait for the result of the recursive call to compute its result

Tracing the calls

- Calls to factorial:

```
factorial(4):
  return 4 * factorial(3);    =4 * 6 = 24
  calls factorial(3):
    return 3 * factorial(2);    =3 * 2 = 6
    calls factorial(2):
      return 2 * factorial(1);    =2 * 1 = 2
      calls factorial(1):
        return 1 * factorial(0);  =1 * 1 = 1
        calls factorial(0):
          return 1;
```

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

Recursive functions over lists

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

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

- 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?

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?
  - sum (list) = sum (list[0..n-2]) + list[n-1]

Recursive function example
sum of a list: array

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

For a list with size = 4:

```
      a[3] + sum(a,3)=
      a[3] + a[2] + sum(a,2)=
      a[3] + a[2] + a[1] + sum(a,1)=
      a[3] + a[2] + a[1] + a[0] +sum(a,0)=
      a[3] + a[2] + a[1] + a[0] + 0
```


Recursive function example
sum of a list: vector

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

- Aren't we changing the vector argument each time?
  - No (why not?)
  - So something else bad is happening each time.

Recursive function example
sum of a list: vector

```
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> & v) {
   vector<int> v1 (v); //make one copy
   return sumRec(v1);
}
```

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

Recursive function example
sum of a list: linked list

- Add a sum function to list_3358_pointers.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);
   }
}
```

Summary of the list examples

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

	Array	Vector	Linked list
Base case	`size==0`	`v.size()==0`	`p==NULL`
last(or first) element	`a[size-1]`	`v.back()`	`p->value` (first element)
shorter list	`use size-1`	`v.pop_back()*`	`p->next`

*may need to copy original vector

Recursive function example
count characters in a string

- Recursive function to count the number of times a specific character appears in a string
- We will use the string member function substr to make a smaller string
  - str.substr (int pos, int length);
  - pos is the starting position in str
  - length is the number of characters in the result

```
string x = “hello there”;
cout << s.substr(3,5);
```

lo th

Recursive function example
count characters in a string

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

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) = y              if y divides x evenly, otherwise
gcd(x,y) = gcd(y,remainder of x/y)
```

- It's a recursive definition
- If x < y, then x % y is x.
- Keeps the larger number in x

Recursive function example
greatest common divisor

- Code:

```
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;
}
```


Recursive function example
greatest common divisor

- 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:

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

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:

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

int main() {
    cout << "The first 14 fibonacci numbers: " << endl;
    for (int i=0; i<14; i++) {
        int count = 0;
        int x = fib(i,count);
        cout << "fib (" << i << ")= " << x
             << " # of recursive calls to fib = " << count << endl;
    }
}
```


Recursive function example
Fibonacci numbers

- Counting calls to fib: output

```
The first 14 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
...
fib (40)= 102,334,155  # of recursive calls to fib = 331,160,281
```


Recursive function example
Fibonacci numbers

- Why are there so many calls to fib?

```
fib(n) calls fib(n-1) and fib(n-2)
```

- Say it computes fib(n-2) first.
- When it computers fib(n-1), it computes fib(n-2) **again**

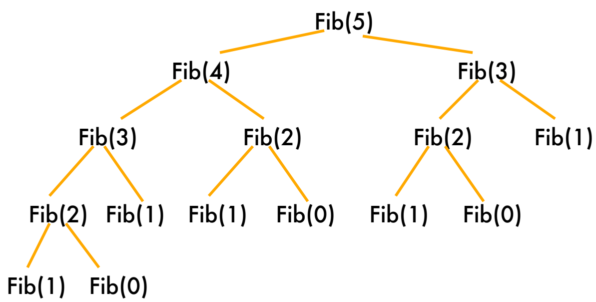
```
fib(n-1) calls fib((n-1)-1) and fib((n-1)-2)
             = fib(n-2)     and fib (n-3)
```

- 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 fib(5)



Recursive function example

Fibonacci numbers

- The number of recursive calls is
  - larger than the Fibonacci number we are trying to compute
  - exponential, in terms of n
- Never solve the same instance of a problem in separate recursive calls.
  - make sure f(m) is called only once for a given m

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

```
int binarySearch(const int array[], int size, int value)
{
    int first = 0,          // First array element
    last = size - 1,        // Last array element
    middle,                 // Mid point of search
    position = -1;          // Position of search value
    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;
}
```

Binary Search
Example

The target of your search is 42. Given the following list of integers, record the values of first, last, and middle during a binary search. Assume the following numbers are in an array.

1 7 8 14 20 42 55 67 78 101 112 122 170 179 190

Repeat the exercise with a target of 82

Binary Search
Recursive version

```
int binarySearchRec(const int array[], int first, int last, int value)
{
    int middle; // Mid point of search

    if (first > last)
        return -1;
    middle = (first + last)/2;
    if (array[middle]==value)
        return middle;
    if (array[middle]<value)
        return binarySearchRec(array, middle+1,last,value);
    else
        return binarySearchRec(array, first,middle-1,value);
}

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


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.

```
1*2 = 2
2*2 = 4
4*2 = 8
8*2 = 16
16*2 = 32
32*2 = 64
```

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:

$\log_B N = k$ if $B^k = N$

The logarithm is the exponent

- 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

  $O(\log N)$