CSCI203 – Algorithms and data structures

Recursion

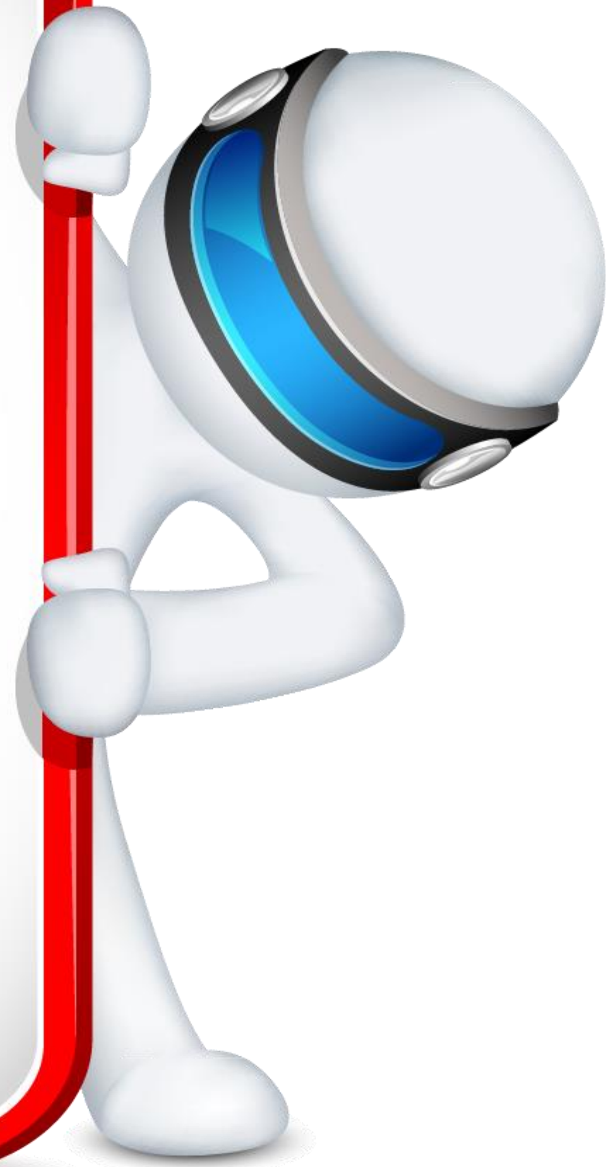
Sionggo Japit

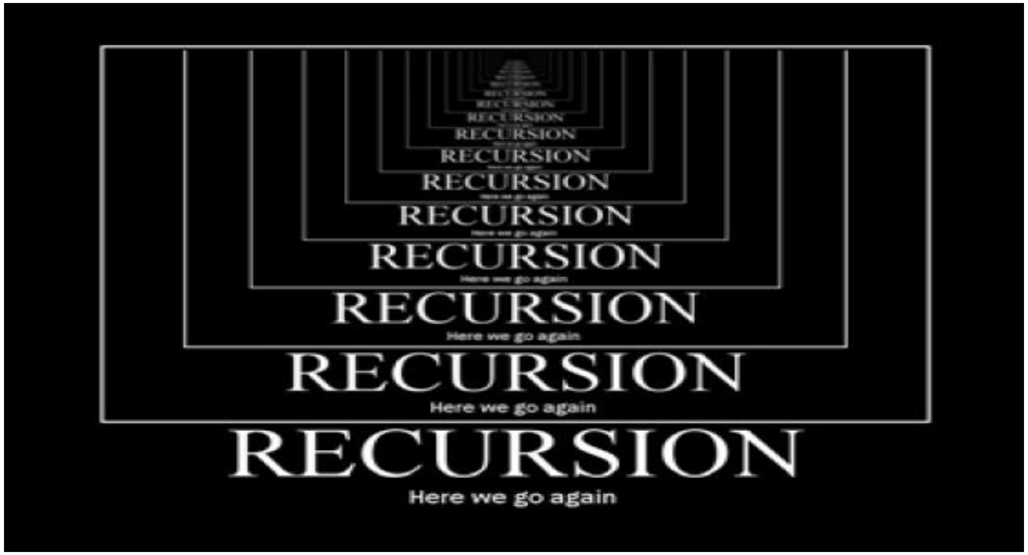
sjapit@uow.edu.au

2 January 2023

Lecture Outline

- Recursion
- Recurrence Relation

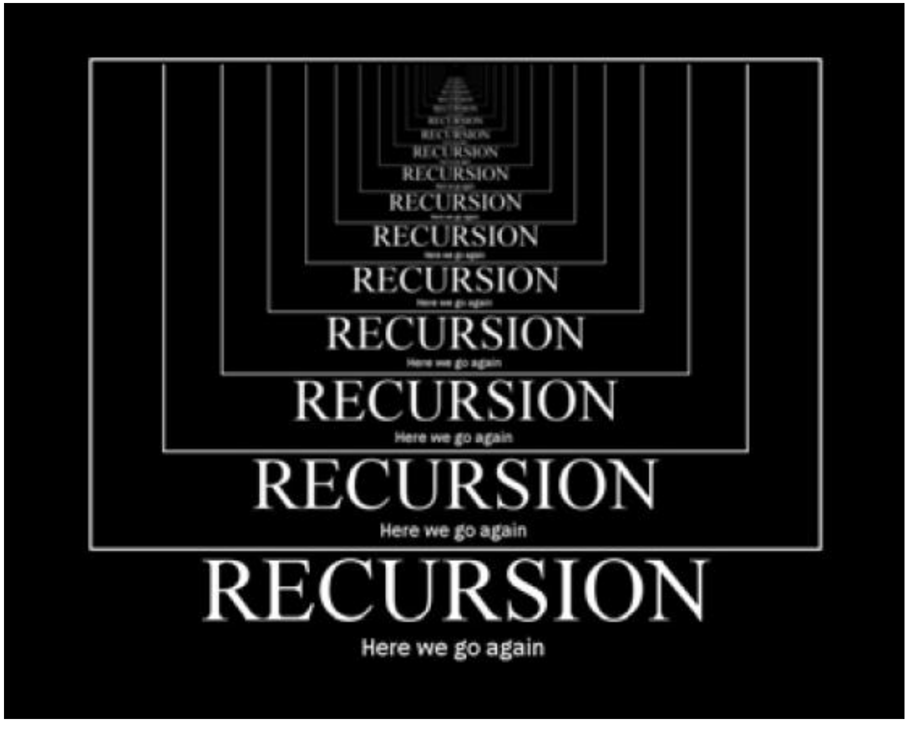




Recursion

Recursion

- Recursion is a repetitive process in which an algorithm calls itself.



Recursion

- Each recursive call solves an identical, but smaller, problem.
- A test for the base case enables the recursive calls to stop.
- Eventually, one of the smaller problems must be the base case.
  - 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.

Recursion

Fibonacci numbers

- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …
- Definition:

$$f(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ f_{n-1} + f_{n-2}, & n \geq 2 \end{cases}$$

The Fibonacci numbers are the sequence of numbers $\{F_n\}_{n=1}^{\infty}$ defined by the linear recurrence equation

$$F_n = F_{n-1} + F_{n-2}$$

With $F_0 = 0 \ and \ F_1 = 1.$

Recursion

- It appears that a simple recursive algorithm will do a good job of calculating f_n.

```
algorithm fib (val num <integer>)
if (num is 0 or num is 1)
    return num
end if
return (fib (num -1) +
            fib (num -2))
end fib
```

```
long fib (long num)
{
    if (num == 0 || num == 1)
        return num;
    return (fib (num -1) +
                fib (num -2));
}
```

Recursion

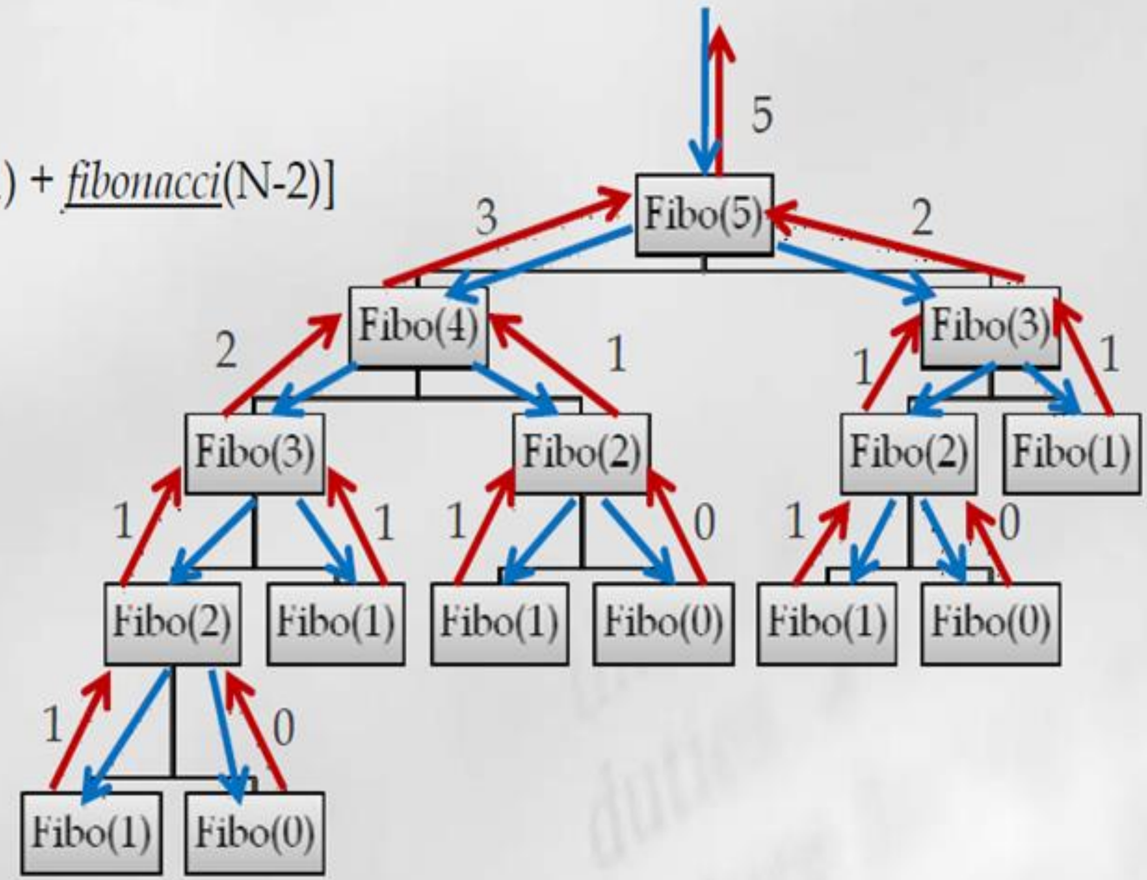
- How good is this solution?
  - Note that we calculate the same function more than once.
  - This can't be a good thing.
  - Just how bad is it?

Recursion

- Consider fibrec(5)
  - To compute fibrec(5), the algorithm needs fibrec(4) and fibrec(3)
  - But fibrec(4) needs fibrec(3) and fibrec(2)
  - And fibrec(3) needs fibrec(2) and fibrec(1)
  - Finally fibrec(2) needs fibrec(1) and fibrec(0)
  - So, to evaluate fibrec(5) we evaluate fibrec(4) once, fibrec(3) twice, fibrec(2) three times, fibrec(1) five times and fibrec(0) three times

Recursion

```
function fibonacci(N)
    if (N<2) then
        return N
    return [fibonacci(N-1) + fibonacci(N-2)]
```



Recursive Tree

Recursion

- The following table tabulates the number of invocations (calls) that a recursively implemented Fibonacci function invokes itself.

No.	Calls	No.	Calls	No.	Calls
1	1	8	67	15	1973
2	3	9	109	20	21,891
3	5	10	177	25	242,785
4	9	11	287	30	2,692,573
5	15	12	465	35	29,860,703
6	25	13	753	40	331,160,281
7	41	14	1219		

Recursion

- The time taken to calculate f_n with the recursive algorithm is proportional to f_n.
- It can be proved that the runtime of the recursive algorithm is $O((3/2)^n)$. [Note: the proof is out of the scope of this module. However, if you are interested, you can refer to https://en.wikipedia.org/wiki/Fibonacci_number#Relation_to_the_golden_ratio for the proof.]
- The runtime grows exponentially, the recursive algorithm is definitely not practical!

Recursion

- Fibonacci numbers - An iterative Algorithm

```
function fibiter(n)
    i = 1; j = 0
    for k = 1 to n do
        j = i + j
        i = j – i
    return j
```

- Is this solution better? What is the running time complexity?
  - The time taken to evaluate f_n is linearly proportionate to *n*; that is, the running time is O(n).

Recursion

- Comparing the algorithms (results are from empirical studies.)

N	10	20	30	50	60
fibrec	7.14 msec	3.26 msec	2.98 msec	53.22 msec	158 min
fibiter	9.78 msec	2.68 msec	7.99 msec	4.03 msec	12.96 msec

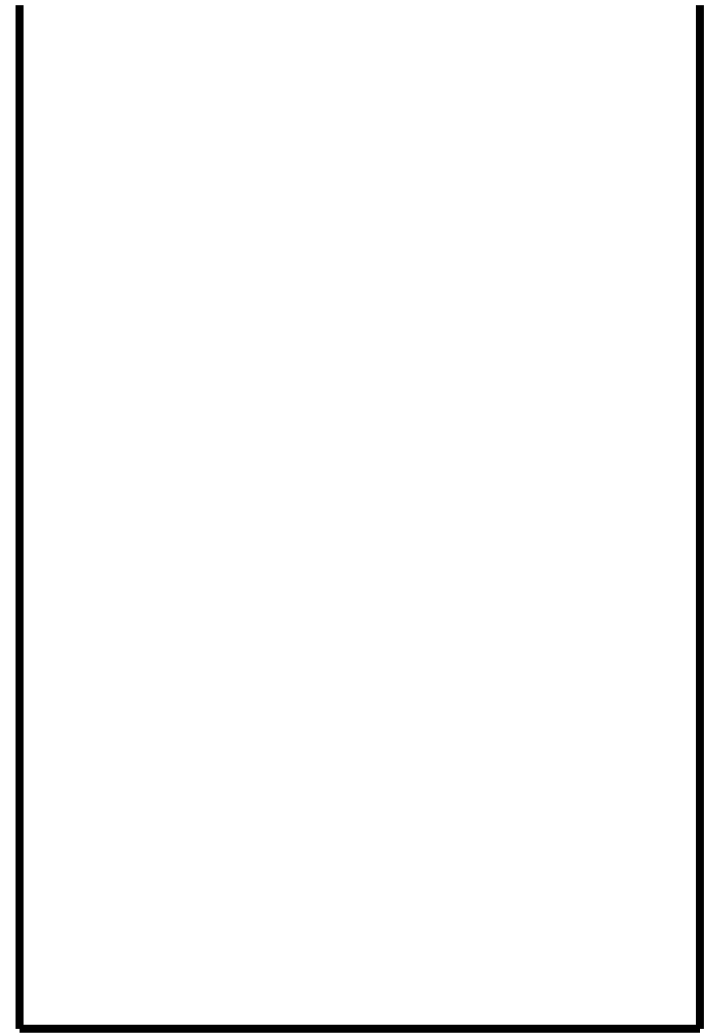
- In fact, there is an algorithm that evaluates f_n in time proportional to $\lg(n)$.

Recursion and Stacks

- Every recursive call has an Activation Record (AR) that includes the spaces needed for parameters, local variables, a return value, and a place to return control when done. This AR is created and placed on the system stack.
- When a copy of the recursively called function is completed, its AR instance (ARI) is popped / removed from the system stack.

Recursion and Stacks

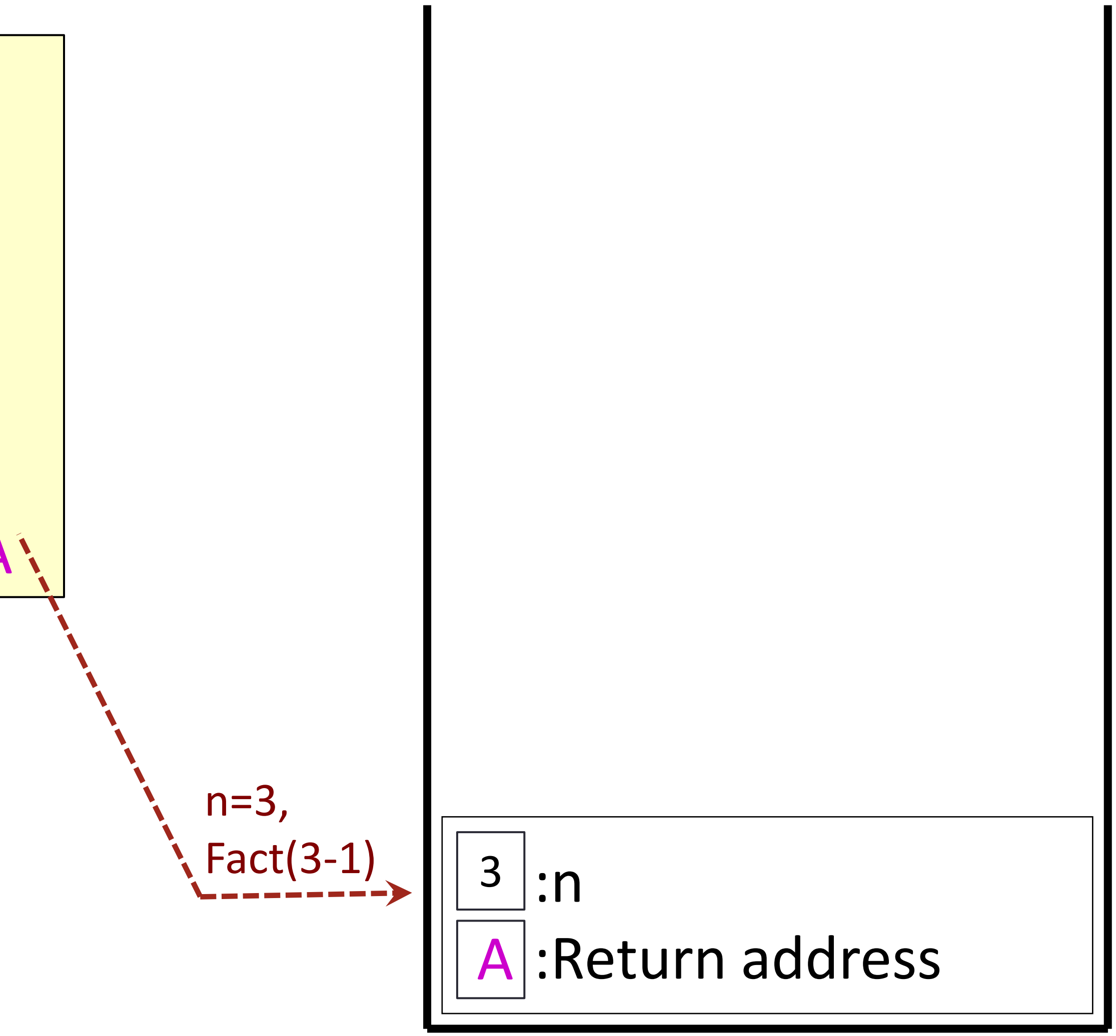
```
Algorithm: Fact(n)
{
  if n = 0
    return 1;
  else
    return n * Fact( n – 1 )
}
```



System Stack

Recursion and Stacks

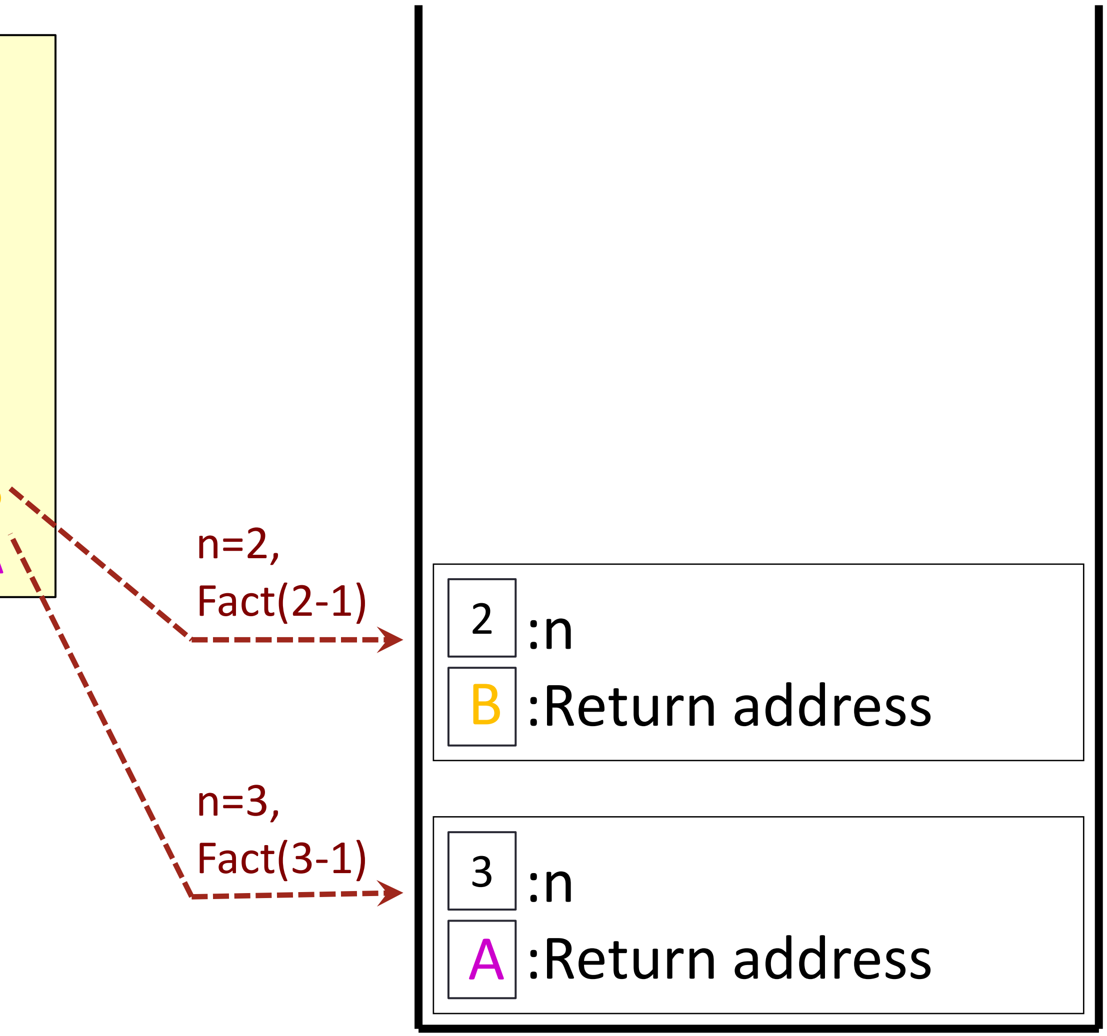
```
Algorithm: Fact(n)
{
  if n = 0
    return 1;
  else
    return n * Fact( n – 1 )
}                          A
```



System Stack

Recursion and Stacks

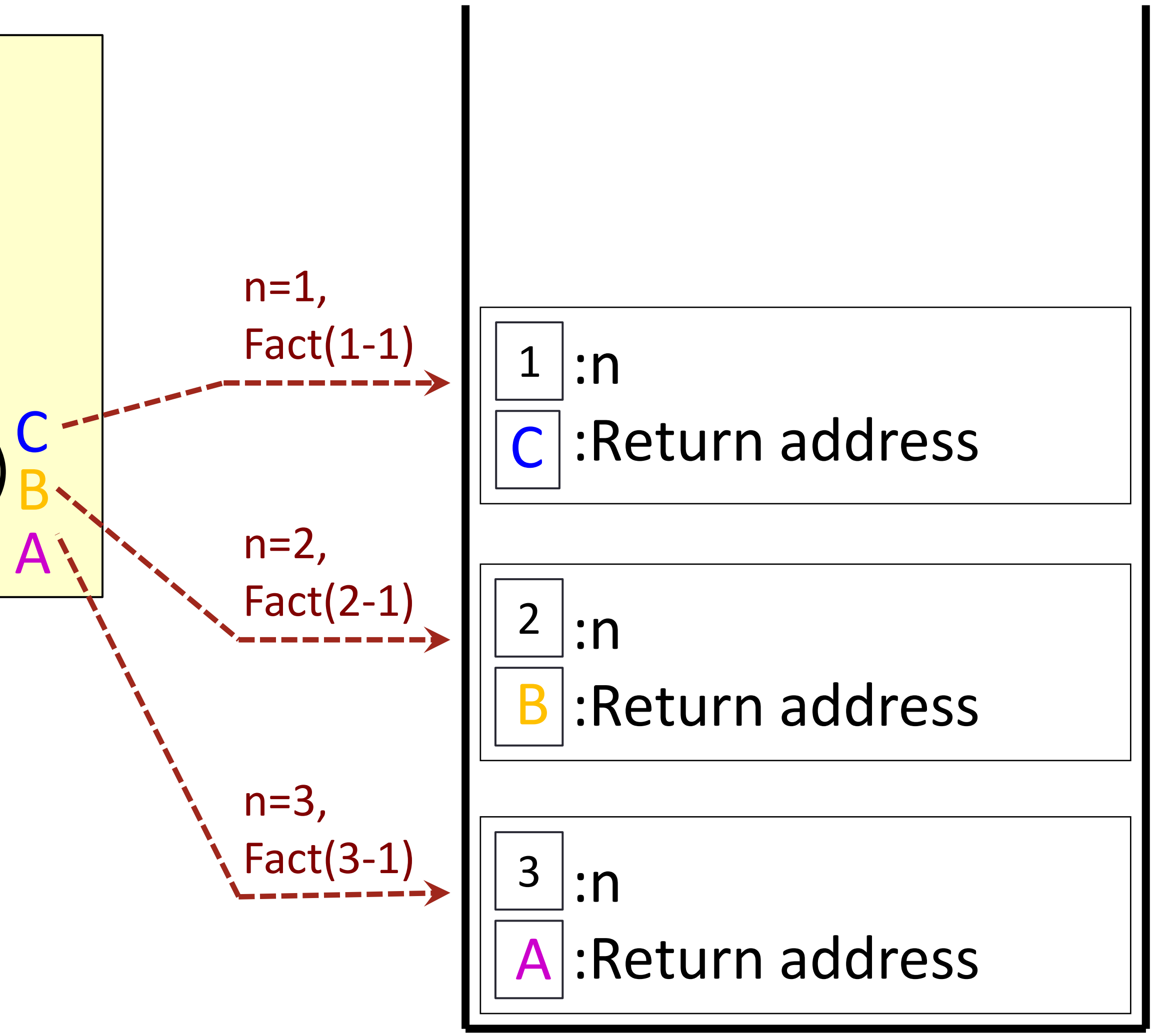
```
Algorithm: Fact(n)
{
  if n = 0
    return 1;
  else
    return n * Fact( n – 1 ) B
}                            A
```



System Stack

Recursion and Stacks

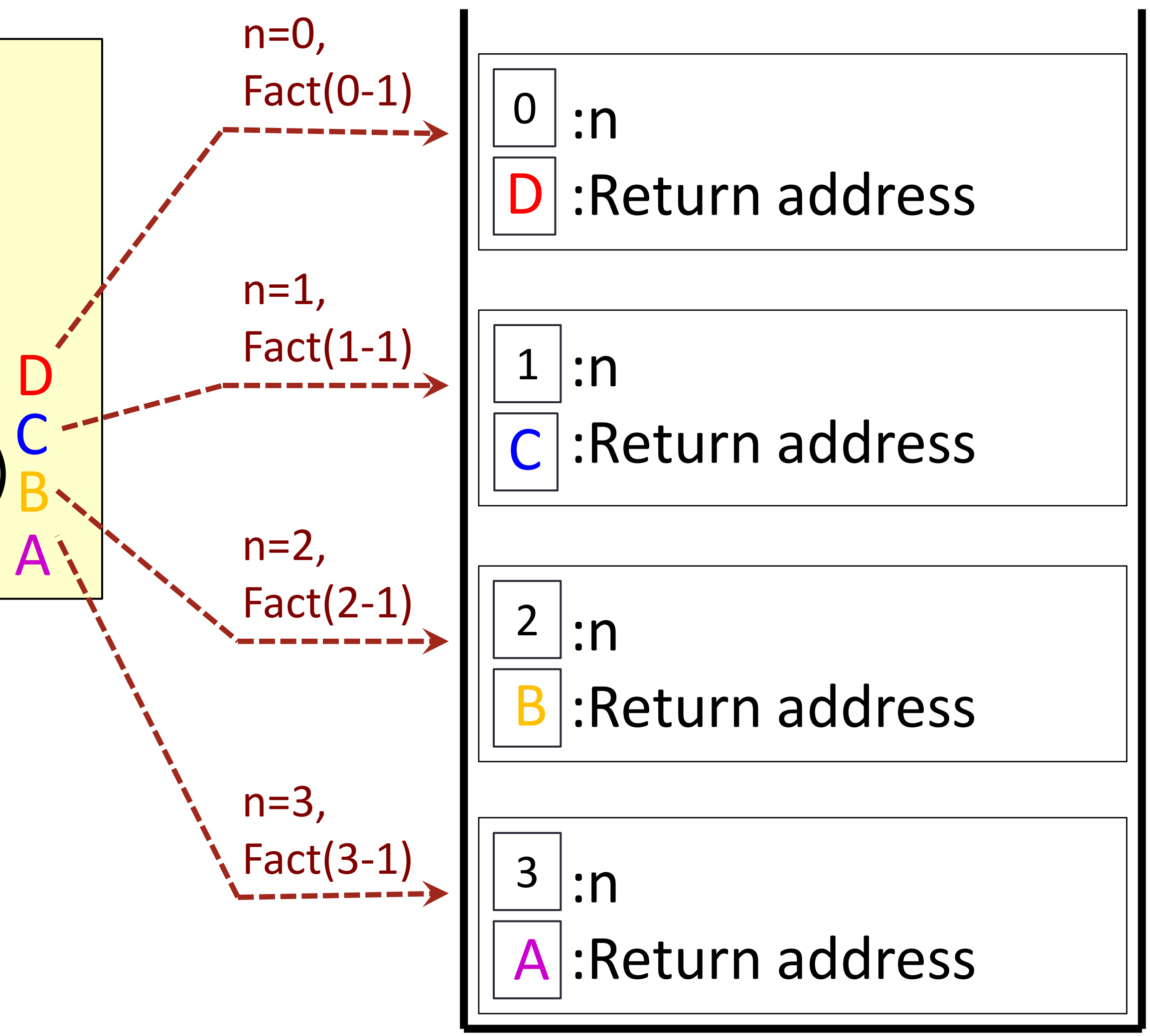
```
Algorithm: Fact(n)
{
  if n = 0
    return 1;
  else
    return n * Fact( n – 1 )
}
```



System Stack

Recursion and Stacks

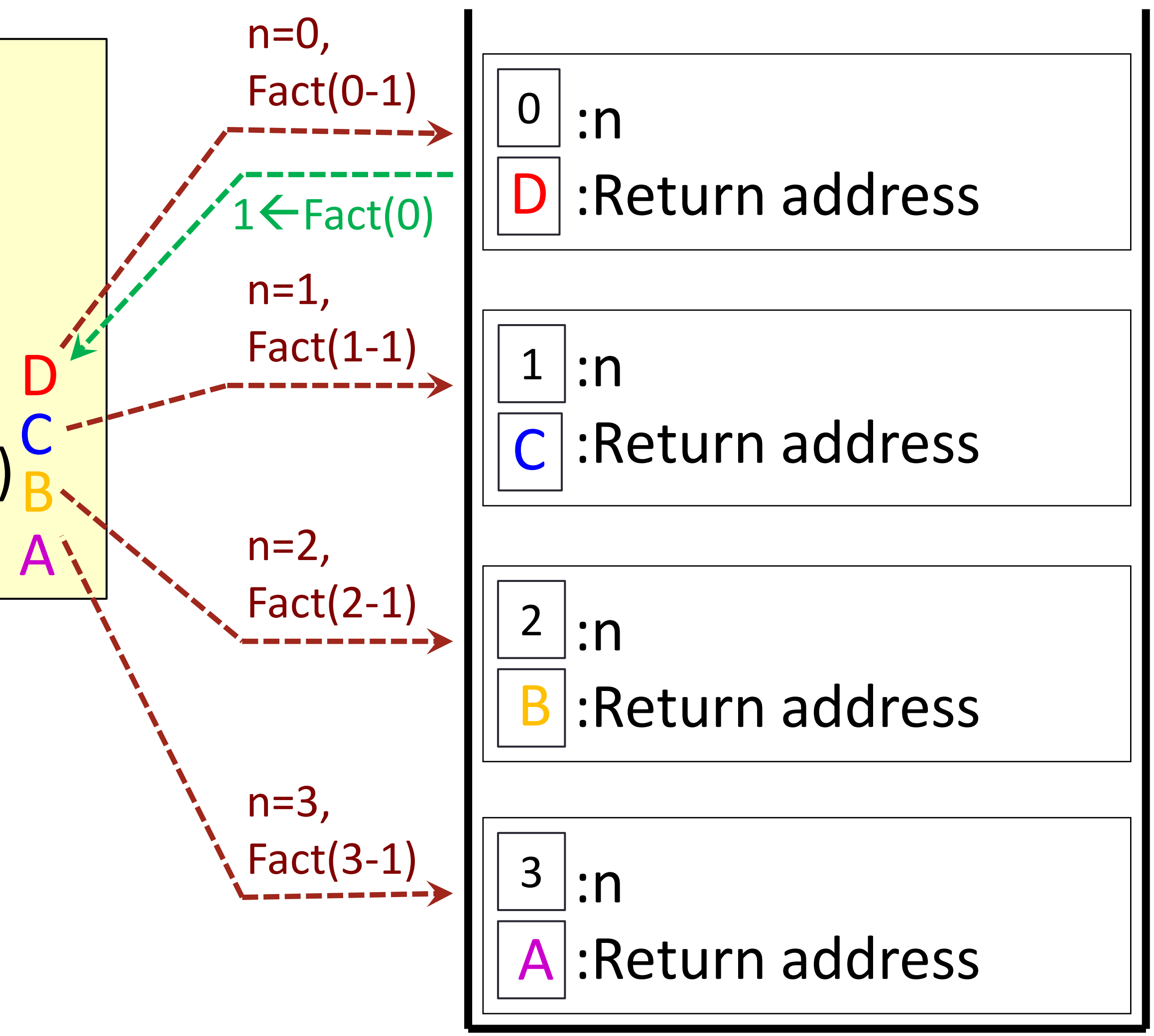
```
Algorithm: Fact(n)
{
  if n = 0
    return 1;
  else
    return n * Fact( n – 1 )
}
```



System Stack

Recursion and Stacks

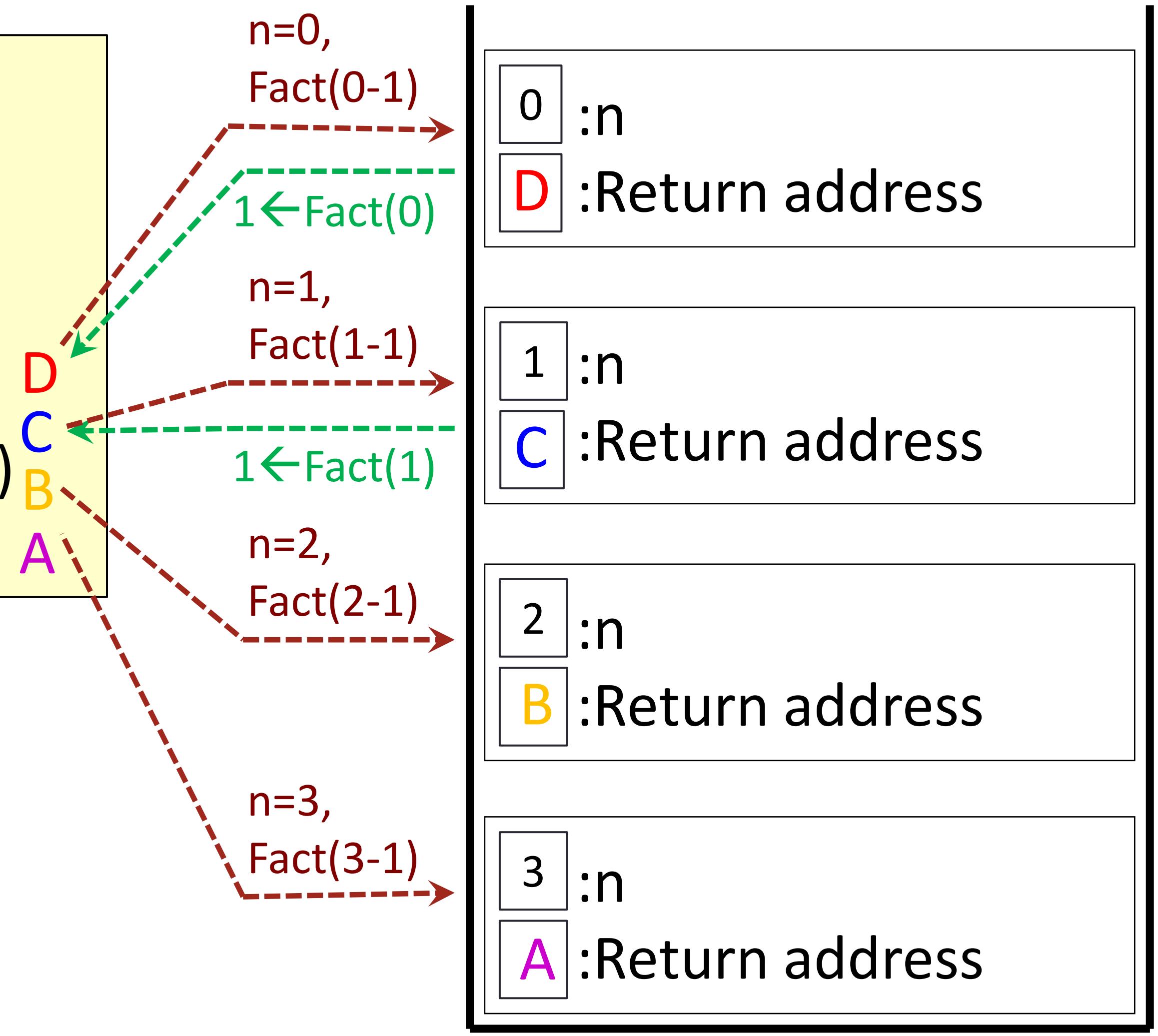
```
Algorithm: Fact(n)
{
  if n = 0
    return 1;
  else
    return n * Fact( n – 1 )
}
```



System Stack

Recursion and Stacks

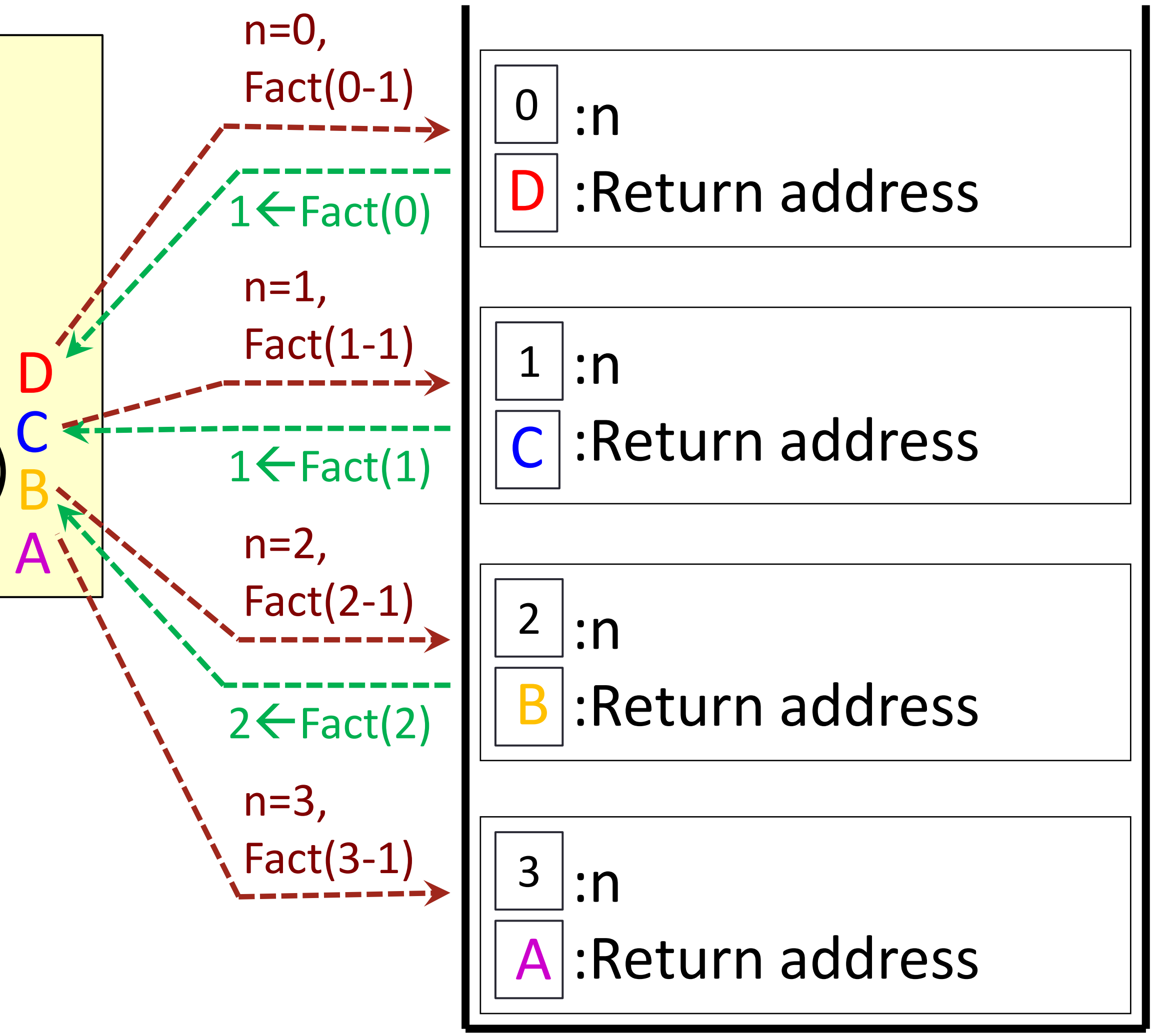
```
Algorithm: Fact(n)
{
  if n = 0
    return 1;
  else
    return n * Fact( n – 1 )
}
```



System Stack

Recursion and Stacks

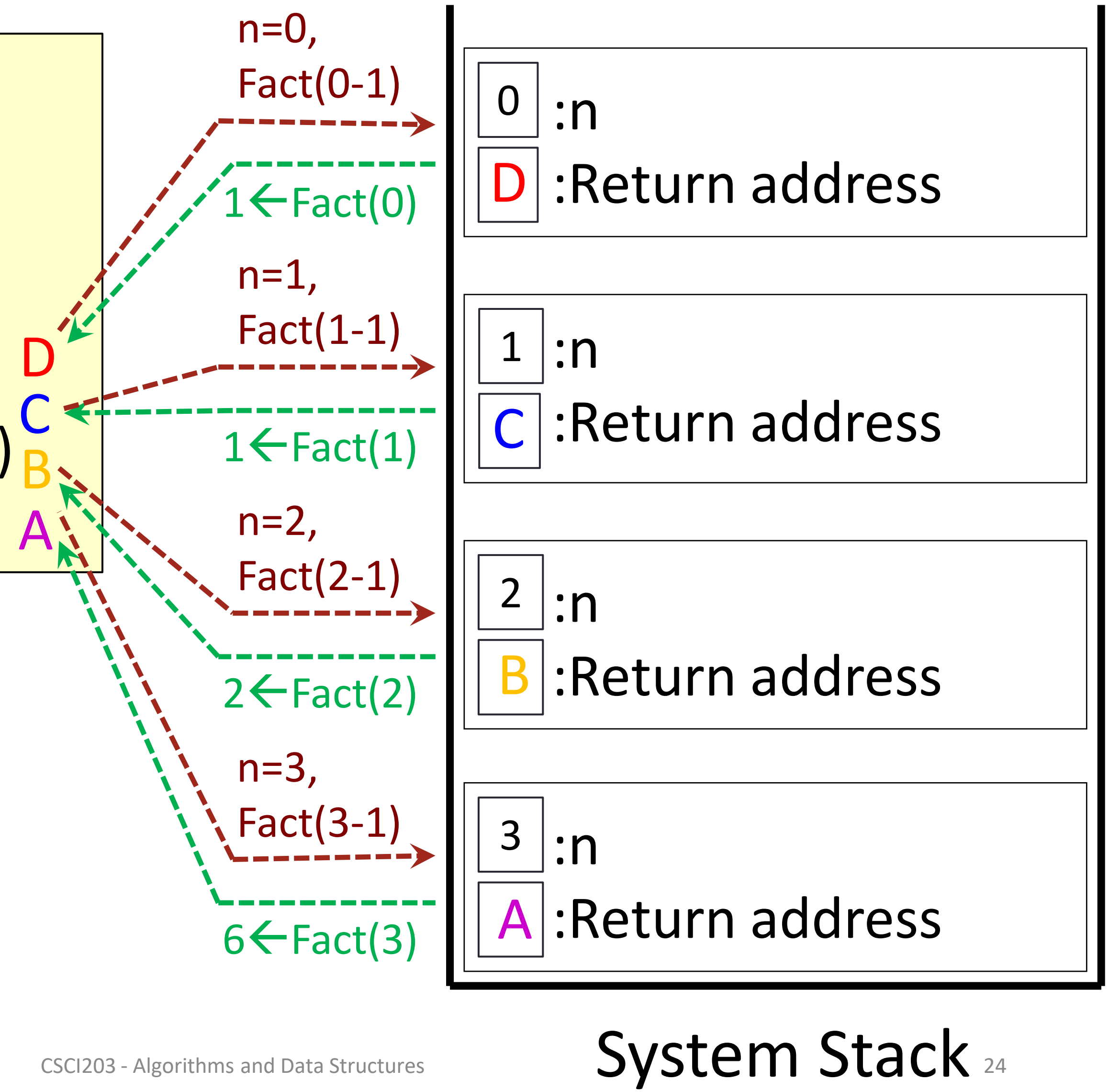
```
Algorithm: Fact(n)
{
  if n = 0
    return 1;
  else
    return n * Fact( n – 1 )
}
```



System Stack

Recursion and Stacks

```
Algorithm: Fact(n)
{
  if n = 0
    return 1;
  else
    return n * Fact( n – 1 )
}
```



Type of Recursion

- Recursion algorithms can be implemented in various forms:
  - Linear recursion
    - Linear, Tail recursion,
    - Linear, Non-tail recursion
  - Non-linear recursion
    - Non-linear, non-tail recursion.

Type of Recursion

- Linear vs Non-linear

A recursion function is **linear recursion** if each execution of it "**calls itself at most once**," that is, each activation of it involves at most one new activation.

Otherwise, it is a **non-linear recursion**.

Type of Recursion - examples

- Implementation of Fibonacci function using linear recursion

```
int FibNumL (int n, int x, int y)
{
  // Base conditions
  if (n == 1)
    return  y;
  else
  // Recursive call of function linearly
    return ( FibNumL (n – 1, y, x+y) );
}
```

Type of Recursion - examples

- Implementation of Fibonacci function using linear recursion

```
int FibNumL (int n, int x, int y)
{
  // Base conditions
  if (n == 1)
    return  y;
  else
  // Recursive call of function linearly
    return ( FibNumL (n – 1, y, x+y) );
}
```

At one time there is only one recursive function is called, hence linear.

Type of Recursion - examples

- Implementation of Fibonacci function using non-linear recursion

```
int FibNumNL (int n)
{
  // Base conditions
  if (n < 1)
    return  -1;
  if (n == 1 || n == 2)
    return  1;

  // Recursive call by binary method
  return ( FibNumNL (n – 1) + FibNumNL (n – 2) );
}
```

Type of Recursion - examples

- Implementation of Fibonacci function using non-linear recursion

```
int FibNumNL (int n)
{
  // Base conditions
  if (n < 1)
    return  -1;
  if (n == 1 || n == 2)
    return  1;

  // Recursive call by binary method
  return ( FibNumNL (n – 1) + FibNumNL (n – 2) );
}
```

At one time more than one recursive functions are called, hence non-linear.

Type of Recursion

- Tail vs Non-tail

A recursion function is **tail recursion** if each execution of it *either:*

- defines the function outright (base cases) or
- calls itself "only" with different values for arguments (or parameters).

and the recursive call is the last statement in a method.

Otherwise, it is a **non-tail recursion**.

Type of Recursion - examples

- Implementation of Factorial function using tail recursion

```
int FactorialT (int n, long accumulator)
{
  // Base conditions
  if (n == 0)
    return  accumulator;
  else
  // Recursive call linearly
    return ( FactorialT (n – 1, n * accumulator) );
}
```

Type of Recursion - examples

- Implementation of Factorial function using tail recursion

```
int FactorialT (int n, long accumulator)
{
  // Base conditions
  if (n == 0)
    return  accumulator;
  else
  // Recursive call linearly
    return ( FactorialT (n – 1, n * accumulator) );
}
```

The function calls itself with different values for arguments, and the recursive call is the last statement in the function; hence it is a tail recursion.

Type of Recursion - examples

- Implementation of Factorial function using non-tail recursion

```
int FactorialNT (long n)
{
  // Base conditions
  if (n < 0)
    return  -1;
  if (n == 0)
    return  1;
  else
  // Recursive call linearly
    return ( n * FactorialNT (n – 1) );
}
```

Type of Recursion - examples

- Implementation of Factorial function using non-tail recursion

```
int FactorialNT (long n)
{
  // Base conditions
  if (n < 0)
    return  -1;
  if (n == 0)
    return  1;
  else
  // Recursive call linearly
    return ( n * FactorialNT (n – 1) );
}
```

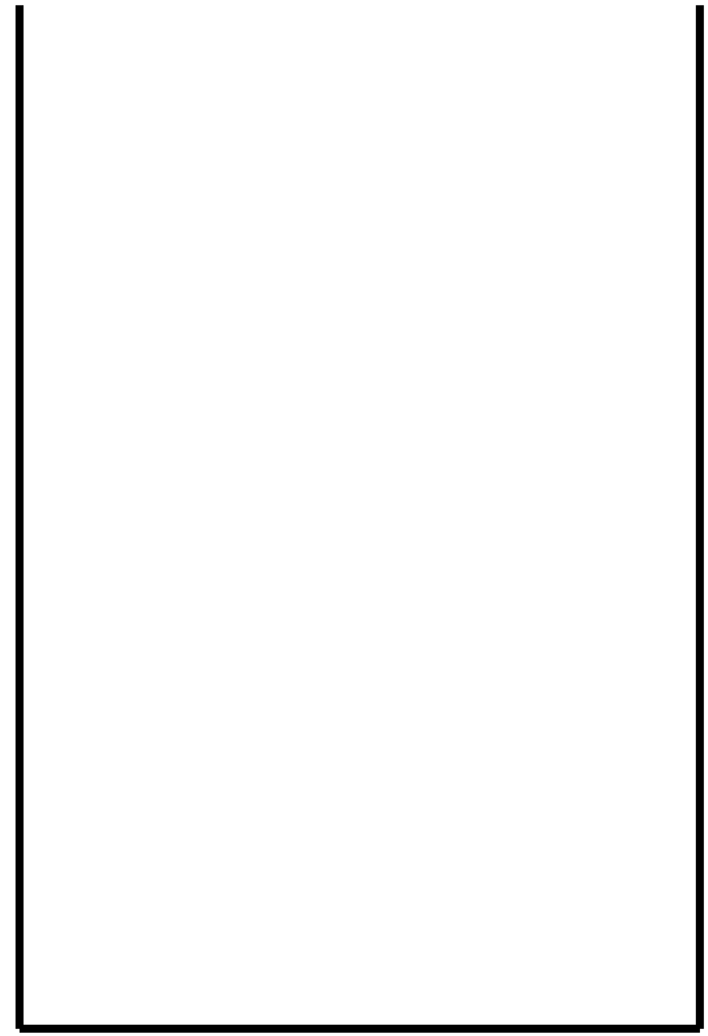
The function calls itself with different values for arguments, however, there is still multiplication operation is needed; hence it is a non-tail recursion.

Recursion and Stacks

Activation records of tail-recursion function

X = Fact (3,1);

```
Algorithm: Fact(int n, long a)
{
  if n == 0
    return a;
  else
    return Fact( n – 1, n * a )
}
```



System Stack

Recursion and Stacks

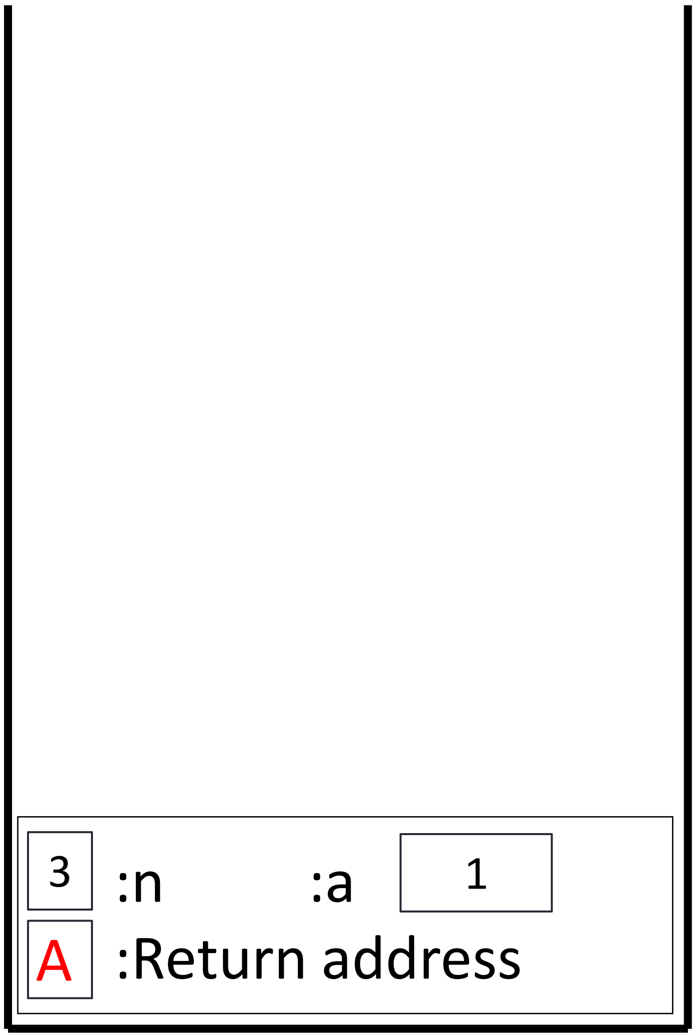
Activation records of
tail-recursion function

X = Fact (3,1);

```
Algorithm: Fact(int n, long a)
{
  if n == 0
    return a;
  else
    return Fact( n – 1, n * a )
}
```

n=3, a=1
Fact(3-1, 3 * 1)

A



System Stack

Recursion and Stacks

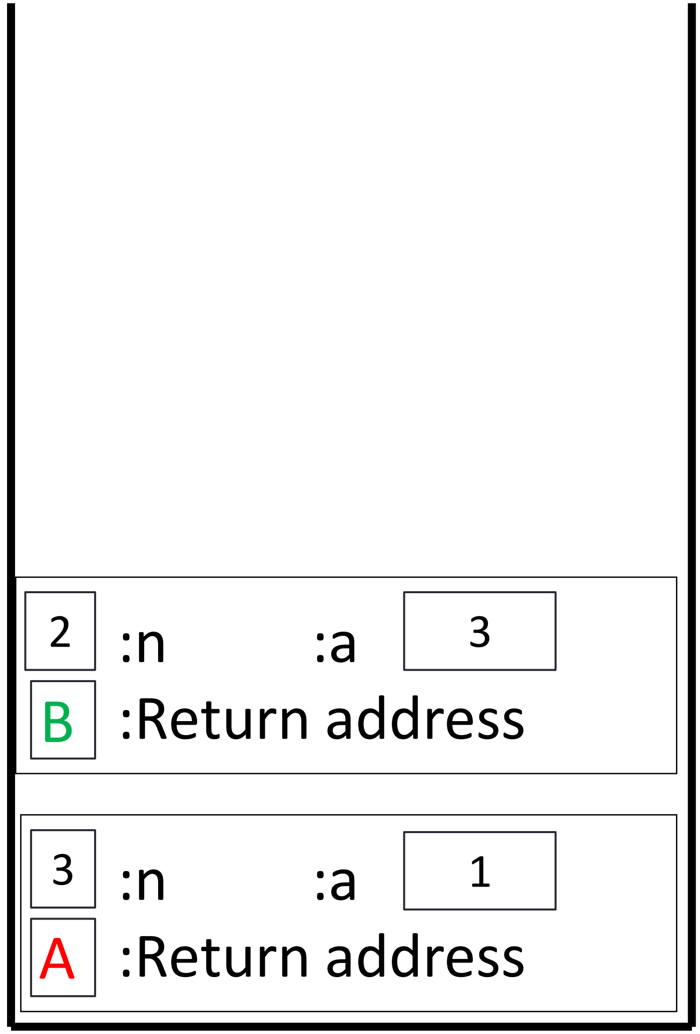
Activation records of
tail-recursion function

X = Fact (3,1);

```
Algorithm: Fact(int n, long a)
{
  if n == 0
    return a;
  else                          B
    return Fact( n – 1, n * a )
}                              A
```

n=2, a=3
Fact(2-1, 2 * 3)

n=3, a=1
Fact(3-1, 3 * 1)



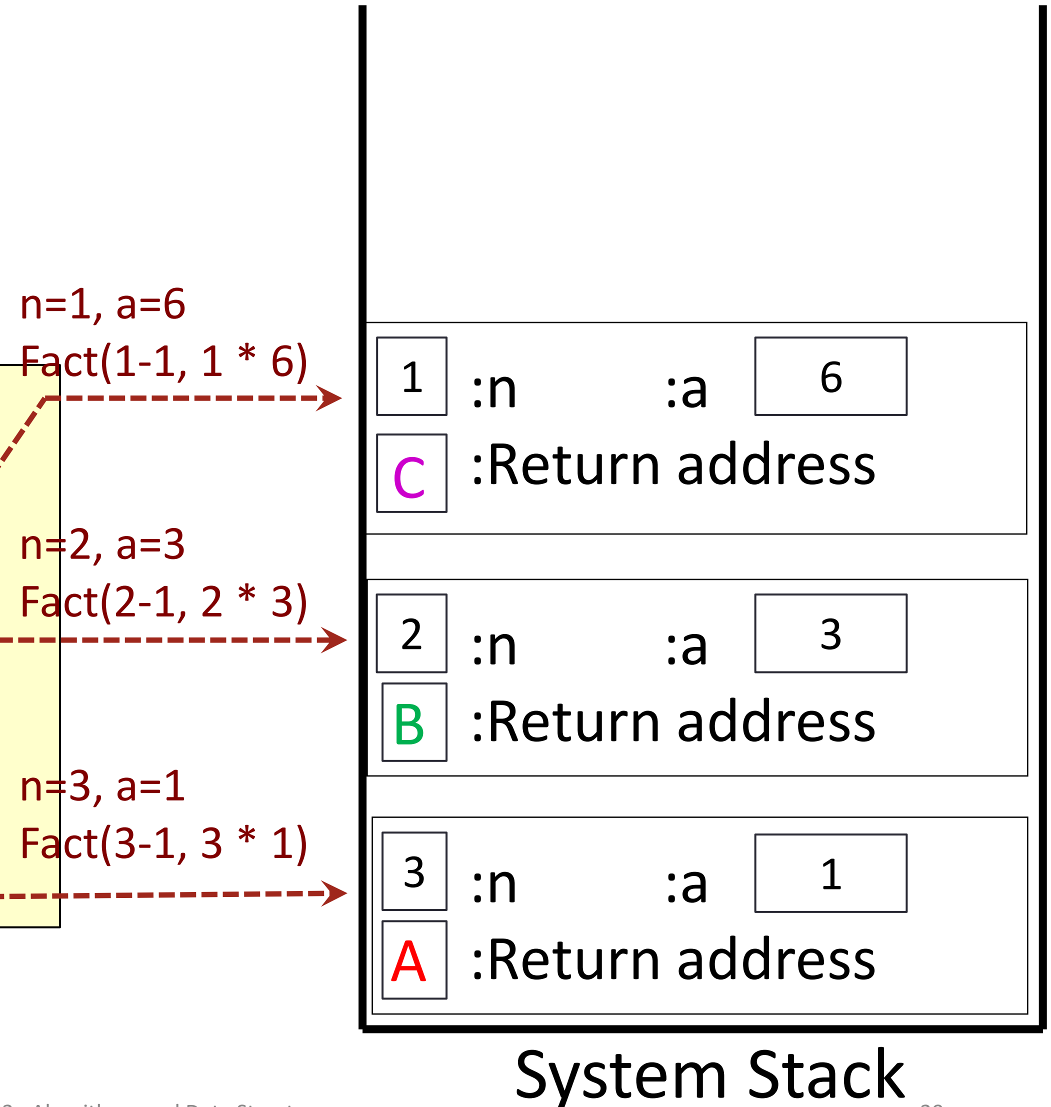
System Stack

Recursion and Stacks

Activation records of tail-recursion function

X = Fact (3,1);

```
Algorithm: Fact(int n, long a)
{
  if n == 0
    return a;
  else                         C   B
    return Fact( n – 1, n * a )
}                                  A
```

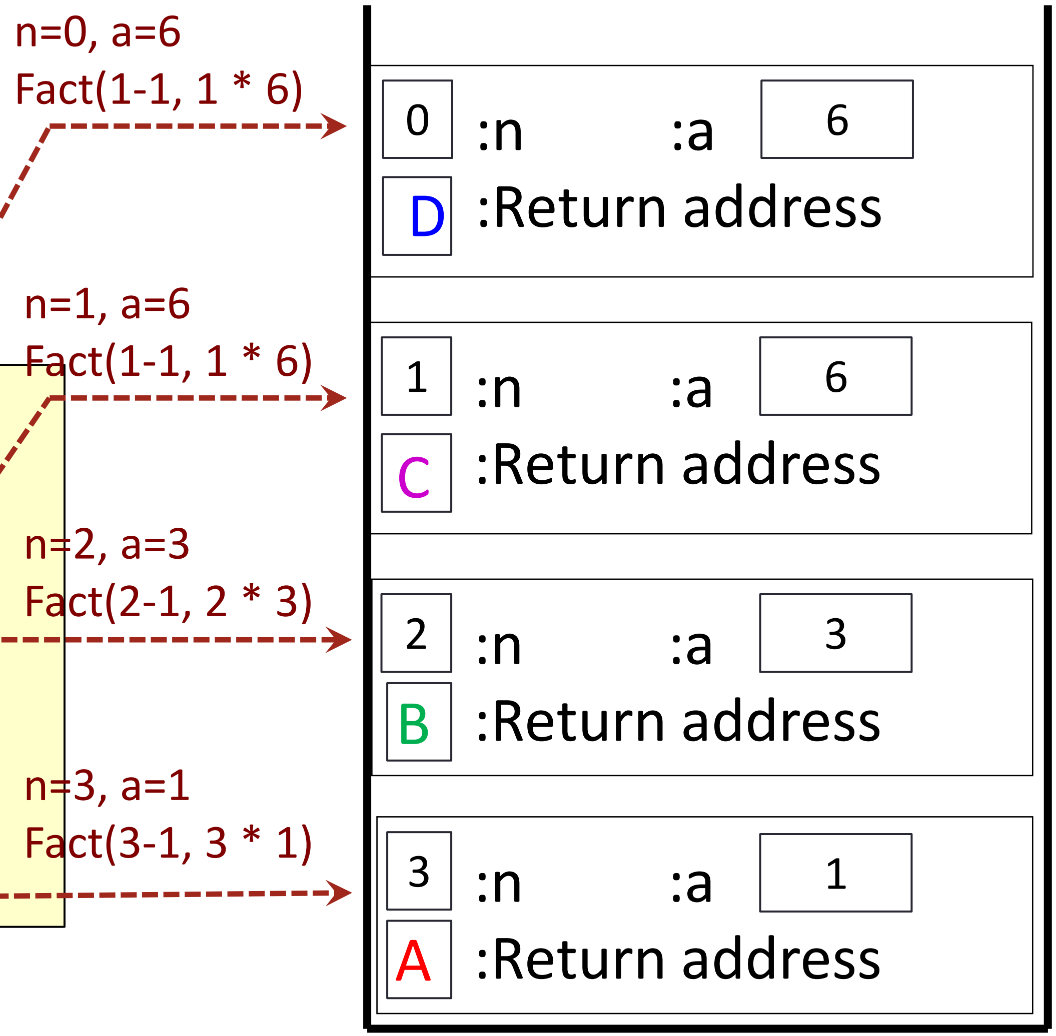


Recursion and Stacks

Activation records of tail-recursion function

X = Fact (3,1);

```
Algorithm: Fact(int n, long a)
{
  if n == 0
    return a;
  else                    D   C   B
    return Fact( n – 1, n * a )
}                               A
```



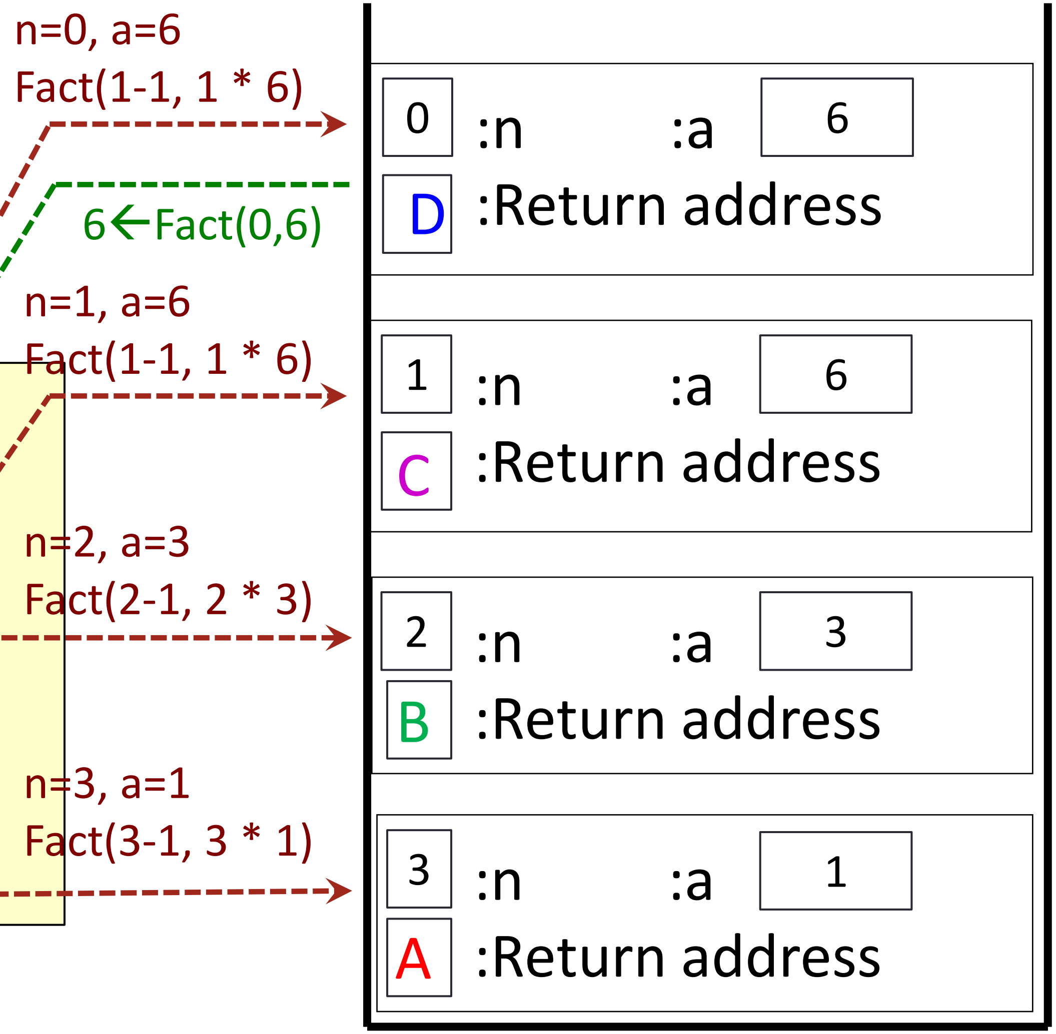
System Stack

Recursion and Stacks

Activation records of tail-recursion function

X = Fact (3,1);

```
Algorithm: Fact(int n, long a)
{
  if n == 0
    return a;
  else                  D   C   B
    return Fact( n – 1, n * a )
}                           A
```



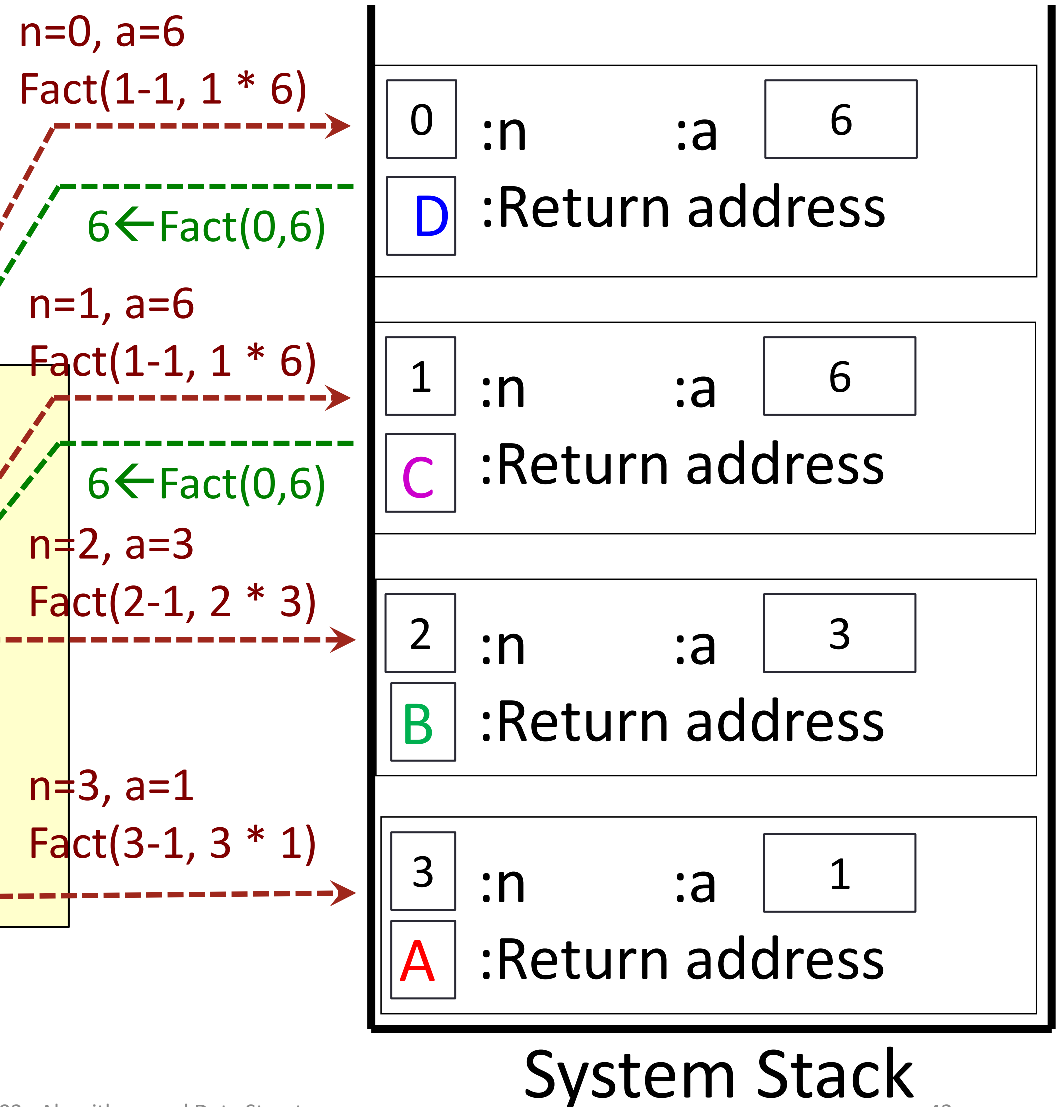
System Stack

Recursion and Stacks

Activation records of tail-recursion function

X = Fact (3,1);

```
Algorithm: Fact(int n, long a)
{
  if n == 0
    return a;
  else
    return Fact( n – 1, n * a )
}
```

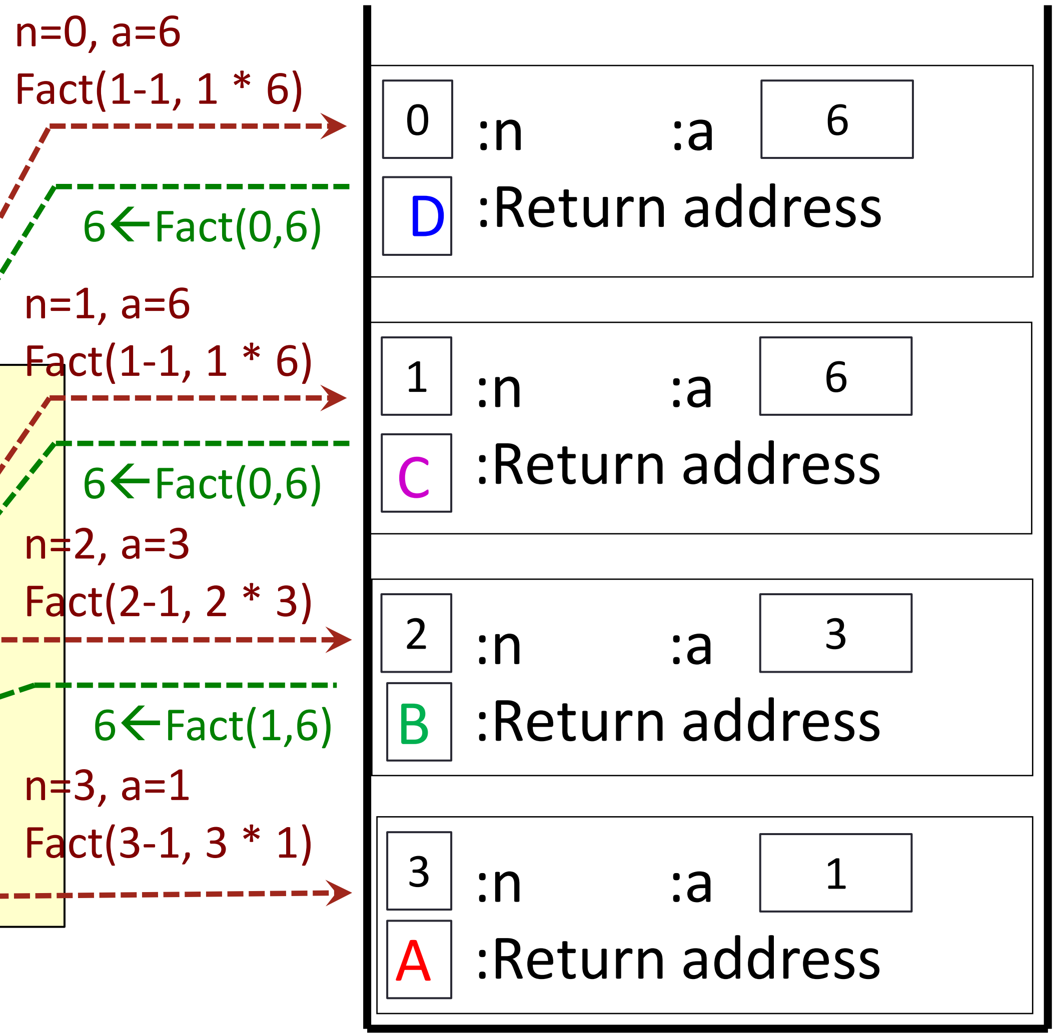


Recursion and Stacks

Activation records of tail-recursion function

X = Fact (3,1);

```
Algorithm: Fact(int n, long a)
{
  if n == 0
    return a;
  else                    D    C   B
    return Fact( n – 1, n * a )
}                              A
```



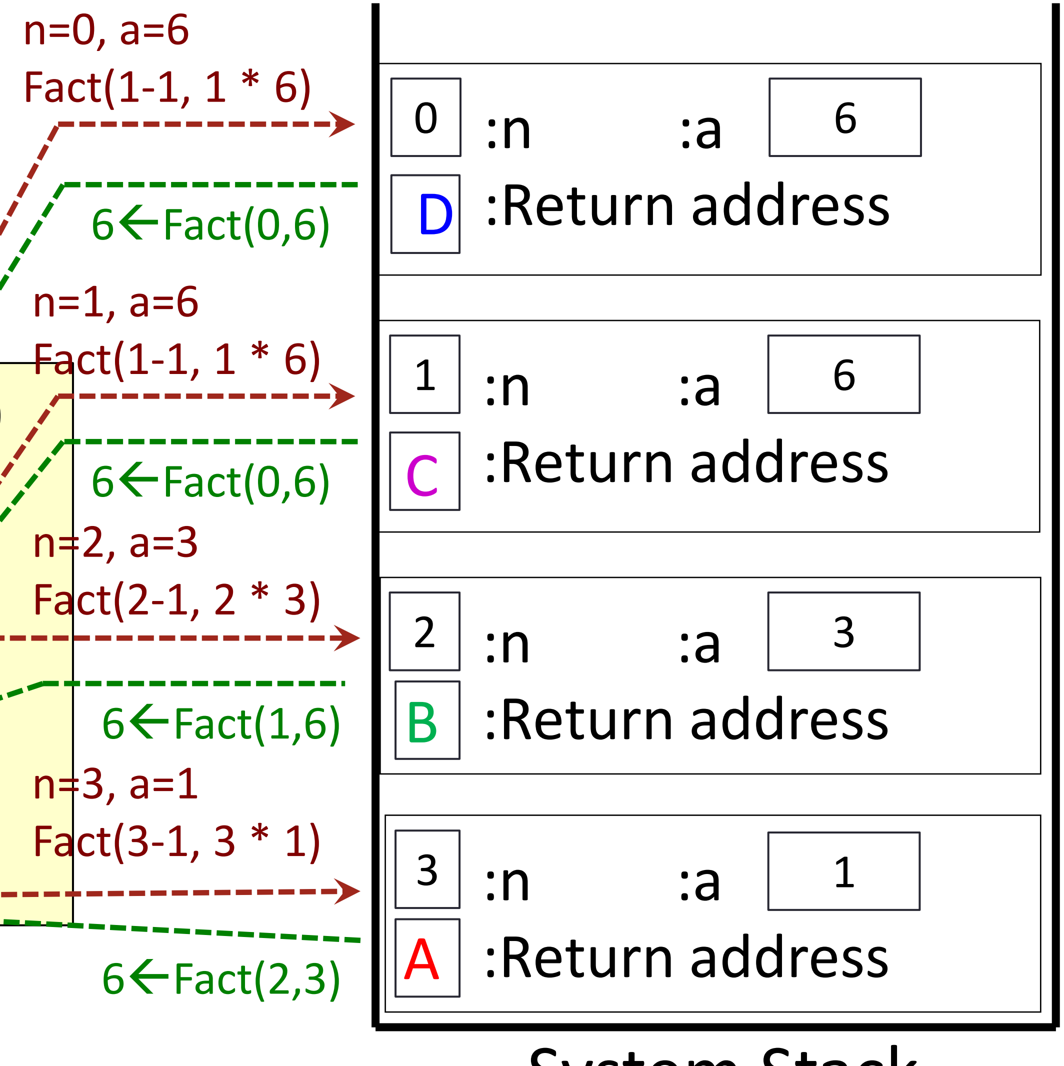
System Stack

Recursion and Stacks

Activation records of tail-recursion function

X = Fact (3,1);

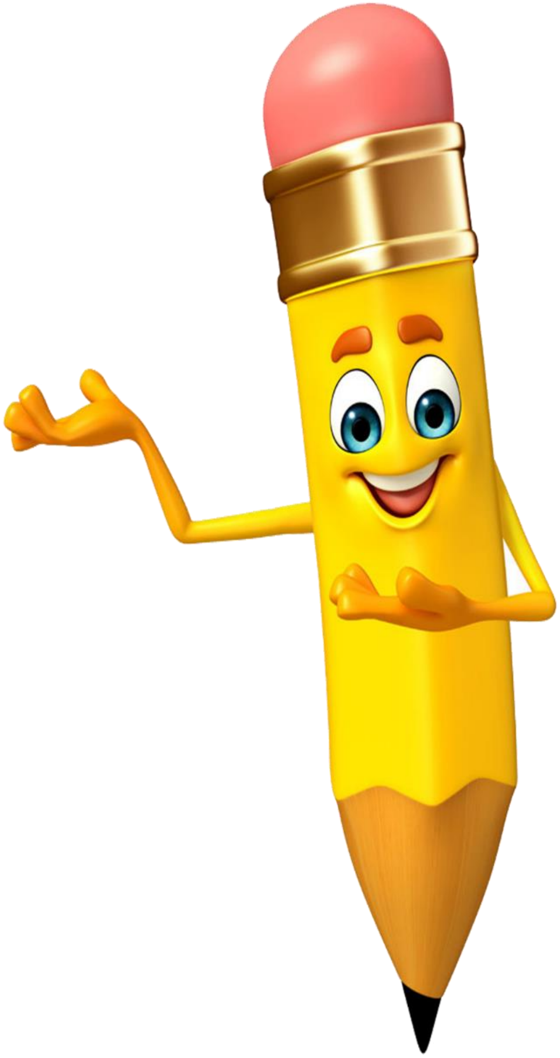
```
Algorithm: Fact(int n, long a)
{
  if n == 0
    return a;
  else                      D    C   B
    return Fact( n – 1, n * a )
}                                 A
```



6 ← Fact (3,1);

Recurrence Relation

$$T(0) = 2$$
$$T(1) = 2$$
$$T(n) = 2T(n-1) + 1$$



What is a recurrence relation?

- A recurrence relation, denoted as $T(n)$, is a recursive form of equation that involves some constant and integer variable n.
- Like all recursive functions, it has both recursive case and base case.
- For example:

$$f(n) = \begin{cases} 0, & x = 0 \\ 1, & x = 1 \\ f_{n-1} + f_{n-2}, & x \geq 2 \end{cases}$$

$$T(0) = 2$$
$$T(1) = 2$$
$$T(n) = 2T(n-1) + 1$$

What is a recurrence relation?

- A recurrence relation, denoted as $T(n)$, is a recursive form of equation that involves some constant and integer variable n.
- Like all recursive functions, it has both recursive case and base case.
- For example:

Recurrence relations

$$f(n) = \begin{cases} 0, & x = 0 \\ 1, & x = 1 \\ f_{n-1} + f_{n-2}, & x \geq 2 \end{cases}$$

$$T(0) = 2$$
$$T(1) = 2$$
$$T(n) = 2T(n-1) + 1$$

What is recurrence relation?

$T(0) = 2$
$T(1) = 2$
$T(n) = 2T(n-1) + 1$

What is recurrence relation?

$$T(0) = 2$$
$$T(1) = 2$$
$$T(n) = 2T(n-1) + 1$$

The term(s) of the relation that does not contain T is called the **base case** of the recurrence relation.

What is recurrence relation?

$$T(0) = 2$$
$$T(1) = 2$$
$$T(n) = 2T(n-1) + 1$$

The term(s) of the relation that does not contain T is called the **base case** of the recurrence relation.

The term(s) of the relation that contains T is called the **recurrent or recursive case**.

What is recurrence relation?

$$T(0) = 2$$
$$T(1) = 2$$
$$T(n) = 2T(n-1) + 1$$

The term(s) of the relation that does not contain T is called the **base case** of the recurrence relation.

The term(s) of the relation that contains T is called the **recurrent or recursive case**.

- Recurrence relations are useful for expressing the running times, that is, the number of basic operations executed, of recursive algorithms.

Forming recurrence relations

- For a given recursive method, the base case and the recursive case of its recurrence relation correspond directly to the base case and the recursive case of the method.
- For example, what is the recurrence relation for the following recursive function?

```
public void f (int n) {
    if (n == 0)
        return 1;
    else
        return n * f (n – 1);
}
```

Forming recurrence relations

- For a given recursive method, the base case and the recursive case of its recurrence relation correspond directly to the base case and the recursive case of the method.
- For example, what is the recurrence relation for the following recursive function?

```
public void f (int n) {
    if (n == 0)
        return 1;
    else
        return n * f (n – 1);
}
```

What function is this? ☺

Forming recurrence relations

```
public void f (int n) {
     if (n = 0)
          return 1;
     else
          return n * f (n – 1);
}
```

Forming recurrence relations

```
public void f (int n) {
    if (n = 0)
        return 1;
    else
        return n * f (n – 1);
}
```

The base case is reached when $n = 0$. The function performs **ONE** comparison and a return statement. Hence, the number of operation is some constant, that is $\boldsymbol{T(0) = a}$.

Forming recurrence relations

```
public void f (int n) {
    if (n = 0)
        return 1;
    else
        return n * f (n – 1);
}
```

The base case is reached when $n = 0$. The function performs **ONE** comparison and a return statement. Hence, the number of operation is some constant, that is $\boldsymbol{T(0) = a}$.

When $n > 0$, the function performs **ONE** recursive call with a parameter $n\text{–}1$, then a multiplication and a return statement. Hence, the number of operations consists of a recursive term plus some constant, that is $\boldsymbol{T(n) = T(n-1) + b}$.

Forming recurrence relations

```
public void f (int n) {
    if (n = 0)
        return 1;
    else
        return n * f (n – 1);
}
```

The base case is reached when $n = 0$. The function performs **ONE** comparison and a return statement. Hence, the number of operation is some constant, that is $\boldsymbol{T(0) = a}$.

When $n > 0$, the function performs **ONE** recursive call with a parameter $n–1$, then a multiplication and a return statement. Hence, the number of operations consists of a recursive term plus some constant, that is $\boldsymbol{T(n) = T(n-1) + b}$.

Hence, the recurrence relations are:

$T(0) = a$ for some constant a

$T(n) = T(n-1) + b$ for a constant b and a recursive term

Forming recurrence relations

- Another example:

```
public int myFunction (int n) {
    if (n == 1)
        return 1;
    else
        return 2 * myFunction (n / 2) + myFunction (n / 2) + 1;
}
```

- The base case is reached when $n == 1$. The function performs one comparison and **ONE** return statement. Hence, $T(1)$ is some constant c.
- When $n > 1$, the function performs **TWO** recursive calls, each with the parameter $n/2$, and some constant number of basic operations.

Forming recurrence relations

Hence, the recurrence relations are:

$T(1) = c$ for some constant c

$T(n) = 2T\left(\frac{n}{2}\right) + b$ for a constant b and a recursive term

Solving recurrence relations

- There are four methods to solve recurrence relations that represent the running time of recursive methods:
  - Recursion tree method
  - **Iteration method (unrolling and summing)**
  - **Master method**
  - Substitution method

- We will discuss the iteration method and the master method here.

Solving recurrence relations – Iteration method

Steps:

- Expand the recurrence relation
- Express the expansion as a summation by plugging the recurrence back into itself until we see a pattern.
- Evaluate the summation.

Solving recurrence relations – Iteration method

Example:

Form and solve the recurrence relation for the running time of factorial function and hence determine its big-O complexity:

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

Solving recurrence relations – Iteration method

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

Solving recurrence relations – Iteration method

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

$$T(0) = c$$

$$\begin{aligned} T(n) &= b + T(n-1) \\ &= b + b + T(n-2) \\ &= b + b + b + T(n-3) \\ &\ldots \\ &= kb + T(n-k) \end{aligned}$$

Solving recurrence relations – Iteration method

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

$$T(0) = c$$

$$\begin{aligned} T(n) &= b + T(n - 1) \\ &= b + b + T(n - 2) \\ &= b + b + b + T(n - 3) \\ &\dots \\ &= kb + T(n - k) \end{aligned}$$

When $k = n$, we have:

$$T(n) = nb + T(n - n)$$
$$= bn + T(0)$$
$$T(n) = bn + c \text{ since } T(0) = c$$

Solving recurrence relations – Iteration method

What does $T(n) = bn + c$ indicate?

Solving recurrence relations – Iteration method

What does $T(n) = bn + c$ indicate?

The running time complexity of the factorial function is $O(n)$.

Analysis of recursion binary search

- Search for x in a sorted array A.

$$
\begin{aligned}
&\text{Binary} - \text{Search}(A, p, q, x) \\
&\quad \text{if } p \geq q \text{ return } -1; \\
&\quad r = \lfloor (p+q)/2 \rfloor \\
&\quad \text{if } x = A[r] \text{ return } r \\
&\quad \text{else if } x < A[r] \text{ Binary} - \text{Search}(A, p, r-1, x) \\
&\quad \text{else Binary} - \text{Search}(A, r+1, q, x)
\end{aligned}
$$

- The initial call is Binary-Search(A, 1, n, x)

Recursion

Write down the **recurrence equation** which describes the running time of the algorithm Binary-Search(*A, 1, n, x*) as a function of *n*.

$$\text{Binary} - \text{Search}(A, p, q, x)$$
$$\quad \text{if } p \geq q \text{ return } -1;$$
$$\quad r = \lfloor (p+q)/2 \rfloor$$
$$\quad \text{if } x = A[r] \text{ return } r$$
$$\quad \text{else if } x < A[r] \text{ Binary} - \text{Search}(A, p, r-1, x)$$
$$\quad \text{else Binary} - \text{Search}(A, r+1, q, x)$$

Recursion

Write down the **recurrence equation** which describes the running time of the algorithm Binary-Search(*A, 1, n, x*) as a function of *n*.

$\text{Binary} - \text{Search}(A, p, q, x)$

$\quad \text{if } p \geq q \text{ return } -1;$

$\quad r = \lfloor (p+q)/2 \rfloor$

$\quad \text{if } x = A[r] \text{ return } r$

$\quad \text{else if } x < A[r] \text{ Binary} - \text{Search}(A, p, r-1, x)$

$\quad \text{else Binary} - \text{Search}(A, r+1, q, x)$

$$T(n) = c \qquad if\ n = 1$$

$$T(n) = k + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \qquad if\ n > 0$$

Recursion

What is the upper-bound (big-O) complexity of the algorithm Binary-Search(*A, 1, n, x*)?

$$T(n) = b + T\left(\frac{n}{2}\right)$$

$$= b + b + T\left(\frac{n}{(2)^2}\right)$$

$$= b + b + b + T\left(\frac{n}{(n)^3}\right)$$

$$\dots$$

$$= kb + T\left(\frac{n}{(2)^k}\right)$$

Recursion

when $\frac{n}{2^k} = 1$, we have $n = 2^k$, and this happen when $p = q$.

Since $n = 2^k$, we have $k = \lg n$ (Note: lg n = $\log_2 n$)

Substituting k with $\lg n$ into $T(n) = kb + T\left(\frac{n}{(2)^k}\right)$, we have $T(n) = (\lg n)b + T\left(\frac{n}{(2)^{\lg n}}\right)$.

Recursion

Hence the total cost $T(n) = b(\lg n) + T\left(\frac{n}{2^{\lg n}}\right)$

$= b(\lg n) + T(1)$ (Note: $2^{\lg n} = n$)

$= b(\lg n) + c$

$\Rightarrow O(\lg n)$

Solving recurrence relations – Master method

- The Master method is a general method for solving recurrence relations that arise frequently in divide and conquer algorithms, which have the following form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Where:

- a is a constant and ≥ 1
- b is a constant and > 1
- $f(n)$ is a function of $\text{non} - \text{negative integer}\ n$.

Solving recurrence relations – Master method

- The constant a is the number of sub-problems into which a problem of size n was divided, and for each division, it is divided into b.
- For example, a list of size n is divided into 5 sub-lists, and each time a list or sub-list is divided into two. Hence $a = 5$ and $b = 2$.

Solving recurrence relations – Master method

- Recurrence relation of the form $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ can be bounded asymptotically as follows:

Case 1: If $f(n) = O\left(n^{\log_b a-\varepsilon}\right)$ for some constant $\varepsilon > 0$, then $\boldsymbol{T(n) = \Theta\left(n^{\log_b a}\right)}$

Solving recurrence relations – Master method

Case 2: If $f(n) = \Theta\left(n^{\log_b a}\right)$, then, then $\boldsymbol{T(n) = \Theta\left(n^{\log_b a}\, lg\, n\right)}$

Case 3: If $f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right)$ for some constant $\varepsilon > 0$, and if $af\left(\frac{n}{b}\right) \leq cf(n)$ for some constant $c < 1$, and all sufficiently large n, then $\boldsymbol{T(n) = \Theta\left(f(n)\right)}$.

Solving recurrence relations – Master method

- For a recurrence relation, we test to verify if the function $f(n)$ satisfies one of the following:
  - Case 1: $f(n) \in O\left(n^{log_b a - \varepsilon}\right)$ for some $\varepsilon > 0$.
  - Case 2: $f(n) \in \Theta\left(n^{log_b a}\right)$
  - Case 3: $f(n) \in \Omega\left(n^{log_b a + \varepsilon}\right)$ for some $\epsilon > 0$, and if $a \times f\left({}^{n}/_{b}\right) \leq c \times f(n)$ for some constant $c < 1$, and all sufficiently large n.

- If case 1 is satisfied, then $T(n) = \Theta\left(n^{log_b a}\right)$.
- If case 2 is satisfied, then $T(n) = \Theta\left(n^{log_b a}\, lg\, n\right)$.
- If case 3 is satisfied, then $T(n) = \Theta(f(n))$.

Solving recurrence relations – Master method

The master theorem cannot be applied if:

- $T(n)$ is not monotone, e.g., $T(n) = \sin n$.
- $f(n)$ is not polynomial, e.g., $T(n) = 2T(n/2) + 2^n$.
- b cannot be expressed as a constant, e.g., $T(n) = T(\sqrt{n})$.

- Another important note: the Master Theorem does not solve a recurrence relation.

Solving recurrence relations – Master method

- For example, from the earlier analysis of the Binary Search algorithm, we have

$$T(n) = c + T\left(\frac{n}{2}\right)$$

Solving recurrence relations – Master method

- For example, from the earlier analysis of the Binary Search algorithm, we have

$$T(n) = c + T\left(\frac{n}{2}\right)$$

$$\boxed{T(n) = aT\left(\frac{n}{b}\right) + f(n)}$$

Using Master method, $a = 1, b = 2$, and $f(n) = c$, we can determine the run-time complexity of a Binary Search algorithm as follow:

Solving recurrence relations – Master method

- For example, from the earlier analysis of the Binary Search algorithm, we have

$$T(n) = c + T\left(\frac{n}{2}\right)$$

$$\boxed{T(n) = aT\left(\frac{n}{b}\right) + f(n)}$$

Using Master method, $a = 1, b = 2$, and $f(n) = c$.

Test for case 1:

Is $f(n) = c \in O\left(n^{\log_b a - \varepsilon}\right)$

$\in O\left(n^{\log_2 1 - \varepsilon}\right)$

$\in O(n^{0-\varepsilon})$ for some $\varepsilon > 0$?

Solving recurrence relations – Master method

Since $\varepsilon > 0$, then $f(n) = c \notin O(n^{0-\varepsilon})$. Hence, the first form of the Master theorem cannot be used.

Solving recurrence relations – Master method

Test for case 2:

Is $f(n) = c \in \Theta\left(n^{log_b a}\right) \in \Theta\left(n^{log_2 1}\right) \in \Theta(n^0)$
$\in \Theta(1)$?

Since $f(n) = c \in \Theta(1)$, the second form of the Master theorem can be used, that is,
$T(n) = \Theta\left(n^{log_b a} \lg n\right) = \Theta(n^0 \lg n) = \Theta(\lg n)$

Solving recurrence relations – Master method

Test for case 2:

Is $f(n) = c \in \Theta\left(n^{log_b a}\right) \in \Theta\left(n^{log_2 1}\right) \in \Theta(n^0)$
$\in \Theta(1)$?

Since $f(n) = c \in \Theta(1)$, the second form of the Master theorem can be used, that is,
$T(n) = \Theta\left(n^{log_b a} \lg n\right) = \Theta(n^0 \lg n) = \Theta(\lg n)$

Hence, the running complexity of Binary Search algorithm is $\Theta(\lg n)$.

Other examples

What is the asymptotic order of the following recurrence relation?

$$T(n) = 4T\left(\frac{n}{2}\right) + n$$

Other examples

What is the asymptotic order of the following recurrence relation?

$$T(n) = 4T\left(\frac{n}{2}\right) + n$$

$$\boxed{T(n) = aT\left(\frac{n}{b}\right) + f(n)}$$

Using Master method, $a = 4, b = 2, and\ f(n) = n$.

Other examples

Test for case 1:

Is $f(n) = n \in O\left(n^{\log_b a - \varepsilon}\right)$ for some $\varepsilon > 0$?

$n \in O\left(n^{\log_2 4 - \varepsilon}\right)$

$n \in O(n^{2-\varepsilon})$

If $\varepsilon = 1$, for example, then $f(n) = n \in O(n^{2-1})$.

Other examples

Test for case 1:
Is $f(n) = n \in O\left(n^{\log_b a - \varepsilon}\right)$ for some $\varepsilon > 0$?

$n \in O\left(n^{\log_2 4 - \varepsilon}\right)$
$n \in O(n^{2-\varepsilon})$

If $\varepsilon = 1$, for example, then $f(n) = n \in O(n^{2-1})$.

Hence, the first form of the Master theorem can be used, that is, $T(n) = \Theta\left(n^{\log_b a}\right) = \Theta\left(n^{\log_2 4}\right) = \Theta(n^2)$.

Other examples

$\therefore$ the asymptotic complexity of the recurrence relation $T(n) = 4T\left(\frac{n}{2}\right) + n$ is $\Theta(n^2)$.

Other examples

What is the asymptotic order of the following recurrence relation?

$$T(n) = 4T\left(\frac{n}{2}\right) + n^2$$

Other examples

What is the asymptotic order of the following recurrence relation?

$$T(n) = 4T\left(\frac{n}{2}\right) + n^2$$

Test using case 1:

Is $f(n) = n^2 \in O\left(n^{\log_b a - \varepsilon}\right)$ for some $\varepsilon > 0$?

$n^2 \in O\left(n^{\log_2 4 - \varepsilon}\right)$

$n^2 \in O(n^{2 - \varepsilon})$

Other examples

If $\varepsilon = 1$, for example, then $f(n) = n^2 \notin O(n^{2-1})$. Hence, the first form of the Master theorem cannot be used.

Other examples

If $\varepsilon = 1$, for example, then $f(n) = n^2 \notin O(n^{2-1})$. Hence, the first form of the Master theorem cannot be used.

Test using case 2:

If $f(n) = n^2 \in \Theta\left(n^{\log_b a}\right)$

$n^2 \in \Theta\left(n^{\log_2 4}\right)$

$n^2 \in \Theta(n^2)$?

Other examples

If $\varepsilon = 1$, for example, then $f(n) = n^2 \notin O(n^{2-1})$. Hence, the first form of the Master theorem cannot be used.

Test using case 2:

If $f(n) = n^2 \in \Theta\left(n^{\log_b a}\right)$

$n^2 \in \Theta\left(n^{\log_2 4}\right)$

$n^2 \in \Theta(n^2)$?

Since $f(n) = n^2 \in \Theta(n^2)$, the second form of the Master theorem can be used, that is,
$T(n) = \Theta\left(n^{\log_b a} \log_2 n\right) = \Theta(n^2 \log_2 n)$.

Other examples

Since $f(n) = n^2 \in \Theta(n^2)$, the second form of the Master theorem can be used, that is,
$T(n) = \Theta\left(n^{\log_b a} \log_2 n\right) = \Theta(n^2 \log_2 n)$.

$\therefore$ the asymptotic complexity of the recurrence relation $T(n) = 4T\left(\frac{n}{2}\right) + n^2$ is $\Theta(n^2 \log_2 n)$.

Other examples

What is the asymptotic order of the following recurrence relation?

$$T(n) = 4T\left(\frac{n}{2}\right) + n^3$$

Other examples

What is the asymptotic order of the following recurrence relation?

$$T(n) = 4T\left(\frac{n}{2}\right) + n^3$$

Test for case 1:

Is $f(n) = n^3 \in O\left(n^{\log_b a - \varepsilon}\right)$
$\in O\left(n^{\log_2 4 - \varepsilon}\right)$
$\in O(n^{2-\varepsilon})$ for some $\varepsilon > 0$?

If $\varepsilon = 1$, for example, then $f(n) = n^3 \notin O(n)$.

Other examples

Hence, the first form of the Master theorem cannot be used.

Other examples

Hence, the first form of the Master theorem cannot be used.

Test for case 2:

Is $f(n) = n^3 \in \Theta\left(n^{\log_b a}\right)$

$n^3 \in \Theta\left(n^{\log_2 4}\right)$

$n^3 \in \Theta(n^2)$?

Other examples

Hence, the first form of the Master theorem cannot be used.

Test for case 2:

Is $f(n) = n^3 \in \Theta\left(n^{\log_b a}\right)$

$n^3 \in \Theta\left(n^{\log_2 4}\right)$

$n^3 \in \Theta(n^2)$?

Since $f(n) = n^3 \notin \Theta(n^2)$, the second form of the Master theorem cannot be used.

Other examples

Test for case 3:

Is $f(n) = n^3 \in \Omega\left(n^{\log_b a+\varepsilon}\right)$

$\in \Omega\left(n^{\log_2 4+\varepsilon}\right)$

$\in \Omega(n^{2+\varepsilon})$ for some $\varepsilon > 0$?

Other examples

Test for case 3:

Is $f(n) = n^3 \in \Omega\left(n^{\log_b a + \varepsilon}\right)$

$\in \Omega\left(n^{\log_2 4 + \varepsilon}\right)$

$\in \Omega(n^{2+\varepsilon})$ for some $\varepsilon > 0$?

If $\varepsilon = 1$, for example, then $f(n) = n^3 \in \Omega(n^{2+1}) \in \Omega(n^3)$.

Other examples

- Next, we prove that $af\left(\frac{n}{b}\right) < cf(n)$ for some constant $c < 1$ and all sufficiently large n.
- That is,

$$af\left(\frac{n}{b}\right) = 4f\left(\frac{n}{2}\right) = 4 \times \frac{n^3}{2^3} = {}^1\!/_2\, n^3$$

- Since ${}^1\!/_2\, n^3 \; < cf(n)$ for some $c = 1/2$ and $n \geq 1$, hence case 3 of master theorem applies.
- Thus $T(n) = \Theta(f(n)) = \Theta(n^3)$

Other examples

∴ the asymptotic complexity of the recurrence relation $T(n) = 4T\left(\frac{n}{2}\right) + n^3$ is $\Theta(n^3)$.