Recursion

EECS 233

Non-Recursive Programming

- Non-recursive programming is more familiar.
- Calculate Σi, $i=1,\ldots,n$ using a “for” loop

```
int sum(int n) {
    int i, sum;
    sum = 0;
    for (i = 1; i <= n; i++)
        sum += i;
    return sum;
}
```

What is Recursion?

- A *recursive* method is a method that invokes itself.

```
int sum(int n) {
    if (n <= 0)
        return 0;

    int sum_result = n + sum(n – 1);
    return sum_result;
}
```

```
sum(6)      = 6 + sum(5)
            = 6 + 5 + sum(4)
            = …
```

How Does Recursion Work?

- A recursive method solves a larger problem by reducing it to smaller and smaller sub-problems.
- We keep doing this until we reach a sub-problem that is trivial to solve directly. This is known as the *base case*.

```
int sum(int n) {
    if (n <= 0)                   // base case
        return 0;
    int sum = n + sum(n – 1);     // recursive call
    return sum;
}
```

- The *base case* stops the recursion.
- If the base case hasn't been reached, we:
  - make one or more recursive calls to solve smaller problems
  - use the solutions to the smaller problems to solve the original problem

Tracking Recursive Calls

- Assume the main() method calls x = sum(3). What happens?

```
int sum(int n) {
        if (n <= 0)                          // base case
                return 0;
        int sum_res = n + sum(n – 1);        // recursive call
        return sum_res;
}
main() {
      … x = sum(3); …
}
```

main() calls sum(3)
    sum(3) calls sum(2)
        sum(2) calls sum(1)
            sum(1) calls sum(0)
                sum(0) returns 0
            sum(1) returns 1 + 0, or 1
        sum(2) returns 2 + 1, or 3
    sum(3) returns 3 + 3, or 6
main() assigns 6 to x

n=0
sum’s return addr in “sum”
n=1
sum’s return addr in “sum”
n=2
sum’s return addr in “sum”
n=3
sum’s return addr in “main”
x
args

How To Design A Recursive Method?

- Basic structure:

```
recursiveMethod (arguments) {
    If (stopping condition)          // base case
        … // handle the base case

    else {                           // recursive case
        … // possibly do something here
        recursiveMethod (modified arguments);
        … // possibly do something here
    }
}
```

- When we make the recursive call, we typically use arguments that bring us closer to the base case.
  - example: sum(n – 1) brings us one step closer to n = 0

- We must ensure that the method will terminate, regardless of the initial input. Otherwise, we can get "infinite" recursion!

Example 1: Counting the Occurrences of a Character in a String

- For example, there are three occurrences of “c” in “Occurrences”

- Thinking recursively:
  - How can we break this problem down into a smaller sub-problem(s)?
  - What is the base case(s)?
  - Do we need to combine the solutions to the sub-problems? If so, how should we do so?

```
int occurrences (String s, char c) {
        if (s.length() == 0) return 0;
        if (s.charAt(0) == c) return 1+occurrences(s.subString(1), c);
        else return 0+occurrences(s.subString(1), c);
}
```

where

length(): length of the string
charAt(i): the character at index i
subString(i): the substring starting at index i

Example 2: Reversing An Array

- How can we use a *recursive* method to reverse an array of integers “in place” – modifying the original array?

8	23	43	57	37	15	19

becomes

19	15	37	57	43	23	8

- Thinking recursively:
  - How can we break this problem down into one or more smaller sub-problems?
  - What is the base case(s)?
  - Do we need to combine the solutions to the sub-problems? If so, how should we do so?

Example 2: Reversing An Array (cont.)

```
void reverse(int[ ] arr, int left, int right) {
        if (left >= right)
                return;              // base case

        // Swap the “ends”: arr[left] and arr[right].
        int tmp = arr[left];
        arr[left] = arr[right];
        arr[right] = tmp;

        // Reverse the “middle.”
        reverse(arr, left + 1, right – 1);
}
```

arr->	8	23	43	57	37	15	19
	left						right

becomes

arr->	19	23	43	57	37	15	8
		left				right	

Tracking the Recursive Calls

```
void reverse(int[ ] arr, int left, int right) {
    if (left >= right)
        return;         // base case

    // Swap the "ends": arr[left] and arr[right].
    int tmp = arr[left];
    arr[left] = arr[right];
    arr[right] = tmp;

    // Reverse the "middle."
    reverse(arr, left + 1, right – 1);
}
```

arr->	8	23	43	57	37	15	19
	left						right

becomes

arr->	19	23	43	57	37	15	8
		left				right	

```
reverse(arr, 0, 6)
    swap arr[0] and arr[6]
    reverse(arr, 1, 5)
        swap arr[1] and arr[5]
        reverse(arr, 2, 4)
            swap arr[2] and arr[4]
            reverse(arr, 3, 3)
                base case reached (3 >= 3),
                so return.
```

Example 3: Finding a Number in a Phonebook

- Recall the binary search algorithm described last week.

```
findNumber(person, phonebook_size) {
    low = 0
    high = phonebook_size
    while (low <= high) {
            P = floor((low + high) / 2)
            Compare the P-th person in the array and person
            if the same
                    return the corresponding number
            else if the person's name comes earlier in the book
                    high = P – 1
            else
                    low = P + 1
    }
    return NOT_FOUND
}
```

- As we mentioned, this is an example of *binary search*. It has an elegant implementation using recursion.

Recursive Binary Search

- Binary Search Using Recursion. Let's write the method together:

```
findNumber(person, low, high) {
    if (low > high) return NOT_FOUND      // base case 1: not found

    P = floor((low + high) / 2)
    Compare the P-th person in the array and person
    if the same                           //  base case 2: found it
            return the corresponding number
    else if the person's name comes earlier in the book
            findNumber(person, low,P-1)
    else
            findNumber(person, P+1,high)
}
```

- Note that we add two parameters to the method. The initial call would be findNumber(person, 0, 999999), for 1000000 phone numbers in the phonebook.

Recursion vs. Iteration

- Some algorithms are easy to implement using recursion.
  - Examples we've seen
- Recursion is a bit more costly because of the overhead involved in invoking a method (need to allocate stack frames for method calls).
- Recursive methods can often be easily converted to a non-recursive method that uses iteration.
- Rule of thumb: None!
  - if it's easier/faster to solve a problem recursively, use it
  - otherwise, use iteration

Example 4: Calculating the Fibonacci Numbers

- The Fibonacci Sequence
  - The sequence of numbers with a recursive definition:
    - ✓ $fib_1 = 1$
    - ✓ $fib_2 = 1$
    - ✓ $fib_n = fib_{n-1} + fib_{n-2}$

- Here's the start of the sequence:

  1, 1, 2, 3, 5, 8, 13, 21, …

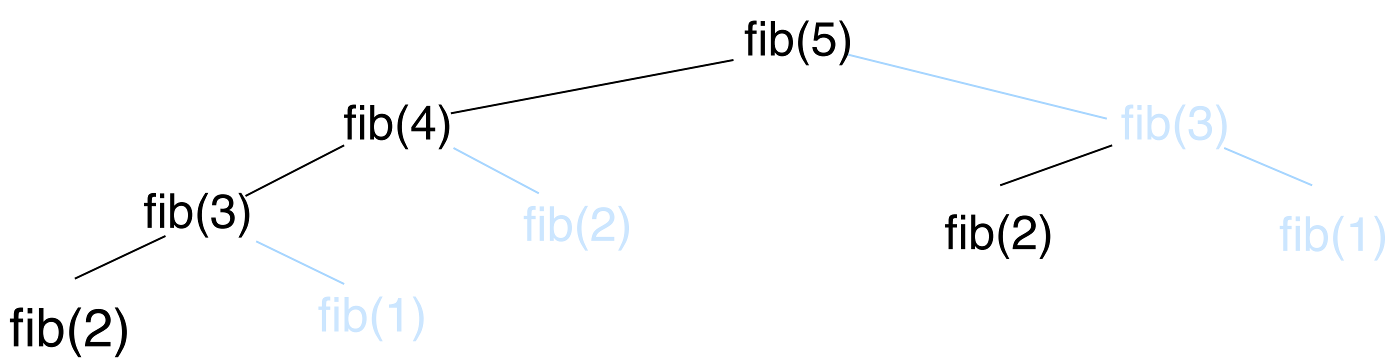
Solution using Recursion

- Recursive definition:
  - $fib_1 = 1$
  - $fib_2 = 1$
  - $fib_n = fib_{n-1} + fib_{n-2}$
- Recursive method for computing fib_n:

```
int fib(int n) {
        if (n <= 0)
            throw an exception
        else if (n == 1 || n == 2)
            return 1;
        else
            ? return fib(n-1)+fib(n-2)
}
```

Tracking the Recursive Calls

- When a recursive method makes more than one recursive call, it can be helpful to draw a *call tree* that traces the method calls.
  - Call tree for fib(5):



- The method makes multiple calls with the same argument.
- As n increases, the number of method calls made to compute fib(n) grows exponentially!

Recursive Fibonacci is extremely inefficient!

Solution using Iteration

- It's possible to write a more efficient Fibonacci function using recursion.

- However, it's easier to do so using iteration:

```
int fib(int n) {
        if (n <= 0)
            throw an exception
        int oneBefore = 1;
        int twoBefore = 1;
        int current = 1;          // answer for n == 1 or n == 2
        for (int i = 3; i <= n; i++) {
            current = oneBefore + twoBefore;
            twoBefore = oneBefore;
            oneBefore = current;
        }
        return current;
}
```

- This algorithm computes a given Fibonacci number only once.
- It keeps track of the two most recent Fibonacci numbers and uses them to compute subsequent ones.

Algorithm Analysis

- Different algorithms for the same problem can have *drastically* different complexity.
  - Using recursion: exponential time (# of recursive calls)
  - Using iteration: linear time (n calculations)

- To compare different algorithms, we need to analyze them
  - Running time
  - Memory space requirement
  - Fault tolerance
  - Number of messages between participating hosts

Relative Growth Rates

- Examples (which grows faster?)
  - 1000000 versus 0.01*sqrt(N)
  - log(N) versus sqrt(N)
  - Nlog(N) versus $N^{1.001}$
  - N^3 versus 10000*N^2
  - $\log^2(N)$ versus 10*log(N^5)
  - 2*$\log_2(N)$ versus $\log_3(N)$
  - N*2^N versus 3^N

Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Common terms for function growth rate

Function Growth Rates: MathematicalDefinitions

Consider positive functions T(N) and f(N).

- $T(N) = \boldsymbol{O}(f(N))$ if there are positive constants c and n_0 such that

  $T(N) \le cf(N)$ for all $N \ge n_0$

  - Example: $10*N^2+10000 = \boldsymbol{O}(N^3)$

- $T(N) = \boldsymbol{\Omega}(f(N))$ if there are positive constants c and n_0 such that

  $T(N) \ge cf(N)$ for all $N \ge n_0$

  - Example: $0.0001*N^3 = \boldsymbol{\Omega}(N^2)$

- $T(N) = \boldsymbol{\theta}(f(N))$ iff $T(N) = \mathbf{O}(f(N))$ and $T(N) = \boldsymbol{\Omega}(f(N))$
  - Example: $0.001*N^2+10000*N = \boldsymbol{\theta}(N^2)$

- $T(N) = \boldsymbol{o}(f(N))$ if for *all* constants c there exists an n_0 such that

  $T(N) < cf(N)$ for all $N > n_0$.

  or

  $T(N) = \boldsymbol{o}(f(N))$ iff $T(N) = \boldsymbol{O}(f(N))$ and $T(N) \neq \boldsymbol{\Omega}(f(N))$

  - Example: $10000*N*\log(N) = \boldsymbol{o}(N^2)$