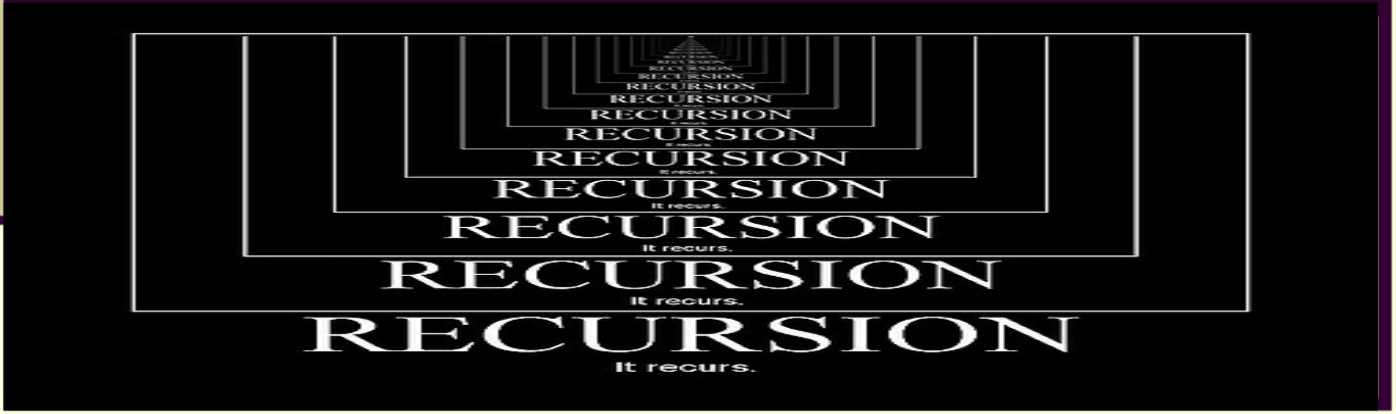
Recursions and Their Complexity Analysis



Objectives

- Recursion: definitions
- Explore the base case and the general case of a recursive algorithms
- Explore how to use recursive functions to implement recursive algorithms
- Complexity analysis of recursive algorithms

Recursive Definitions

- **Recursive algorithm**

  Algorithm that finds the solution to a given problem by reducing the problem to smaller version(s) of itself

  While reducing the problem, a base case is reached.

  Has one or more base cases

  Implemented using recursive functions

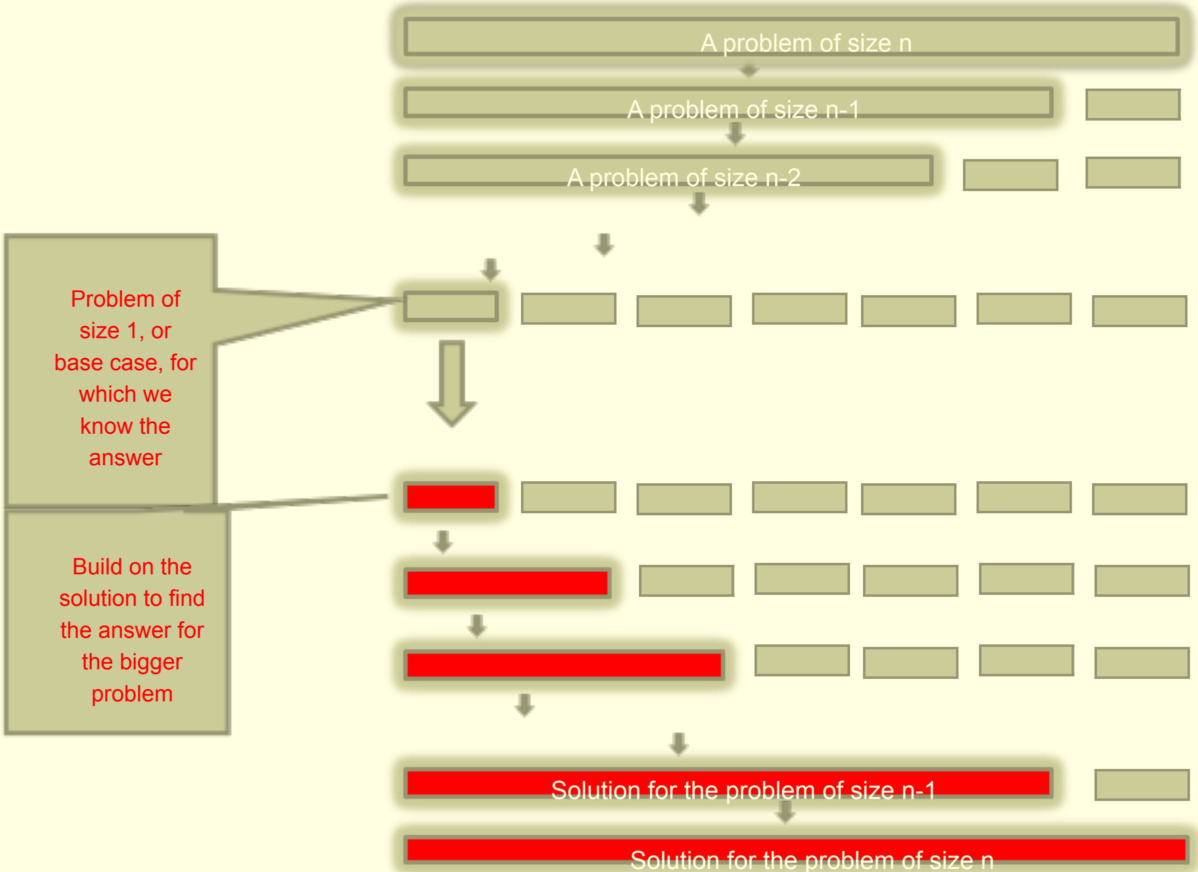
Recursive Definitions

- **Anchor, ground, or base case:**
  Case in recursive definition in which a known in advance
  Stops the recursion
- **General case:**
  Case in recursive definition in which a smaller version of definition itself is called
  Must eventually be reduced to a base case

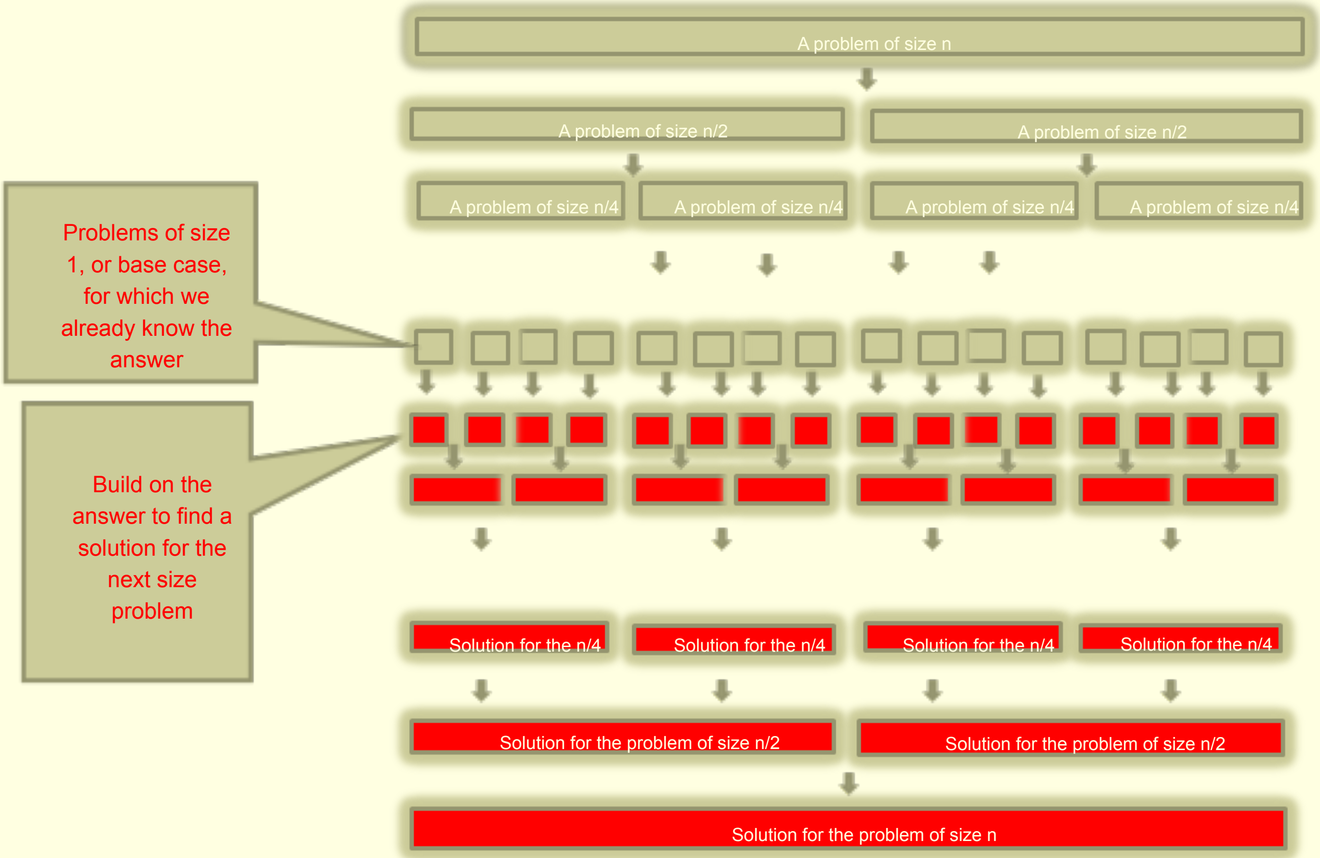
Recursion

- Process of solving a problem by reducing it to smaller versions of itself
- It can be either
  - Reduce and conquer
    - Reduce a problem of size *n* into a problem of size n- *c*, where *c* is a positive constant value.
  - Divide and conquer
    - Reduce a problem of size *n* into one ore more problem of size n/*d*, where *d*=2, 3, …

Recursive Problem Solving: Reduce and Conquer (case of c=1)



Recursive Problem Solving: Divide and Conquer (Divide by 2)



Recursion: Example

- Classic linear recursion example--the factorial function:
- factorial(n) or n! = $1 \cdot 2 \cdot 3 \cdot \cdots \cdot (n-1) \cdot n$
- Recursive definition:

$$factorial(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot factorial(n-1) & otherwise \end{cases}$$

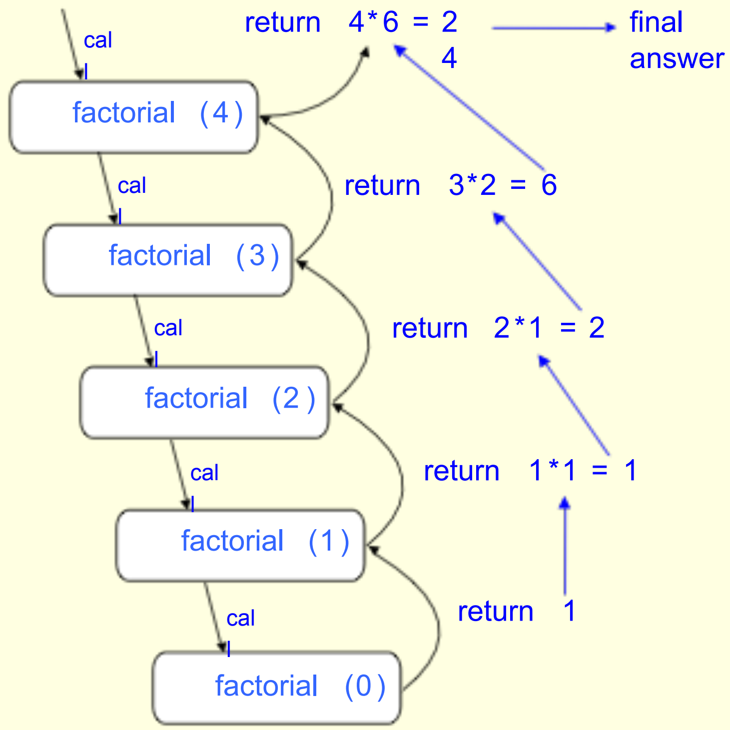
- In this example, *factorial(0)* is called the base case (there should be at least one) and *factorial(n)* for *n > 0* is called recursive case.

Recursive Factorial Function

```
int factorial(int n)
{
  if(n == 0)
     return 1;
  else
     return n * factorial(n- 1);
}
```

Visualizing Factorial

Example recursion trace for factorial(4) :



Designing Recursive Functions: What you Should Do

- Understand problem requirements
- Identify base case(s)
- Provide direct solution to each base case
- Identify general case(s)
- Provide solutions to general cases in terms of smaller versions of the problem itself

Recursive Factorial Function: Ex.

$$factorial(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot factorial(n-1) & otherwise \end{cases}$$

- Base case when n=0, the function returns 1
- General case when n>0, the function returns n*factorial(n-1)

Recursive Factorial Function: Ex.

```
int factorial(int n)
{
  if(n == 0)
     return 1;
  else
     return n * factorial(n- 1);
}
```

Base case

General case

Adding the Numbers in an Array: Ex. Divide and Conquer Approach

- For any array with a size different than one, Divide the array into two (or more) sub-arrays
- Find the sums of the two sub-arrays, and add the values to find the sum of all the elements in the original array

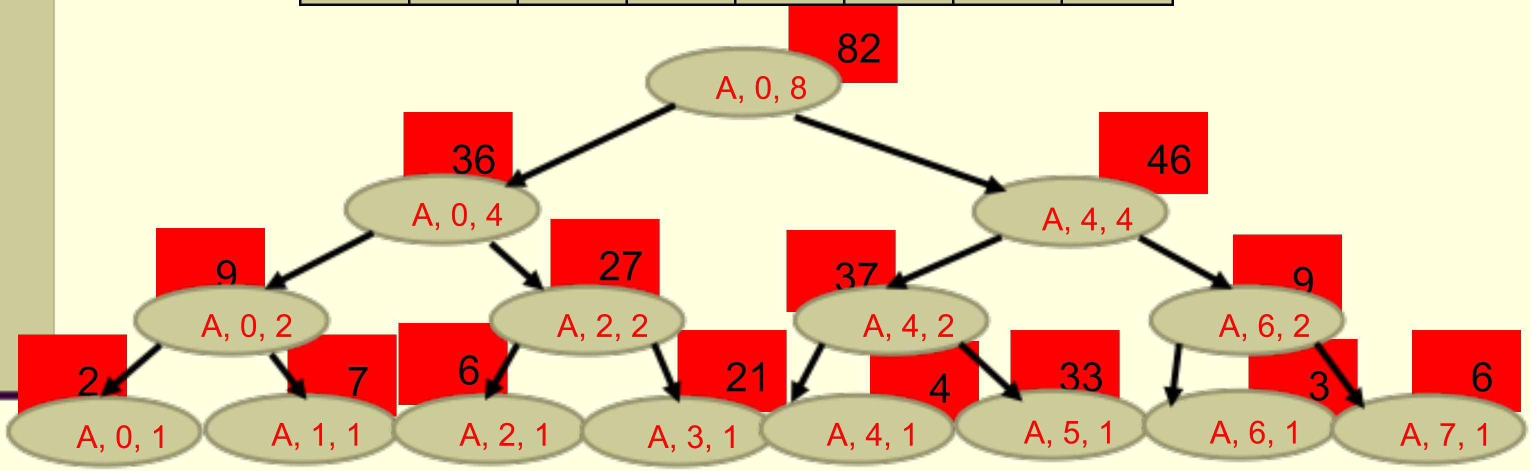
Understand problem requirements	→	√
Identify base cases	→	Array of size 1
Provide direct solution to each base case	→	BinaryRecSum = value of the one element in the array
Identify general case(s)	→	Array of size > 1
Provide solutions to general cases in terms of smaller versions of itself	→	BinaryRecSum (Whole array) = BinaryRecSum(Left half of the array) + BinaryRecSum(Right half of the array)

Binary Recursive Trace

Example: BinaryRecSum(*A, 0, 8*):

A =

2	7	6	21	4	33	3	6



Adding all the Numbers in an Array A: Divide and Conquer Approach

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

Algorithm BinaryRecSum(*A, i, n*):
Input: An array *A*, an int i *(starting index for the array)* and *n (# of elements)*

4	6	23	14	12

Output: The sum of the *n* integers in *A* starting at index *i*
BinaryRecSum(*A, 0, 5*) = 59

if *n* = 1 **then**
return *A*[*i*]
else
return
BinaryRecSum(*A, i, ceil(n/* 2)) + BinaryRecSum(*A, i* + ceil(*n/* 2), *floor(n/* 2))

Adding the Numbers in an Array: Reduce and Conquer Approach

- The sum of *n* elements in the array is equal to the value of the first element plus the sum of the all the remaining *n-1* elements in the array

- Understand problem requirements → √
- Identify base cases → Array of size 1
- Provide direct solution to each base case → LinearRecSum = value of the one element in the array
- Identify general case(s) → Array of size > 1
- Provide solutions to general cases in terms of smaller versions of itself → LinearRecSum(n elements) = (Value of the first element) + LinearRecSum(n-1 remaining elements)

Adding all the Numbers in an Integer Array A: Reduce and Conquer Approach

- Unary recursion occurs whenever there is one recursive call for each non-base case.

Algorithm LinearRecSum(*A, i, n*):
Input: An array *A,* an int i *(starting index for the array)* and *n (# of elements)*

4	6	23	14	12

Output: The sum of the *n* integers in *A* starting at index *i*
BinaryRecSum(*A, 0, 5*) = 59

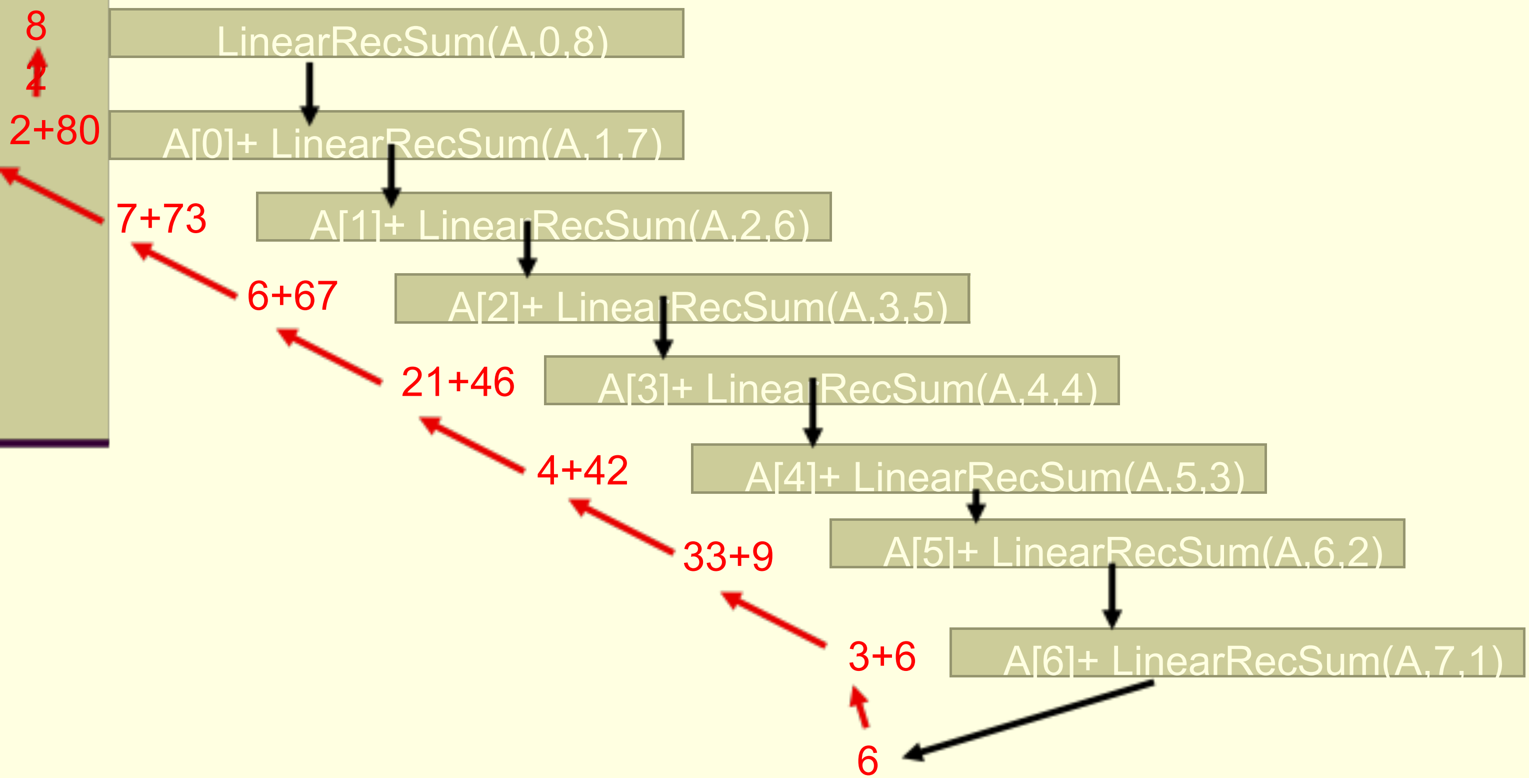
if *n* = 1 **then**
  return *A*[*i*]
else
  return A[i]+ LinearRecSum(*A, i+1, n-1*)

Linear Recursive Trace

Example: LinearRecSum(A, 0, 8):

A =

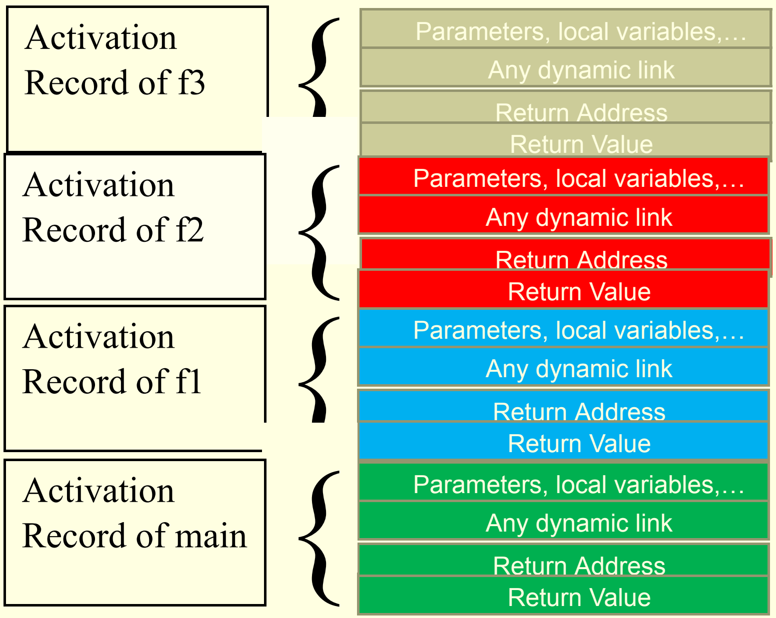
2	7	6	21	4	33	3	6



SPACE COMPLEXITY ANALYSIS OF RECURSIVE FUNCTIONS

How does the OS Perform Recursions?

- How does the run-time stack looks like, if a function *main()* calls f1, and f1 calls f2, and f2 calls f3.

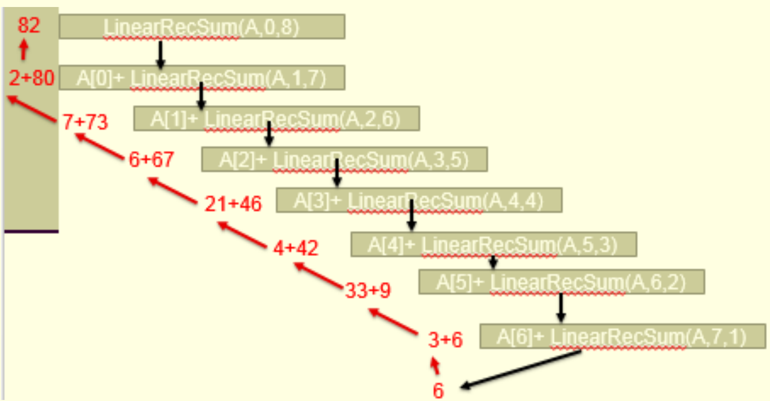


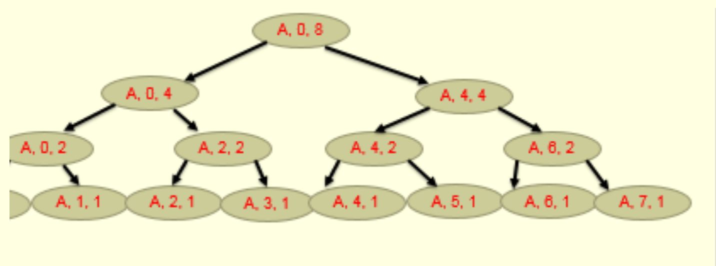
Space Complexity

What is the size OS stack required to execute LinearRecSum and BinaryRecSum, given that the size of the array is *n*?? (Hint: use the size of the activation block)

A =

2	7	6	21	4	33	3	6





n* (size of function activation block)
O(n)

lg n* (size of function activation block)
O(lg n)

TIME COMPLEXITY ANALYSIS OF REDUCE AND CONQUER RECURSIVE FUNCTIONS

Complexity Analysis of Recursive Algorithms

Steps in mathematical analysis of worst case recursive algorithms with an input of size *n*:

- Decide on parameter *n* indicating ***input size***
- Identify algorithm's *basic operation* (***Not the base case***)
- Compute the number of Basic operations C(.) for the base case and for the general case. (typically it is a recurrence function for the general case)
- Solve the recurrence to obtain a closed form or estimate the order of magnitude of the solution(using the Master Method). (To get a closed form of a recurrence relation, you can use backward or forward substitutions or another method).

Recursive Factorial Function

```
int factorial(int n)
{
  if(n == 0)
   return 1;
  else
   return n * factorial(n- 1);
}
```

Base case

General case

What would make sense as a BO?
What would be the size of the input problem?

Complexity Analysis of Recursive Algorithms: Factorial

Steps in mathematical analysis of recursive algorithms:

- Decide on parameter *n* indicating <u>*input size*</u>
- Identify algorithm's <u>*basic operation*</u>
- Compute the number of Basic operations C(n) for the base case and for the general case. (typically it is a recurrence function for the general case)
- Solve the recurrence to obtain a closed form or estimate the order of magnitude or complexity of the solution.

- → *n*
- → multiplication
- → $C(0) = 0;$

$C(n) = C(n-1) + 1;$

- → $C(n) = 0+1+1+\ldots+1$

$= n$

$C(n)$ is $O(n)$

Recurrence Relationship for the Number of Basic Operations vs. the Original Recursive Function

- The recurrence function for the number of basic operations is different from the original recursive function
- Factorial
  - *factorial*(*n*) := 1 if n = 0
  - *factorial*(*n*) := *factorial*(*n-1*) * *n* *if n >=1*
- Recurrence for number of multiplications to compute *n*!:
  - C(0) = 0; if n = 1
  - C(n) = C(n-1) +1; *if n >=1*

Solving Recurrence Relation Using Forward Substitution: Factorial

C(n) = C(n-1) + 1 C(0) = 0

C(n-1) = C(n-2) +1

C(n-2) = C(n-3) +1

:

C(n) = C(n-1) + 1

= C(n-2) + 1 +1

= C(n-3) + 1 + 1+ 1

= C(n-4) + 1 +1 +1+ 1

:

= C(0) +1 +1 +1 +1…..+1 (there are *(n)* 1’s)

= 0 + n

= n

O(n)

Solving Recurrence Relation Using Forward Substitution.

- Ex 1
- **C(n) = C(n-1) +4   C(1) = 2**
  - C(n-1) = C(n-2) + 4
  - C(n-2) = C(n-3) + 4
  - :
  - :
  - C(1) = 2
  - C(n)   = C(n-1) + 4
  - = C(n-2) + 4 + 4
  - :
  - C(n) = C(1) + 4+4+4…+4 + 4
  - C(n) = 2 + 4+4+4…+4 + 4   //with n-1 "4's"
  - C(n) = (n-1) * 4 + 2 = 4n -2

Solving Recurrence Relation Using Backward Substitution: Factorial

$C(n) = C(n-1) + 1 \qquad C(0) = 0$

$C(1) = C(0) + 1 = 0 + 1 = 1$

$C(2) = C(1) + 1 = 1 + 1 = 2$

$C(3) = C(2) + 1 = 2 + 1 = 3$

$C(4) = C(3) + 1 \quad = 3 + 1 = 4$

:

:

$C(n) = n$

$O(n)$

TIME COMPLEXITY ANALYSIS OF DIVIDE AND CONQUER RECURSIVE FUNCTIONS

Approximating the Order of Growth of the Recurrence Relation: Master Method Apply to Divide-and-Conquer Cases

If a problem of size *n* is solved recursively by diving the problem into *a* sub-problems, each of size *n/b*, and if the amount of work required to divide the problem and to combine the solutions is *f(n)* then we can say that:

C(*n*) = *aC*(*n/b*) + *f* (*n*) where *f* (*n*) is O(*nk*) *then*

If *a* < *bk* then *C*(*n*) is O(*nk*)

If *a* = *bk* then *C*(*n*) is O(*nk* lg *n*)

If *a* > *bk* then *C*(*n*) is O(*n*log *b* *a*)

- Examples:

C(n) = 2C(n/2) + n

a=2, b=2, k=1 (since f(n) = n is O(*n1*)),

a = bk => C(n) is O(nlg n)

C(n) = C(n/2) + 1 //(f(n) = 1 is O(1)□ k = 0),

a=1, b=2, k=0, a = bk => C(n) is O(lg n)

C(n) = 3 C(n/2) + n //(f(n) = n is O(n)□ k = 1),

a=3, b=2, k=1, a > bk => C(n) is O(nlog23)

C(n) = 8C(n/2) + n2+ n – 100 //(f(n) = n2+ n – 100 is O(n2)□ k = 2),

a=8, b=2, k=2, (a= 8) ? (bk = 22) => C(n) is O(nlog28)

Adding all the Numbers in an Integer Array A: Binary Recursive Method

Algorithm BinaryRecSum(*A, i, n*):
  Input: An array *A* and integers *i* and *n*

4	6	23	14	12

  Output: The sum of the *n* integers in *A* starting at index *i*
    BinaryRecSum(*A, 0, 5*) = 59

if *n* = 1 **then**
  return *A*[*i*]
return BinaryRecSum(*A, i, Ceil(n/* 2)) +
    BinaryRecSum(*A, i* + ceil(*n/* 2)*, floor(n/* 2))

C(n) = 2 C(n/2) + 1
a =2, b =2, ; f(n) = 1 ==> k = 0 ;
a > bk ==> C(n) since *a > bk* then *C*(*n*) is O($n^{\log_2 2}$) = *O(n)*

Computing Fibonacci Numbers

- Fibonacci numbers are defined recursively:

  $F_0 = 0$
  $F_1 = 1$
  $F_i = F_{i-1} + F_{i-2}$ for $i > 1$.

- The Fibonacci sequence:
  0, 1, 1, 2, 3, 5, 8, 13, 21, …
- As a recursive algorithm (first attempt):

Algorithm BinaryFib(k):
    Input: Nonnegative integer k
    Output: The kth Fibonacci number F_k
    if $k <= 1$ **then**
        return k
    else
        return BinaryFib(k - 1) + BinaryFib(k - 2)

Fibonacci Numbers

- Fibonacci recurrence: (use addition as basic operation)

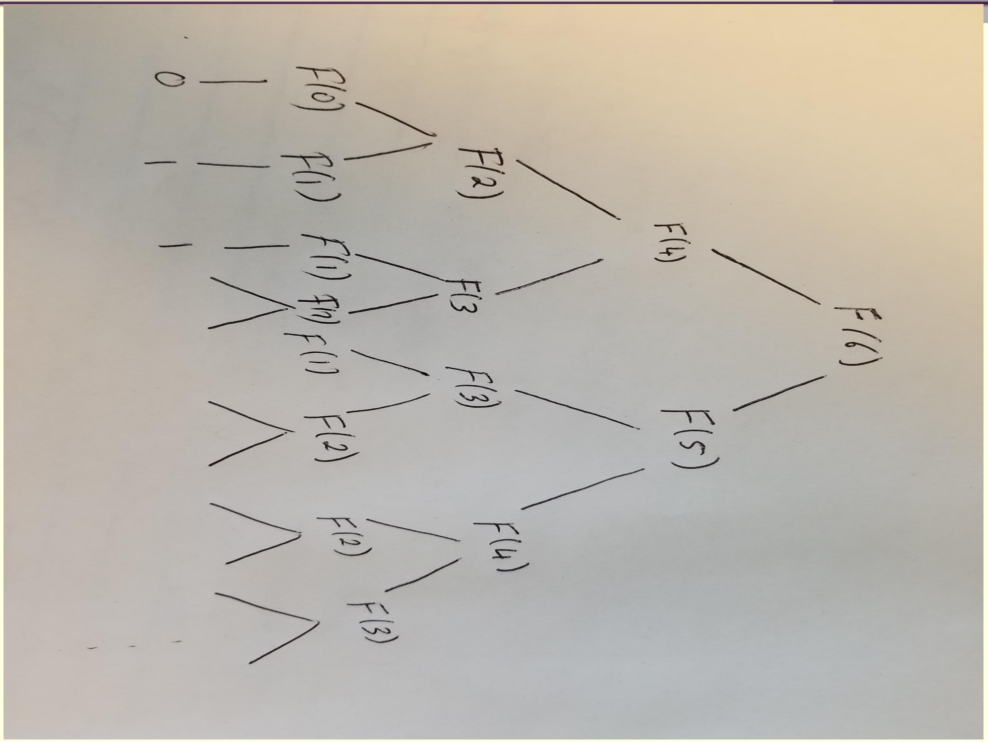
  $C(n) = C(n-1) + C(n-2) + 1$

  or $C(n) - C(n-1) - C(n-2) - 1 = 0$

  $C(0) = C(1) = 0$

2nd order linear homogeneous recurrence relation with constant coefficients

Fibonacci Numbers: Tree call for F(6)



- $C(n) \approx 2n$. It is <u>exponential</u>!

Fibonacci Numbers: Iterative

```
unsigned int interativeFib (unsigned int n) {
if (n<2)
   return n;
else {
   register int tmp, second = 1, first = 0;
   for (i = 2; i<=n; i++){
      tmp = first + second;
      first = second;
      second = tmp;
   }
   return current;
}
}
```

Assuming we are using addition as basic operation

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

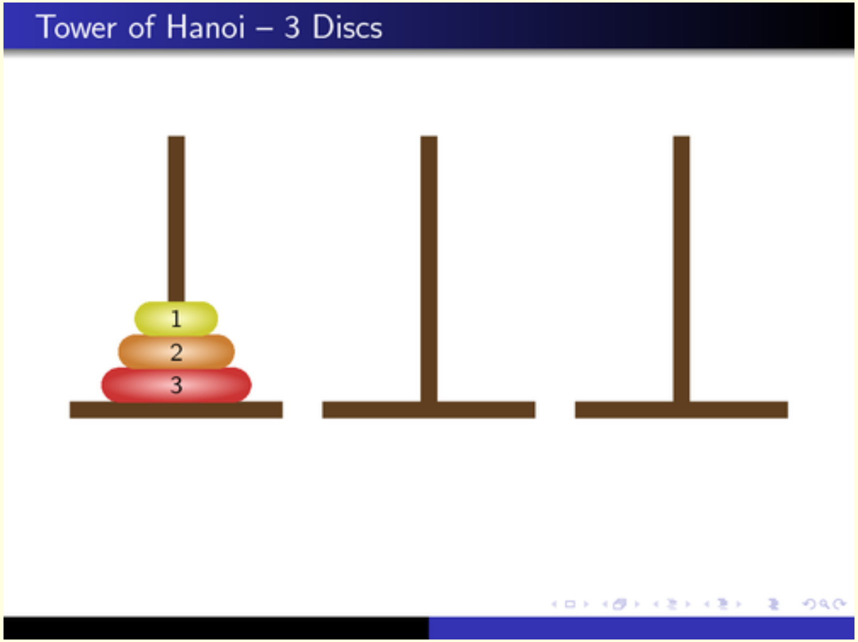
Complexity: O(?) O(n)

Fibonacci Numbers

n	Iterative Fibonacci Complexity 2n-2	Recursive Fibonacci Complexity 2^n
10	18	1024
20	38	1048576
30	58	1073741824
40	78	1.09951E+12
50	98	1.1259E+15
60	118	1.15292E+18
70	138	1.18059E+21
80	158	1.20893E+24
90	178	1.23794E+27
100	198	1.26765E+30

Towers of Hanoi

Remember the Good Old Days



Towers of Hanoi
the legend

- In the temple of Banares, says he, beneath the dome which marks the center of the World, rests a brass plate in which are placed 3 diamond needles, each a cubit high and as thick as the body of a bee. On one of these needles, at the creation, god placed 64 discs of pure gold, the largest disc resting on the brass plate and the others getting smaller and smaller up to the top one. This is the tower of Brahma. Day and night unceasingly the priests transfer the discs from one diamond needle to another according to the fixed and immutable laws of Brahma, which require that the priest on duty must not move more than one disc at a time and that he must place this disc on a needle so that there is no smaller disc below it. When the 64 discs shall have been thus transferred from the needle on which at the creation god placed them to one of the other needles, tower, temple and Brahmans alike will crumble into dust and with a *thunder clap the world will vanish*.

Towers of Hanoi

Relax

If the legend was true, and if the priests were able to move disks at a rate of one disk per second, using the smallest number of moves, it will take them 264−1 seconds or roughly 584.542 billion years to move 64 disks.

Towers of Hanoi Problem

- Invented by Edouard Lucas, in 1883
- Given a tower of *n* disks, initially stacked in increasing size on one of three pegs, the objective is to transfer the entire tower to one of the other pegs, moving only one disk at a time and never a larger one onto a smaller.
- See it at http://www.cut-the-knot.org/recurrence/hanoi.shtml
- Play it at http://www.mazeworks.com/hanoi/

Towers of Hanoi: Algorithm

- Understand problem requirements
- Identify base cases
- Provide direct solution to each base case
- Identify general case(s)
- Provide solutions to general cases in terms of smaller

→ √

→ 1 disk on peg *I that needs to be moved to peg j   1<= i, j, <=3*

→ Move the disk from peg *i* to peg *j*

→ *n* disk on peg *1*

→
- Move *n-1* disk from peg 1 to peg 2
- Move the nth disk from peg 1 to peg 3
- Move *n-1* disk from peg 2 to peg 3

Towers of Hanoi: Recursive Algorithm

```
void moveDisks(int count, int needle1, int needle3, int
   needle2)
{
  if(count > 0)
  {
     moveDisks(count - 1, needle1, needle2, needle3);
     cout<<"Move disk "<<count<<" from "<<needle1
         <<" to "<<needle3<<"."<<endl;
     moveDisks(count - 1, needle2, needle3, needle1);
  }
}
```

Time Efficiency of Recursive Algorithms

Steps in mathematical analysis of recursive algorithms:

- Decide on parameter *n* indicating *input size*
- Identify algorithm's *basic operation*
- Determine *worst* , average and best case for input of size *n*
- Set up a recurrence relation and initial condition(s) for *C*(*n*)-the number of times the basic operation will be executed for an input of size *n* (alternatively count recursive calls).
- Solve the recurrence to obtain a closed form or estimate the order of magnitude of the solution. You can use by backward substitutions or another method.

- → *n*
- → Moving a disk
- → There is no worst, best, and worst case
- → C(1) = 1;
  C(n) = C(n-1) +1 +C(n-1) ;
- Next Slide

Example: Tower of Hanoi

1. Recurrence function

- C(n) = C(n-1) + 1 + C(n-1) = 2 C(n-1) + 1
- C(n) = 2 (2C(n-2) +1) + 1 = 22C(n-2) + 2 +1
- C(n) = 22(2C(n-3)+1) + 2 +1 = 23C(n-3) +4+ 2 +1

:

:

- C(n) = …… = 2iC(n-i) + 2i-1 + 2i-2+…1
- We arrive at the base case when i = n-1
- Remember that : □0□*i*□*n* 2*i* = 2*0* + 2*1* +…+ 2*n* = 2*n*+*1* – 1
- With C(1) = 1,
- C(n) = 2iC(n-i) + 2i-1
- □ C(n) = 2n-1+ 2n-1-1 = 2n-1 □ O(2n)

Recursion or Iteration?

- Tradeoffs between two options
  - Sometimes recursive solution is easier, always consistent with the logic of the original definition of the algorithm
  - Recursive solution is *often* slower, but not if the stack operation is done in hardware
- Recursion should be avoided if some part of the work is unnecessarily repeated to compute an answer, like in the case of Fibonacci

Questions

Questions

Questions

Questions

Questions

Questions