RECURSION AND LOGARITHMS

EECS 2011

Recursion

- In some problems, it may be natural to define the problem in terms of the problem itself.
- Recursion is useful for problems that can be represented by a simpler version of the same problem.
- Example: the factorial function

```
6! = 6 * 5 * 4 * 3 * 2 * 1
```

We could write:

```
6! = 6 * 5!
```

Recursion (cont.)

- Recursion is one way to decompose a task into smaller subtasks. At least one of the subtasks is a smaller example of the same task.

- The smallest example of the same task has a non-recursive solution.

- Example: the factorial function

```
n! = n*(n-1)! and 1! = 1
```

Example: Factorial Function

- In general, we can express the factorial function as follows:

  `n! = n*(n-1)!`

  Is this correct? Well… almost.

- The factorial function is only defined for *positive* integers. So we should be more precise:

```
f(n) = 1              if n = 1
     = n*f(n-1)       if n > 1
```

Factorial Function: Pseudo-code

```
int recFactorial( int n ){
    if( n == 1 )
          return 1;
    else
      return n * recFactorial( n-1 );
}
```

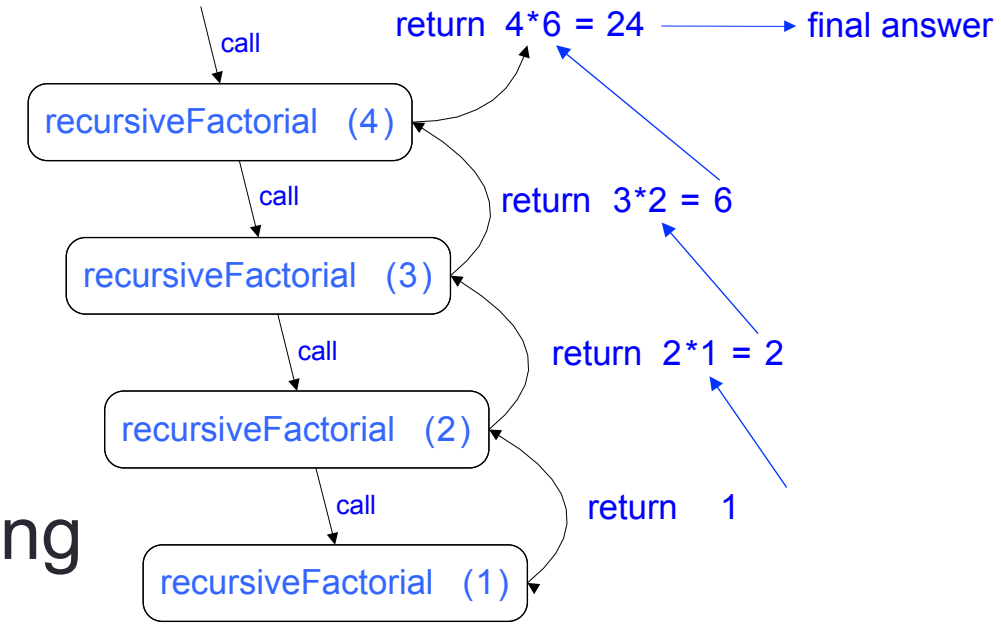
recursion **means that a function calls itself.**

Visualizing Recursion

Recursion trace

- A box for each recursive call
- An arrow from each caller to callee
- An arrow from each callee to caller showing return value

Example recursion trace:



Recursive vs. Iterative Solutions

- For certain problems (such as the factorial function), a recursive solution often leads to short and elegant code. Compare the recursive solution with the iterative solution:

```
int fac(int numb) {
 if (numb == 1)
   return 1;
 else
   return
      (numb*fac(numb-1));
}
```

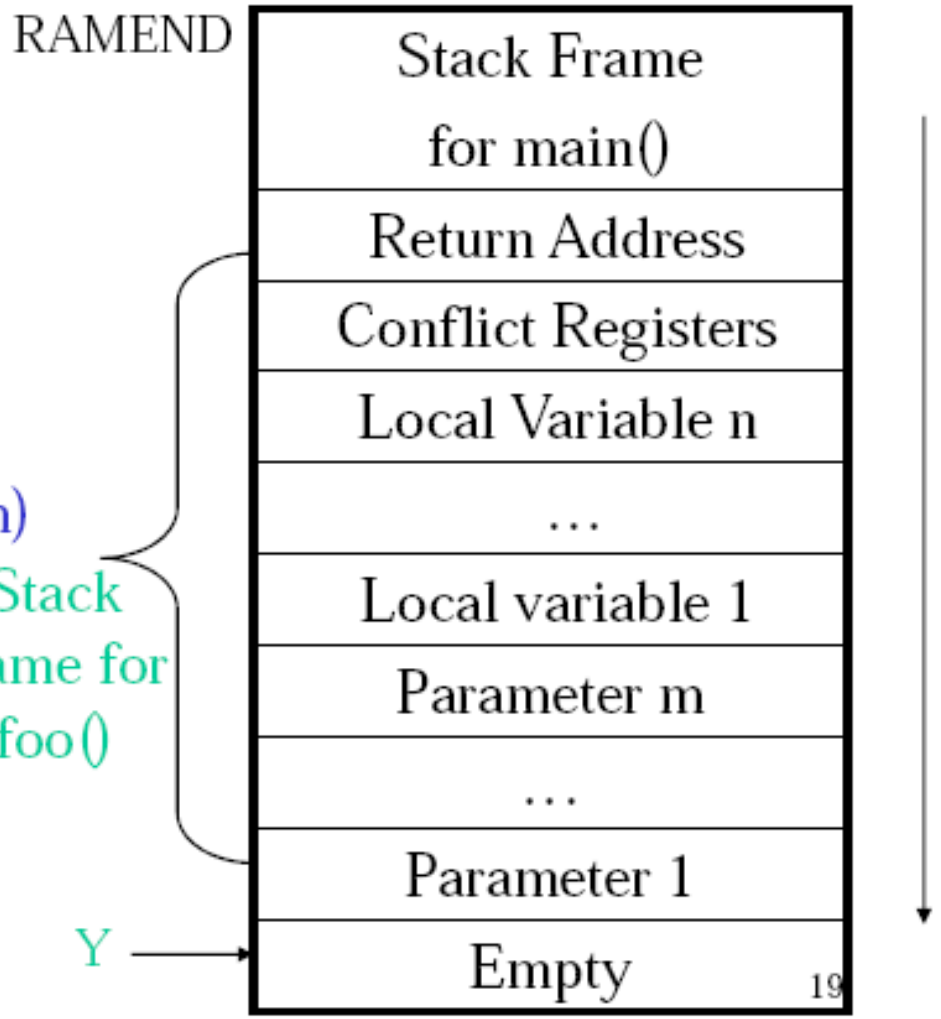
```
int fac(int numb){
  int product = 1;
  while(numb > 1){
    product *= numb;
    numb--;
  }
  return product;
}
```

A Word of Caution

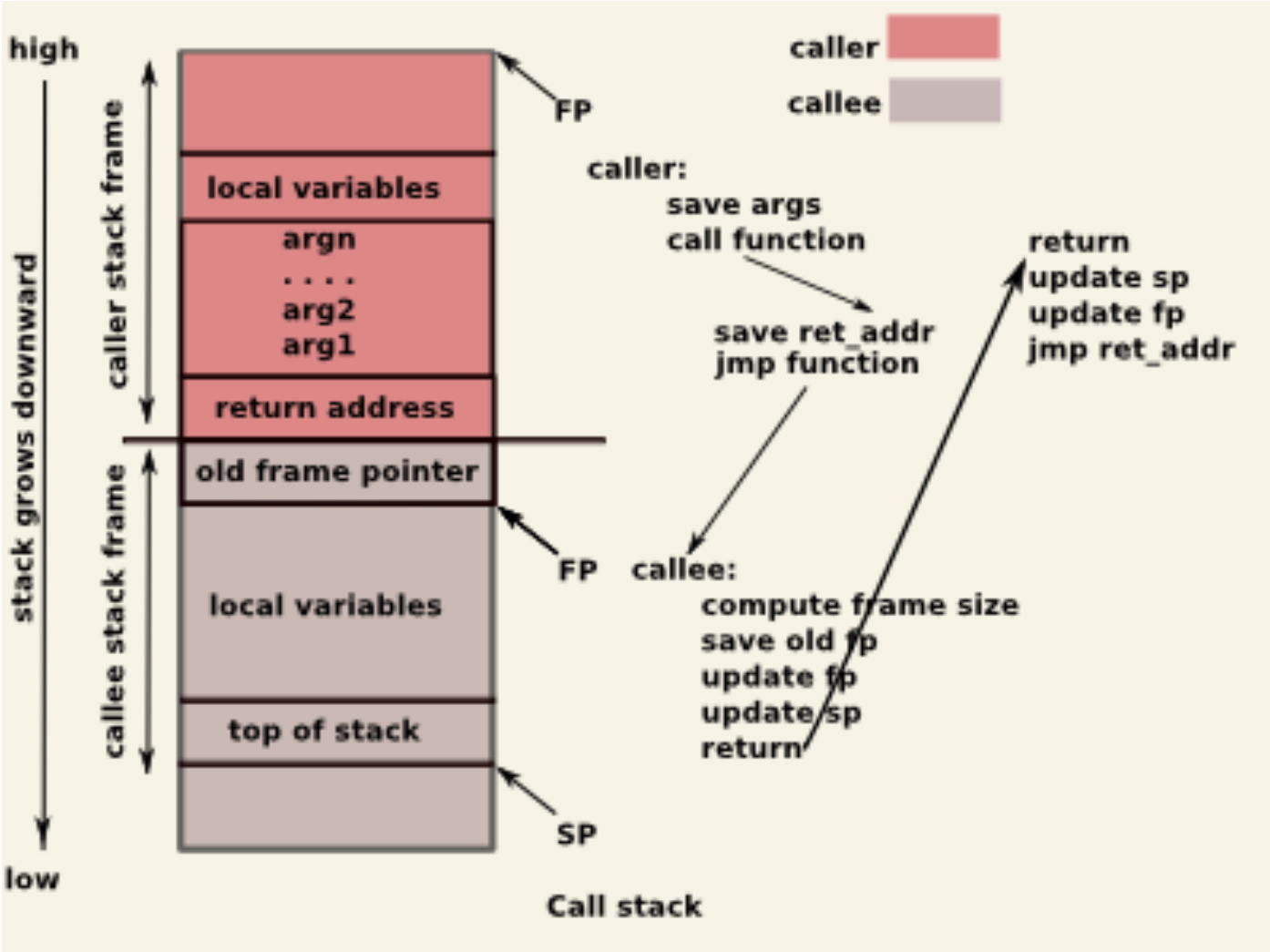
- To trace recursion, function calls operate as a stack – the new function is put on top of the caller.
- We have to pay a price for recursion:
  - calling a function consumes more time and memory than adjusting a loop counter.
  - high performance applications (graphic action games, simulations of nuclear explosions) hardly ever use recursion.
- In less demanding applications, recursion is an attractive alternative for iteration (for the right problems!)

Function Call Stack: Example

```
int main(void)
{ …
  foo(arg1, arg2, …, argm);
}
 void foo(arg1, arg2, …, argm)
{ int var1, var2, …, varn;
  …
}
```



Function Call Stack

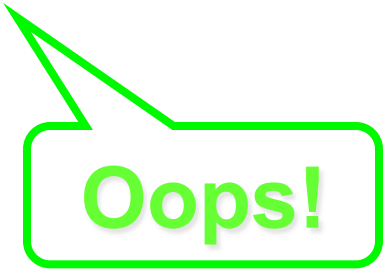


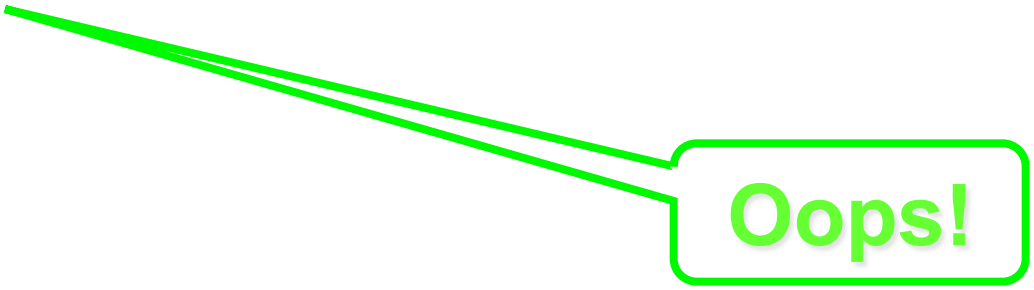
Infinite Loops

If we use iteration, we must be careful not to create an infinite loop by accident.

```
for (int incr=1; incr!=10; incr+=2)
   ...

int result = 1;
while(result > 0){
   ...
   result++;
}
```





Infinite Recursion

Similarly, if we use recursion, we must be careful not to create an infinite chain of function calls.

```
int fac(int numb){
    return numb * fac(numb-1);
}
```

Oops!
No termination condition

```
int fac(int numb){
    if (numb == 1)
        return 1;
    else
        return numb * fac(numb + 1);
}
```

Oops!

Tips

We must always make sure that the recursion *bottoms out*:

- A recursive function must contain at least one non-recursive branch (base case).
- The recursive calls must eventually lead to a non-recursive branch (base case).

General Form of Recursion

- How to write recursively?

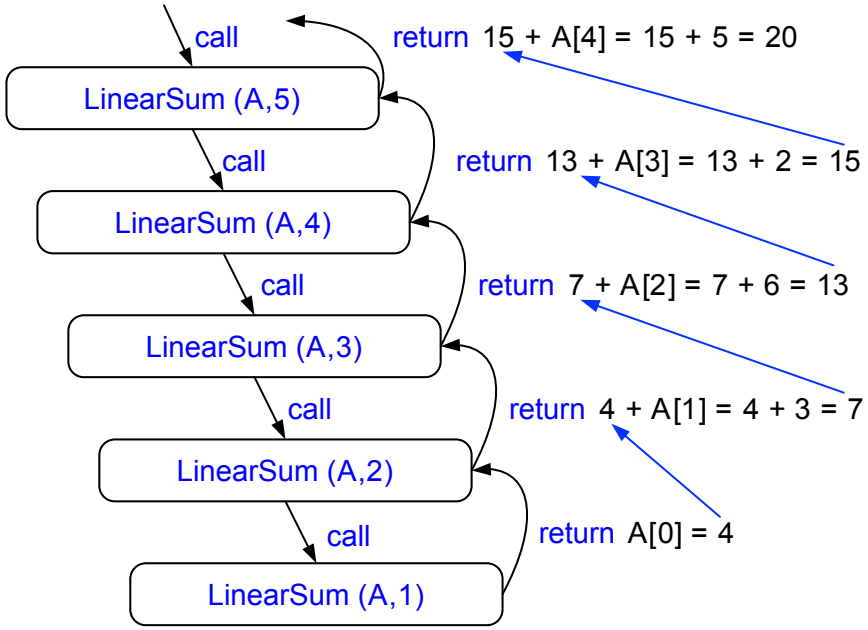
```
int recur_fn( parameters ){
   if ( stopping_condition )     // base case
        return stopping_value;
   if ( stopping_condition_2 )   // base case 2
        return stopping_value_2;
 return recur_fn( revised_parameters )
}
```

Example: Sum of an Array

Algorithm LinearSum(*A*, *n*):
Input:
A integer array *A* and an integer *n* ≥ 1, such that *A* has at least *n* elements
Output:
Sum of the first *n* integers in *A*

```
if n = 1 then
  return A[0];
else
  return LinearSum(A, n - 1)
           + A[n - 1];
```

Example recursion trace:



Example: Reversing an Array

Algorithm ReverseArray(*A*, *i*, *j*):

Input: An array *A* and nonnegative integer indices *i* and *j*

Output: The reversal of the elements in *A* starting at index *i* and ending at *j*

```
if i < j then
    swap A[i] and A[ j];
    ReverseArray( A, i + 1, j – 1 );
return
```

Defining Arguments for Recursion

- In creating recursive methods, it is important to define the methods in ways that facilitate recursion.
- This sometimes requires we define additional paramaters that are passed to the method.
- For example, we defined the array reversal method as ReverseArray(*A, i, j*), not ReverseArray(*A*).

Linear Recursion

- The above 2 examples use **linear recursion.**
  - sum of an array
  - reversing an array

Linear Recursion (2)

- **Test for base cases.**
  - Begin by testing for a set of base cases (there should be at least one).
  - Every possible chain of recursive calls **must** eventually reach a base case, and the handling of each base case should not use recursion.
- ***Recur once.***
  - Perform a single recursive call. (This recursive step may involve a test that decides which of several possible recursive calls to make, but it should ultimately choose to make just one of these calls each time we perform this step.)
  - Define each possible recursive call so that it makes progress towards a base case.

Tail Recursion

- Tail recursion occurs when a linearly recursive method makes its recursive call as its last step.
- The array reversal method is an example.
- Such methods can be easily converted to non-recursive methods (which saves on some resources).
- Example: reversing an array

Algorithm IterativeReverseArray(*A, i, j*):
Input: An array *A* and nonnegative integer indices *i* and *j*
Output: The reversal of the elements in *A* starting at index *i* and ending at *j*
while *i* < *j* **do**
Swap *A*[*i*] and *A*[*j*]
i = *i* + 1
j = *j* - 1
return

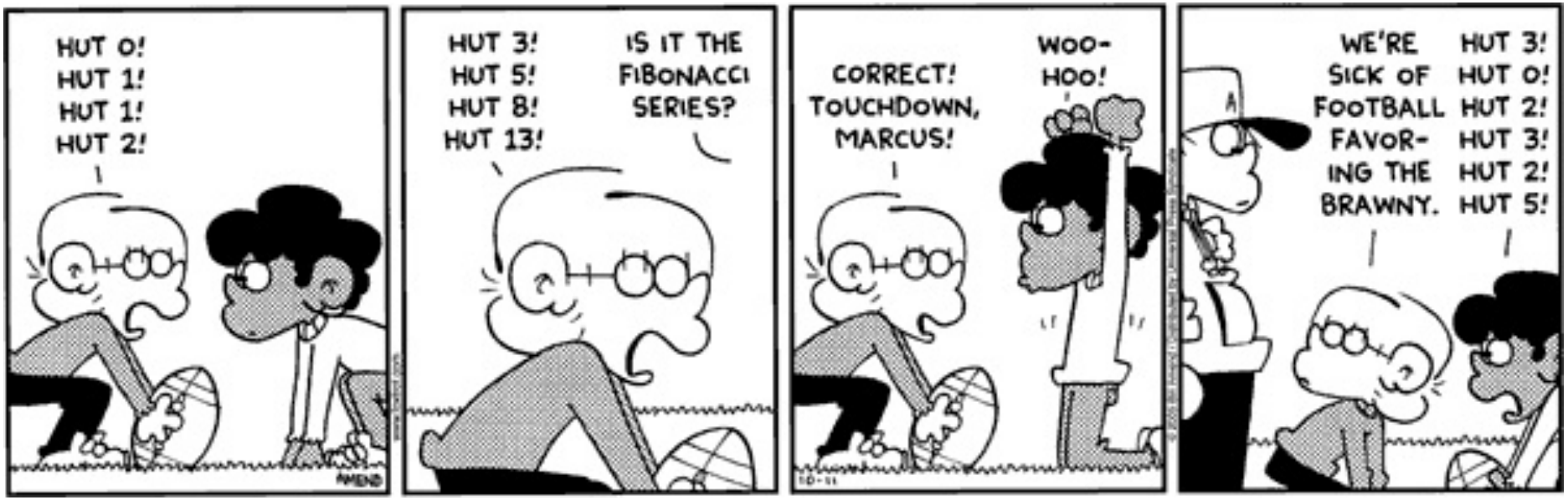
Binary Recursion

- Binary recursion occurs whenever there are **two** recursive calls for each non-base case.

- Example: Fibonacci sequence

  $f(1) = f(2) = 1$

  $f(n) = f(n-1) + f(n-2)$ if $n > 2$

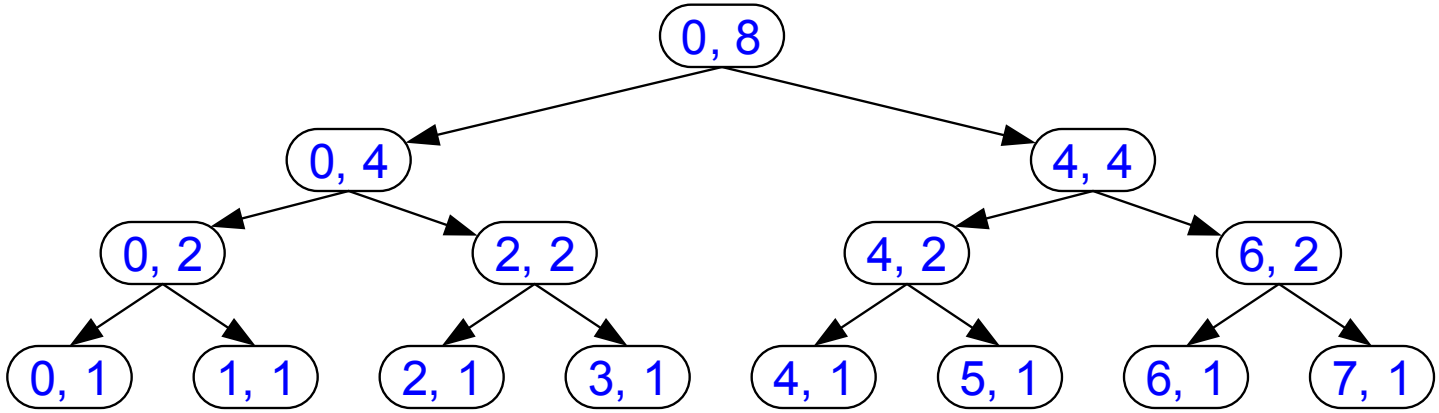

HUT 0!
HUT 1!
HUT 1!
HUT 2!
HUT 3!
HUT 5!
HUT 8!
HUT 13!
IS IT THE FIBONACCI SERIES?
CORRECT! TOUCHDOWN, MARCUS!
WOO-HOO!
WE'RE SICK OF FOOTBALL FAVOR-ING THE BRAWNY.
HUT 3!
HUT 0!
HUT 2!
HUT 3!
HUT 2!
HUT 5!

Another Binary Recusive Method

- Problem: add all the numbers in an integer array A:

Algorithm BinarySum(*A, i, n*):
Input: An array *A* and integers *i* and *n*
Output: The sum of the *n* integers in *A* starting at index *i*
if *n* = 1 **then**
return *A*[*i*];
return BinarySum(*A, i, n*/ 2) + BinarySum(*A, i* + *n*/ 2, *n*/ 2);

- **Example trace:** array A has 8 elements



Recursion: Checklist

- Do I have a base case (base cases)?
  - could be implicit (e.g., simply exit the function)
- Do I have a recursive call (recursive calls)?
- Do I "adjust" the argument(s) of the recursive call(s) correctly?
- Can the recursive call(s) eventually reach the base case(s)?
- Do I write the first call (e.g., in `main()`) correctly?

Multiple Recursion

- Multiple recursion: makes potentially many recursive calls (not just one or two).

- Not covered in this course.

Running Time of Recursive Methods

- Could be just a hidden “for" or “while” loop.
  - See “Tail Recursion” slide.
  - “Unravel” the hidden loop to count the number of iterations.
  - Example: sum of an array, reversing an array
- Logarithmic (next)
  - Examples: binary search, exponentiation, GCD
- Solving a recurrence
  - Example: merge sort (next lecture)

LOGARITHMS

EECS 2011

Logarithmic Running Time

- An algorithm is O(logN) if it takes constant (O(1)) time to cut the problem size by a fraction (e.g., by ½).

- An algorithm is O(N) if constant time is required to merely reduce the problem by a constant amount (e.g., by 1).

Example: Binary Search

- Search for an element in a *sorted* array
  - Sequential search
  - Binary search

- Binary search
  - Compare the search element with the middle element of the array.
  - If not equal, then apply binary search to half of the array (if not empty) where the search element would be.

Binary Search

```
int binarySearch (int[] a, int x)
{
/*1*/    int low = 0, high = a.size() - 1;
/*2*/    while (low <= high)
         {
/*3*/      int mid = (low + high) / 2;
/*4*/      if (a[mid] < x)
/*5*/         low = mid + 1;
/*6*/      else if (x < a[mid])
/*7*/         high = mid - 1;
           else
/*8*/         return mid;  // found
         }
/*9*/    return NOT_FOUND
}
```

Binary Search with Recursion

```
// Searches an ordered array of integers using recursion
int bsearchr(const int data[], // input: array
             int low,          // input: lower bound
             int high,         // input: upper bound
             int value         // input: value to find
         ) // return index if found, otherwise return -1

{  int middle = (low + high) / 2;
   if (data[middle] == value)
      return middle;
   else if (low >= high)
      return -1;
   else if (value < data[middle])
      return bsearchr(data, low, middle-1, value);
   else
      return bsearchr(data, middle+1, high, value);
}
```

Exponentiation x^n

```
long exp(long x, int n)
{
/*1*/    if (n==0)
/*2*/      return 1;
/*3*/    if (n==1)
/*4*/      return x;
/*5*/    if (isEven(n))
/*6*/      return exp(x*x, n/2);
         else
/*7*/      return exp(x*x, n/2)*x;
}
```

Euclid's Algorithm

- Homework: trace the following algorithm. What is its running time? (*Hint*: see next slide)
- Computing the greatest common divisor (GCD) of two integers

```
long gcd (long m, long n) // assuming m>=n
{
/*1*/    while (n!=0)
           {
/*2*/       long rem = m%n;
/*3*/       m = n;
/*4*/       n = rem;
           }
/*5*/    return m;
}
```

Euclid's Algorithm (2)

- Theorem:
  - If M > N, then M mod N < M/2.
- Max number of iterations:
  - 2logN = O(logN)
- Average number of iterations:
  - $(12 \ln 2 \ln N)/\pi^2 + 1.47$

How to get better at writing recursive methods?

- Close the textbook and lecture notes.
- Recall the algorithms in the lecture notes and implement them in Java.
- Implement homework problems in Java.

Next time …

- Merge sort (section 12.1)
- Quick sort (section 12.2)

- Reading for this lecture: chapter 5

Appendix: Saving Register Values during Function Calls

Saving register values

Saving all the registers is time consuming. Other solutions exist, such as saving only those registers currently in use. Either the caller of the function or the function being called is given the responsibility of saving and restoring any registers that it needs. Another possibility is to use *register windows*, in which sections of memory represent different collections of registers, which can be switched by changing the address holding their starting location. Hybrid solutions exist in which certain registers are allocated to the caller and certain ones to the callee (the function that is called). Any of these solve the basic problem of saving values in a limited number of registers when functions are called.

We will have the caller assume responsibility of saving the registers it uses before it calls a function and then restoring those values when the function finishes. This way the function being called is free to use any registers without conflicts arising. An alternate strategy would be to have the callee push the register values for the registers it will use. This requires an initial pass through the callee's code to determine which registers it will use. In general, it is better to give the callee more responsibility during a function call, because the callee's code is only generated once, whereas the caller's code is generated for each function call.