Recursion
Mathematical Background

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)

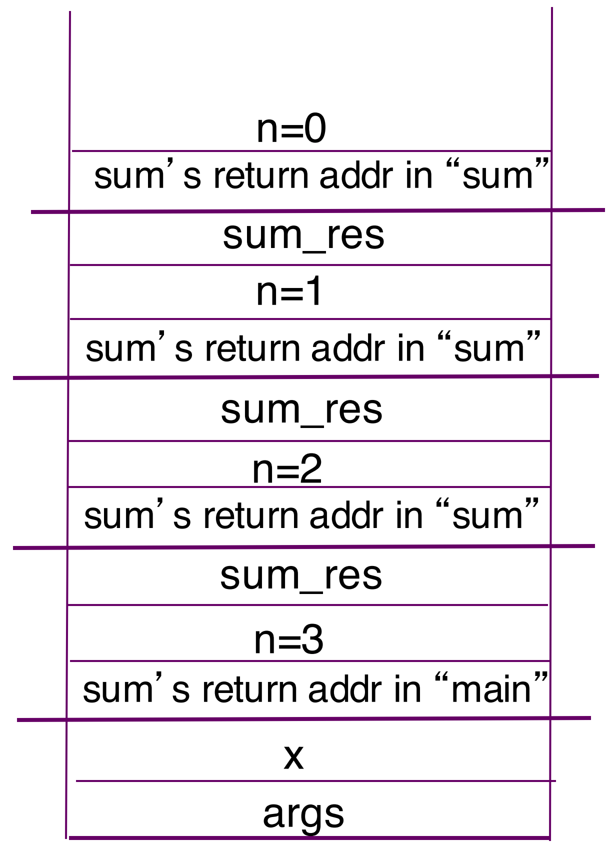
sum(0) returns 0

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



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 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

- Can we 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 myReverse(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."
        myReverse(arr, left + 1, right – 1);
}
void reverse(int[] arr){
       myReverse(arr, 0, arr.length - 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 myReverse(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."
        myReverse(arr, left + 1, right – 1);
}
void reverse(int[] arr){
        myReverse(arr, 0, arr.length);
}
```

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

```
myReverse(arr, 0, 6)
        swap arr[0] and arr[6]
        myReverse(arr, 1, 5)
                swap arr[1] and arr[5]
                myReverse(arr, 2, 4)
                        swap arr[2] and arr[4]
                        myReverse(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 (pseudocode).

```
findNumber(person) {
    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
}
```

Recursive Binary Search

- Binary Search Using Recursion. (Pseudo-code)

```
myFindNumber(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
        ?myFindNumber(person, low,P-1)
    else
        ?myFindNumber(person, P+1,high)
}
findNumber(person){
    myFindNumber(person, 0, phonebook_size – 1)
}
```

- Note that we add two parameters to the method.

Recursion vs. Iteration

Recursive method

```
myFindNumber(person, low, high) {
    if (low > high) return NOT_FOUND
    P = floor((low + high) / 2)
    if (person == names[P])
        return numbers[P]
    else if (person < names[P])
      myFindNumber(person, low,P-1)
    else
      myFindNumber(person, P+1,high)
}
findNumber(person){
   myFindNumber(person, 0, 999999)
```

Iterative Method

```
findNumber(person) {
    low = 0
    high = 999999

    while (low <= high) {
        P = floor((low + high) / 2)
        if (person == names[P])
            return numbers[P]
        else
            if (person < names[P])
                high = P – 1
            else
                low = P + 1
    }
    return NOT_FOUND
}
```

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) and *takes more stack memory*

- Recursive methods can often be easily converted to a non-recursive method that uses iteration.

- Rule of thumb: None!
  - if it's easier/faster and feasible to solve a problem recursively, use it
    - Unless there is danger of running out of memory!
  - otherwise, use iteration

Example 4: Calculating the Fibonacci Numbers

- The Fibonacci Sequence
  - Recursive definition:
    - $fib_1 = 1$
    - $fib_2 = 1$
    - $fib_n = fib_{n-1} + fib_{n-2}$
- The start of the sequence:

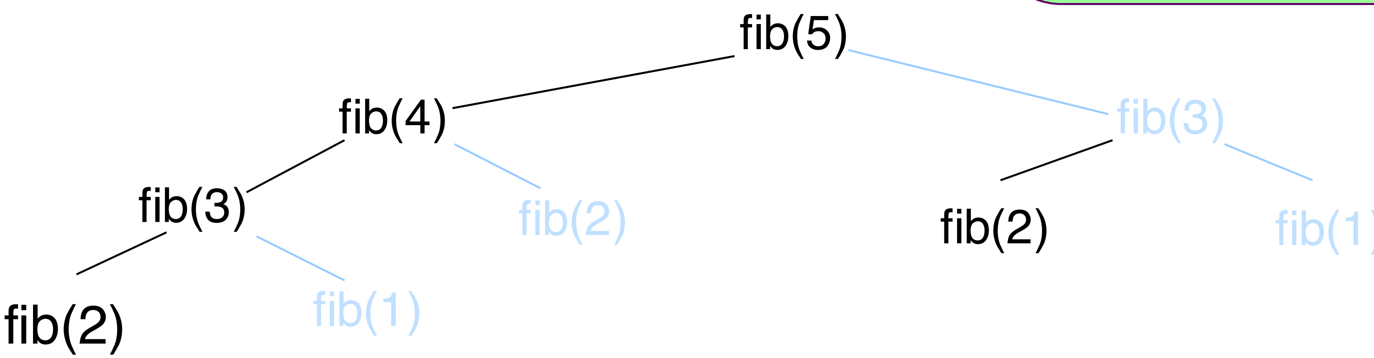
  1, 1, 2, 3, 5, 8, 13, 21, …

- Recursive method for computing fib_n:

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

Tracking the Recursive Calls

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



- A sequence of calls form a *call tree.*
- 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

- The Fibonacci Sequence
  - Recursive definition:
    - $fib_1 = 1$
    - $fib_2 = 1$
    - $fib_n = fib_{n-1} + fib_{n-2}$
- The start of the sequence:

  1, 1, 2, 3, 5, 8, 13, 21, …

- A more efficient Fibonacci function:

```
int fib(int n) {
        if (n <= 0)
            throw an exception
        int oneBefore = 1;
        int twoBefore = 1;
        int current = 1; // answer for n = 1 or  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.
  - Fibonacci using recursion: exponential time (# of recursive calls)
  - Fibonacci 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