Analysis of Recursive Algorithms

CSC 411

Richard Kelley

Plan for Analysis of Recursive Algorithms

- General Plan for Analysis
  - Decide on a parameter indicating an input's size.
  - Identify the algorithm's basic operation.
  - Check whether the number of times the basic op. is executed may vary on different inputs of the same size.
    (If it may, the worst, average, and best cases must be ivestigated separately.)
  - Set up a recurrence relation with an appropriate initial condition expressing the number of times the basic op. is executed.
  - Solve the recurrence (or, at the very least, establish its solution's order of growth) by backward substitutions or another method.

Review: Recursive Thinking (from CSC280)

- *Recursion* is a programming technique in which a method can call itself to solve a problem
- A *recursive definition* is one which uses the word or concept being defined in the definition itself
- In some situations, a recursive definition can be an appropriate way to express a concept
- Before applying recursion to programming, it is best to practice thinking recursively

Non-Recursive Programming

- Consider the problem of computing the sum of all the numbers between 1 and N, inclusive
- If N is 5, the sum is 1 + 2 + 3 + 4 + 5
- A non-recursive version (iterative):

```
public int sum (int num)
{
   int result = 0;
   for (i = 1; i <= num ; i++)
      result = result + i;

   return result;
}
```

Recursive Programming

- This problem can be expressed recursively as:

  The sum of 1 to N is N plus the sum of 1 to N-1

  The sum of 1 to N-1 is N-1 plus the sum of 1 to N-2

```
public int sum (int num)
{
   int result;
   if (num == 1)
      result = 1;
   else
      result = num + sum(num-1);
   return result;
}
```

the base case

the recursive case

Recursive Programming

- A method or function can invoke itself; if set up that way, it is called a *recursive method (function)*
- The code of a recursive method function must be structured to handle both the base case(s) and the recursive case(s)
- Each call sets up a new execution environment, with new parameters and new local variables
- As always, when the method completes, control returns to the method that invoked it (which may be another instance of itself)

Recursive Method

- A method that it calls itself.
- In other words: A method that contains a method call with the same name and signature of that method

- Let’s explain this again with another example!
  - Factorial
    - n! = 1 * 2 * 3 * . . . * n

A non-recursive version (iterative):

- Let us rewrite it as follows:
  - n! = n * n-1 * n-2 * n-3 * … *1

```
int fact ( int n )
{
        int i, f=1; // i: counter, f: will hold the result
        for ( i = n; i > 1 ; i--) // loop: n down to 2
                f * = i; // 1 * n * n-1 * n-2 * … * 2

        return f; // return result to calling program
}
```

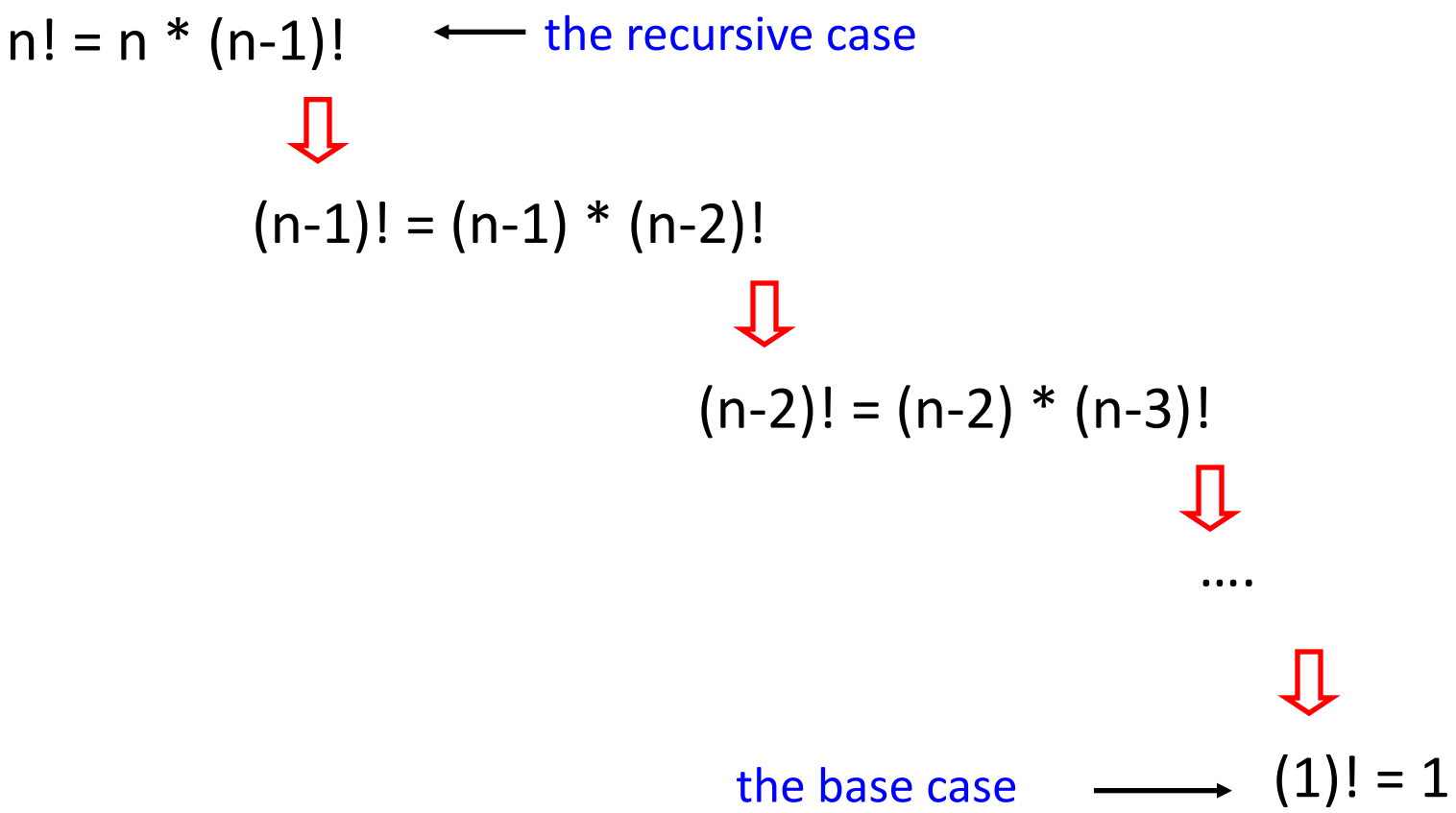
Recursive Thinking

- Mathematical formulas are often expressed recursively
- N!, for any positive integer N, is defined to be the product of all integers between 1 and N inclusive
- This definition can be expressed recursively:

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

- A factorial is defined in terms of another factorial until the base case of 1! is reached

Recursive Thinking



The recursive version

```
int fact ( int n )
{
      if (n == 1 ) // at every call n will decrease by 1
            return 1; // until it reach 1

      else // multiply current n by factorial of (n-1)
            return n * fact (n-1);
}
// important note : n decrease by 1 at each call
//                  until it reach the base !
```

Q: What is the time complexity of the above recursive algorithm?
(will be discussed later)

Another Example: Adding up

- Let sum up squares from n to m (m>=n):
  - SumS(n,m) = $n^2 + (n+1)^2 + (n+2)^2 + \ldots + m^2$
- **A none recursive version (iterative):**

```
public int SumS ( int n, int m )
{
        int i, sum=1; // i: counter, sum: to hold result

        for ( i = n; i <=m ; i++) // loop: n to m
                sum + = i * i; // (n*n) + (n+1)*(n+1) + … + m*m

        return sum; // returns the result
}
```

The recursive version

```
public int SumS (int n, int m )
{
        if (n == m ) // Stop when n reaches m
                return m * m; // and return last squared sum

        else // multiply n by n and add the result of the
                return (n*n) + SumS (n+1,m); // next sum (n+1)
}
// important note : n increase by 1 at each call
//                          until it reaches m
```

Example: Recursive evaluation of $n!$

Definition: $n! = 1 * 2 * \ldots * (n-1) * n$ for $n \geq 1$ and $0! = 1$

Recursive definition of $n!$: $F(n) = F(n-1) * n$ for $n \geq 1$ and
$F(0) = 1$

ALGORITHM $F(n)$

//Computes $n!$ recursively
//Input: A nonnegative integer n
//Output: The value of $n!$
if $n = 0$ **return** 1
else return $F(n-1) * n$

- Size: n
- Basic operation: multiplication
- Recurrence relation: $T(n) = T(n-1) + 1$

Solving the recurrence for T(*n*)

$T(n) = T(n-1) + 1,\ T(0) = 0$

$T(n) = T(n-1) + 1$

$= (T(n-2) + 1) + 1 = T(n-2) + 2$

$= (T(n-3) + 1) + 2 = T(n-3) + 3$

…

$= T(n-i) + i$

…

$= T(n-n) + n$

$= n$

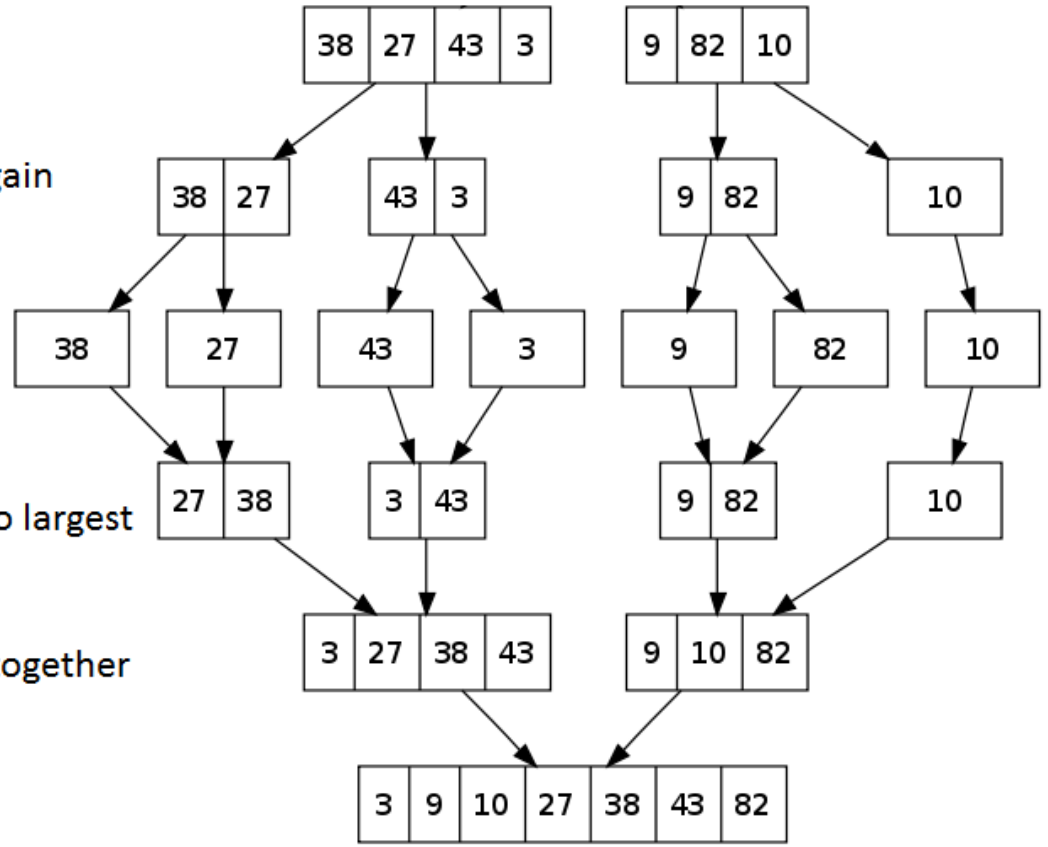
The method is called backward substitution.

Analyzing (non)Recursive Algorithms

- When analyzing *a loop*, we determine the order of the loop body and multiply it by the number of times the loop is executed
- Recursive analysis is similar
- We determine the order of the method body and multiply it by *the order of the recursion* (the number of times the recursive definition is followed)

Example: MergeSort

1. Divide the array into two parts
2. Divide the array into two parts again
3. Break each element into single parts
4. Sort the elements from smallest to largest
5. Merge the divided sorted arrays together
6. The array has been sorted



Example: Mergesort

```
MergeSort(L)
1    if (length of L > 1) {
2        Split list into first half and second half
3        MergeSort(first half)
4        MergeSort(second half)
5        Merge first half and second half into sorted list
6    }
```

Example: MergeSort

```
MergeSort(L)
1    if (length of L > 1) {
2        Split list into first half and second half
3        MergeSort(first half)
4        MergeSort(second half)
5        Merge first half and second half into sorted list
6    }
```

What is the recurrence relation?

$T(n) = 2T(n/2) + n$

Assume $n = 2^k$, and using backward substitution,

$\Rightarrow T(n) = O(n \log n)$

Master Theorem

Let T be an increasing function that satisfies the recurrence relation:

$$T(n) = a\,T(n/b) + cn^d$$

whenever $n = b^k$, where k is a positive integer, a >= 1, b is an integer greater than 1, c is a positive real number, and d is a non-negative real number. Then:

$$T(n) = \begin{cases} O(n^d) & \text{if } a < b^d \quad \text{case 1} \\ O(n^d \log_b n) & \text{if } a = b^d \quad \text{case 2} \\ O(n^{\log_b a}) & \text{if } a > b^d \quad \text{case 3} \end{cases}$$

Master Theorem

Let T be an increasing function that satisfies the recurrence relation:

$$T(n) = a\,T(n/b) + cn^d$$

whenever $n = b^k$, where k is a positive integer, a >= 1, b is an integer greater than 1, c is a positive real number, and d is a non-negative real number. Then:

$$T(n) = \begin{cases} O(n^d) & \text{if } a < b^d \quad \text{case 1} \\ O(n^d \log_b n) & \text{if } a = b^d \quad \text{case 2} \\ O(n^{\log_b a}) & \text{if } a > b^d \quad \text{case 3} \end{cases}$$

a >= 1: The number of subproblems

b > 1: Amount by which problems shrink

cn^d: Amount of nonrecursive work at each level of recursion.

The cases are (effectively) comparing cn^d with n^(log_b(a))

Examples

$T(n) = 2T(n/2) + n$

$a = 2, b = 2, c = 1, d = 1$

$\Rightarrow$ Case 2

$\Rightarrow T(n) = O(n\log_2 n) = O(n\log n)$

$$T(n) = a\,T(n/b) + cn^d$$

$$T(n) = \begin{cases} O(n^d) & \text{if } a < b^d \\ O(n^d \log_b n) & \text{if } a = b^d \\ O(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Examples

$T(n) = 2T(n/2) + n^2$

$a = 2, b = 2, c = 1, d = 2$

$\Rightarrow$ Case 1

$\Rightarrow T(n) = O(n^2)$

$$T(n) = a\,T(n/b) + cn^d$$

$$T(n) = \begin{cases} O(n^d) & \text{if } a < b^d \\ O(n^d \log_b n) & \text{if } a = b^d \\ O(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Examples

$T(n) = 2T(n/2) + \sqrt{n}$

$a = 2, b = 2, c = 1, d = 1/2$

$\Rightarrow$ Case 3

$\Rightarrow T(n) = O(n)$

$$T(n) = a\,T(n/b) + cn^d$$

$$T(n) = \begin{cases} O(n^d) & \text{if } a < b^d \\ O(n^d \log_b n) & \text{if } a = b^d \\ O(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Example: MergeSort

```
MergeSort(L)
1    if (length of L > 1) {
2        Split list into first half and second half
3        MergeSort(first half)
4        MergeSort(second half)
5        Merge first half and second half into sorted list
6    }
```

$T(n) = 2T(n/2) + n$

$a = 2, b = 2, c = 1, d = 1$

$\Rightarrow$ Case 2

$\Rightarrow T(n) = O(n \log n)$

Inadmissible equations

- The following equations cannot be solved using the master theorem:

$T(n) = ½\ T(n/2) + n$

- a is ½
- a needs to be a constant and a >=1

$T(n) = 2T(n/2) + n/\log n$

- the cost of the work done outside the recursive calls is non-polynomial

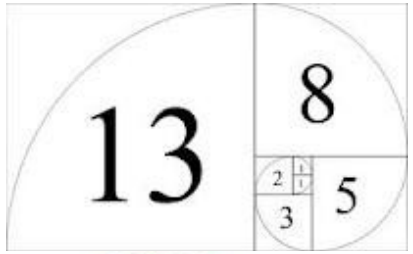
$T(n) = 2^n T(n/2) + n^n$

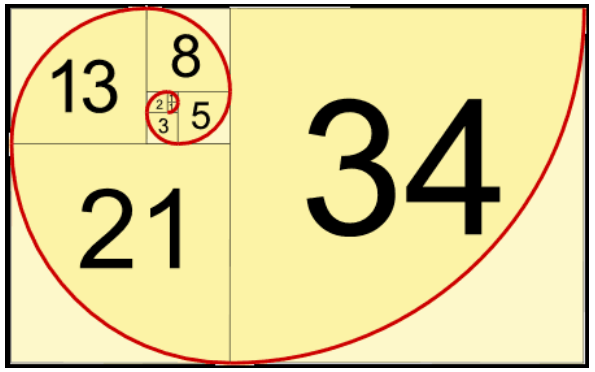
- a is not a constant

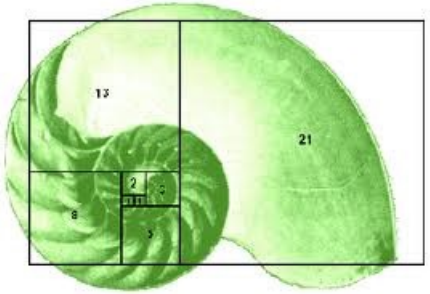
Fibonacci numbers

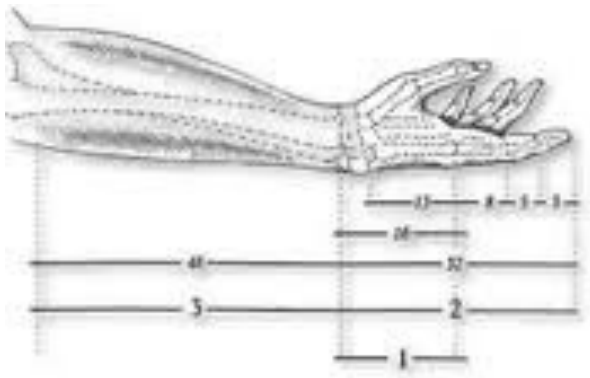
The Fibonacci numbers:

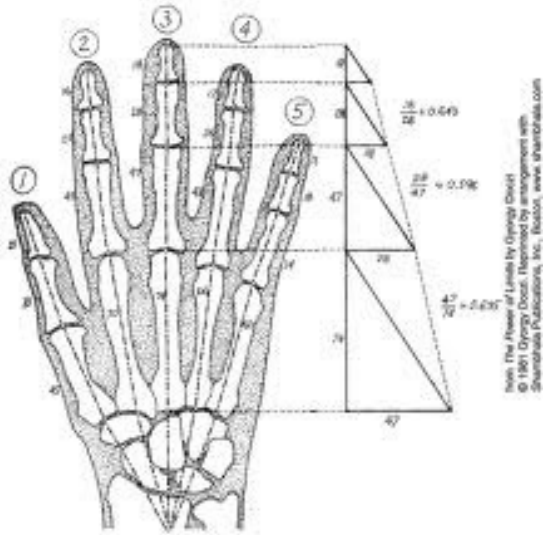
0, 1, 1, 2, 3, 5, 8, 13, 21, …



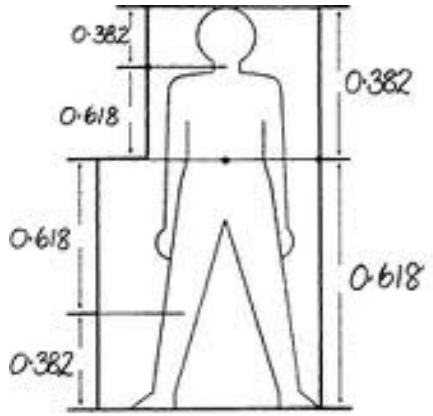








Fibonacci subdivisions in the hand



Definition-based recursive algorithm

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

a strong candidate for the title of
Worst Algorithm in the World

- T(n<=1) = O(1)
- T(n) = T(n-1)+T(n-2)+O(1)

  General 2nd order linear homogeneous recurrence with constant coefficients.

Analysis Of Recursive Fibonacci

$T(n) = c$ if $n = 1$ or $n = 2$ (1)

$T(n) = T(n-1) + T(n-2) + b$ if $n > 2$ (2)

We determine a lower bound on T(n):

Expanding: $T(n) = T(n-1) + T(n-2) + b$

$\geq T(n-2) + T(n-2) + b$

$= 2T(n-2) + b$

$= 2[T(n-3) + T(n-4) + b] + b$ by substituting T(n - 2) in (2)

$\geq 2[T(n-4) + T(n-4) + b] + b$

$= 2^2T(n-4) + 2b + b$

$= 2^2[T(n-5) + T(n-6) + b] + 2b + b$ by substituting T(n - 4) in (2)

$\geq 2^3T(n-6) + (2^2 + 2^1 + 2^0)b$

. . .

$\geq 2^kT(n-2k) + (2^{k-1} + 2^{k-2} + \ldots + 2^1 + 2^0)b$

$= 2^kT(n-2k) + (2^k - 1)b$

The base case is reached when $n - 2k = 2$ →

$k = (n-2)/2$

Hence $T(n) \geq 2^{(n-2)/2}T(2) + [2^{(n-2)/2} - 1]b$

$= (b + c)2^{(n-2)/2} - b$

$= [(b + c)/2]*(2)^{n/2} - b$ →

Recursive Fibonacci is exponential

Fibonacci numbers

The Fibonacci numbers:

0, 1, 1, 2, 3, 5, 8, 13, 21, …

The recurrence relation: $T(n) = T(n-1) + T(n-2) + 1$

What is the order of the algorithm?

General 2nd order linear homogeneous recurrence with constant coefficients:

$$aX(n) + bX(n-1) + cX(n-2) = 0$$

Goal: Solving the above quadratic equation.

Solving $aX(n) + bX(n\text{-}1) + cX(n\text{-}2) = 0$

- Set up the characteristic equation (quadratic)

$$ar^2 + br + c = 0$$

- Solve to obtain roots r_1 and r_2

- General solution to the recurrence

  if r_1 and r_2 are two distinct real roots: $X(n) = \alpha r_1^n + \beta r_2^n$

  if $r_1 = r_2 = r$ are two equal real roots: $X(n) = \alpha r^n + \beta n r^n$

- Particular solution can be found by using initial conditions