Faculty of Computer and Artificial Intelligence
Sadat University

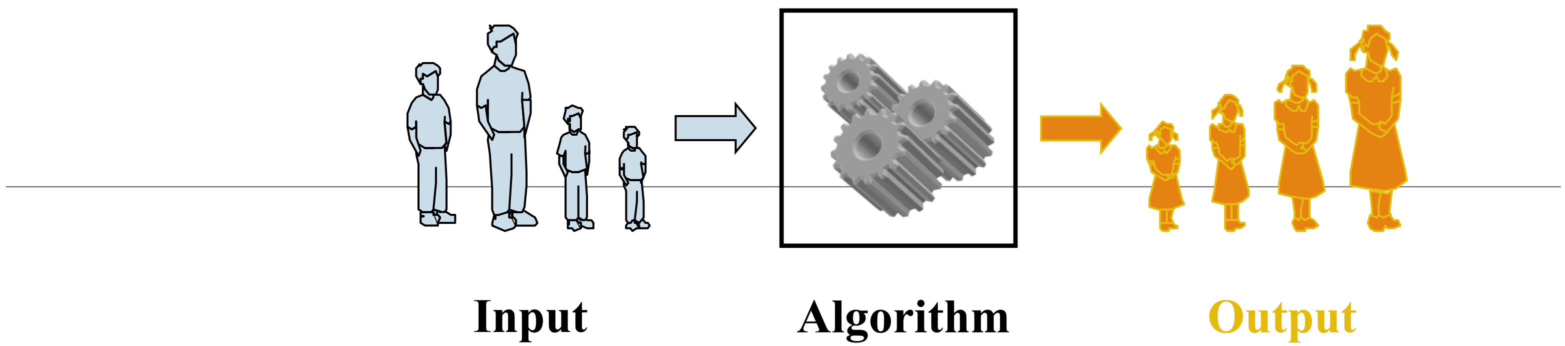
Analysis and Design of Algorithms (CS 302)

Lecture :2

"Introduction (Cont..)+ Recursion "

Dr.Sara A Shehab

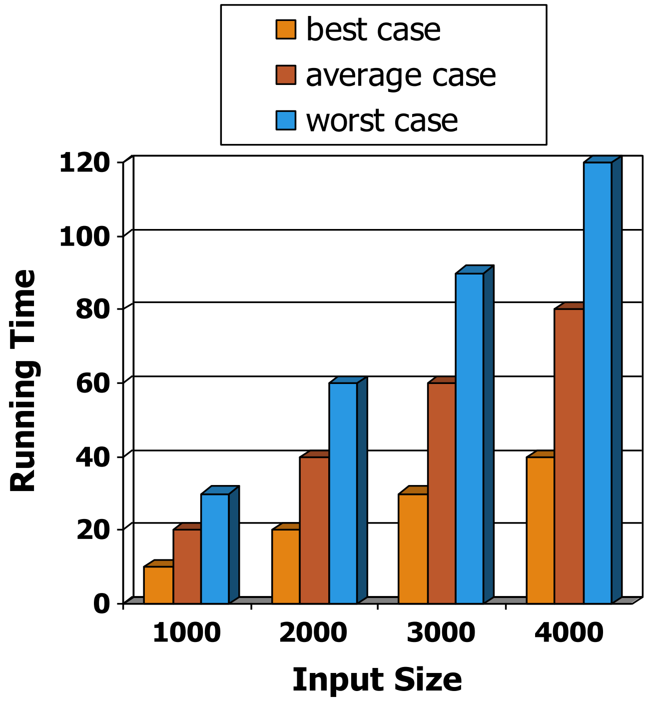
Analysis of Algorithms



An **algorithm** is a step-by-step procedure for solving a problem in a finite amount of time.

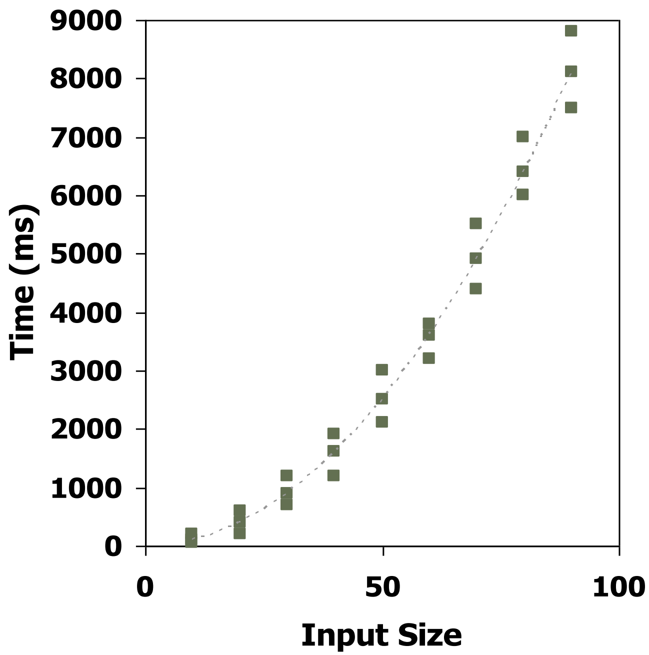
Running Time

- Most algorithms transform input objects into output objects.
- The running time of an algorithm typically grows with the input size.
- Average case time is often difficult to determine.
- We focus on the worst case running time.
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics



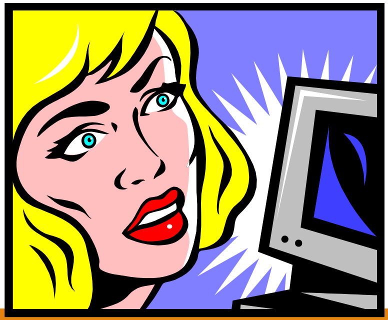
Experimental Studies

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a method like System.currentTimeMillis() to get an accurate measure of the actual running time
- Plot the results



Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used



Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, n.
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Pseudocode

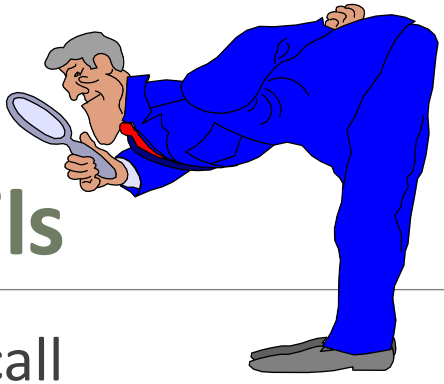
- High-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

Example: find max element of an array

Algorithm *arrayMax*(A, n)
  Input array A of n integers
  Output maximum element of A

  $currentMax \leftarrow A[0]$
  for $i \leftarrow 1$ **to** $n - 1$ **do**
    if $A[i] > currentMax$ **then**
      $currentMax \leftarrow A[i]$
  return $currentMax$

Pseudocode Details



Control flow
- **if** … **then** … [**else** …]
- **while** … **do** …
- **repeat** … **until** …
- **for** … **do** …
- Indentation replaces braces

Method declaration

Algorithm *method* (*arg* [, *arg*…])

Input …

Output …

Method call

var.method (*arg* [, *arg*…])

Return value

return *expression*

Expressions

$\leftarrow$Assignment (like $=$ in Java)

$=$ Equality testing (like $==$ in Java)

n^2 Superscripts and other mathematical formatting allowed

Pseudocode Details

Control Structure	Ending Phrase Word
If then Else	Endif
Case	Endcase
While	Endwhile
For	Endfor

Pseudocode Details

Selection Control Structures

```
Pseudocode: If then Else
If age > 17
  Display a message indicating you can vote.
Else
  Display a message indicating you can't vote.
Endif
```

```
Pseudocode: Case
Case of age
  0 to 17    Display "You can't vote."
  18 to 64   Display "You're in your working years."
  65 +       Display "You should be retired."
Endcase
```

Pseudocode Details

Iteration (Repetition) Control Structures

```
Pseudocode: While
count assigned zero
While count < 5
  Display "I love computers!"
  Increment count
Endwhile
```

```
Pseudocode: For
For x starts at 0, x < 5, increment x
  Display "Are we having fun?"
Endfor
```

Pseudocode Details

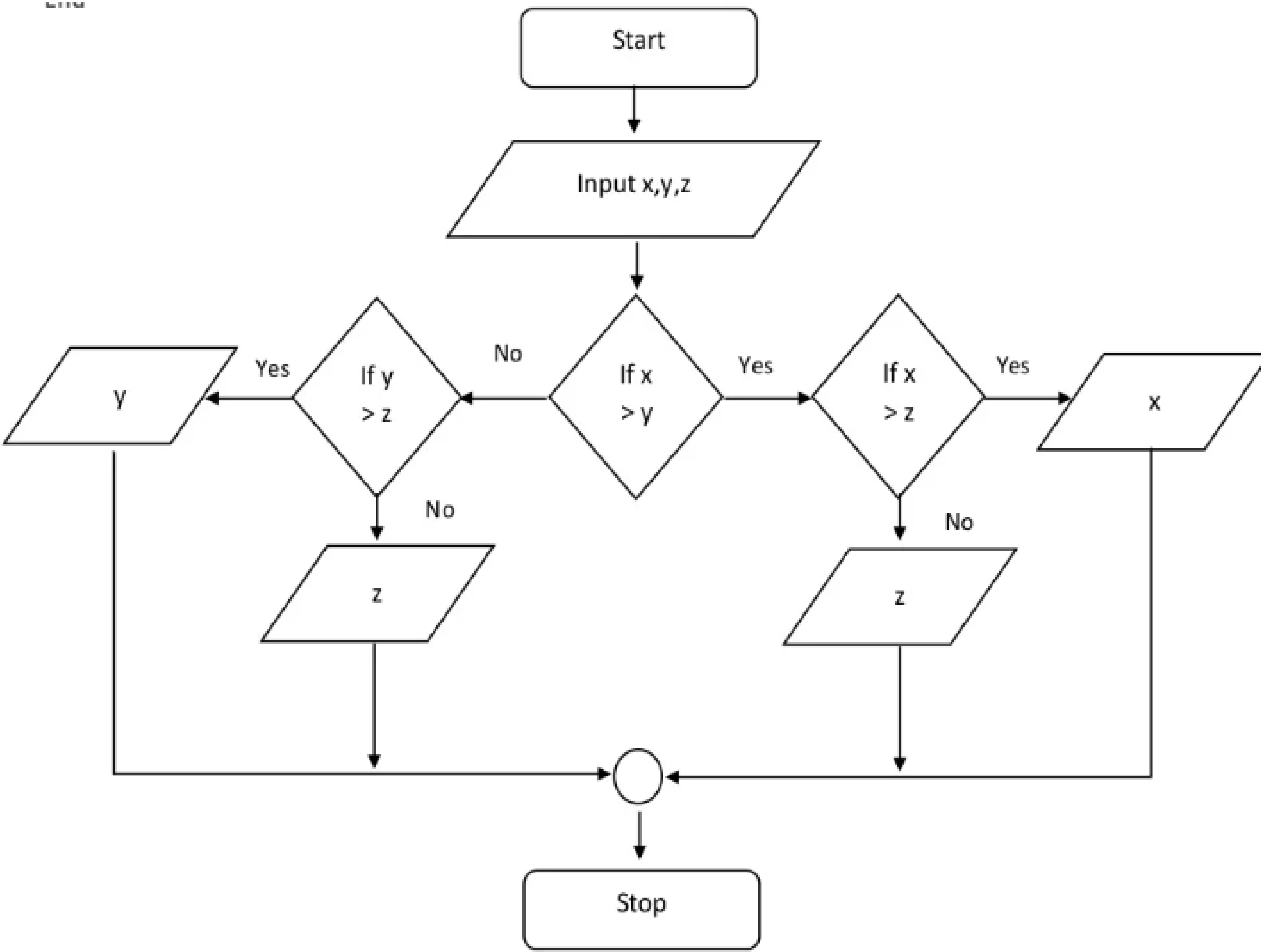
```
Pseudocode: Do While
count assigned five
Do
  Display "Blast off is soon!"
  Decrement count
While count > zero
```

```
Pseudocode: Repeat Until
count assigned five
Repeat
  Display "Blast off is soon!"
  Decrement count
Until count < one
```

Pseudocode

4. To read 3 numbers from user and display maximum.

```
Begin
        Read x,y,z
        If x > y Then
                If x > z Then
                        Display x
                Else:
                        Display z
        Else:
        If y > z Then
                Display y
`       Else:
                Display z
        End if
                End if
        End if
End
```


Start
Input x,y,z
If x
> y
No
Yes
If y
> z
Yes
No
If x
> z
Yes
No
y
z
z
x
Stop

Try this .....

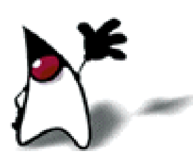
Read 10 numbers from the user and display the maximum number ?

Solution

Types of Algorithms

- Algorithm types we will consider include:
  - Simple recursive algorithms
  - Backtracking algorithms
  - Divide and conquer algorithms
  - Dynamic programming algorithms
  - Greedy algorithms

Recursion



Objectives

- Become familiar with the idea of recursion
- Learn to use recursion as a programming tool
- Become familiar with the binary search algorithm as an example of recursion
- Become familiar with the merge sort algorithm as an example of recursion

Overview

Recursion: a definition in terms of itself.

Recursion in algorithms:

- Natural approach to **some** (not all) problems
- A *recursive algorithm* uses itself to solve one or more smaller identical problems

Recursion in Java:

- Recursive methods implement recursive algorithms
- A *recursive method* includes a call to itself

Key Components of a Recursive Algorithm Design

1. What is a smaller ***identical*** problem(s)?
   - Decomposition
2. How are the answers to smaller problems combined to form the answer to the larger problem?
   - Composition
3. Which is the smallest problem that can be solved easily (without further decomposition)?
   - Base/stopping case

Factorial (N!)

- $N! = (N-1)! * N$ [for $N > 1$]
- $1! = 1$
- 3!

  $= 2! * 3$

  $= (1! * 2) * 3$

  $= 1 * 2 * 3$
- Recursive design:
  - Decomposition: $(N-1)!$
  - Composition: $* N$
  - Base case: 1!

`factorial` Method

```
public static int factorial(int n)
{
  int fact;
  if (n > 1) // recursive case (decomposition)
    fact = factorial(n - 1) * n; // composition
  else // base case
    fact = 1;

  return fact;
}
```

```
public static int factorial(int 3)
{
  int fact;
  if (n > 1)
    fact = factorial(2) * 3;
  else
    fact = 1;
  return fact;
}
```

```
public static int factorial(int 3)
{
  int fact;
  if (n > 1)
    fact = factorial(2) * 3;
  else
    fact = 1;
  return fact;
}
```

```
public static int factorial(int 2)
{
  int fact;
  if (n > 1)
    fact = factorial(1) * 2;
  else
    fact = 1;
  return fact;
}
```

```
public static int factorial(int 3)
{
  int fact;
  if (n > 1)
    fact = factorial(2) * 3;
  else
    fact = 1;
  return fact;
}
```

```
public static int factorial(int 2)
{
  int fact;
  if (n > 1)
    fact = factorial(1) * 2;
  else
    fact = 1;
  return fact;
}
```

```
public static int factorial(int 1)
{
  int fact;
  if (n > 1)
    fact = factorial(n - 1) * n;
  else
    fact = 1;
  return fact;
}
```

```
public static int factorial(int 3)
{
  int fact;
  if (n > 1)
    fact = factorial(2) * 3;
  else
    fact = 1;
  return fact;
}
```

```
public static int factorial(int 2)
{
  int fact;
  if (n > 1)
    fact = factorial(1) * 2;
  else
    fact = 1;
  return fact;
}
```

```
public static int factorial(int 1)
{
  int fact;
  if (n > 1)
    fact = factorial(n - 1) * n;
  else
    fact = 1;
  return 1;
}
```

```
public static int factorial(int 3)
{
  int fact;
  if (n > 1)
    fact = factorial(2) * 3;
  else
    fact = 1;
  return fact;
}
```

```
public static int factorial(int 2)
{
  int fact;
  if (n > 1)
    fact = 1 * 2;
  else
    fact = 1;
  return fact;
}
```

```
public static int factorial(int 1)
{
  int fact;
  if (n > 1)
    fact = factorial(n - 1) * n;
  else
    fact = 1;
  return 1;
}
```

```
public static int factorial(int 3)
{
  int fact;
  if (n > 1)
    fact = factorial(2) * 3;
  else
    fact = 1;
  return fact;
}
```

```
public static int factorial(int 2)
{
  int fact;
  if (n > 1)
    fact = 1 * 2;
  else
    fact = 1;
  return 2;
}
```

```
public static int factorial(int 3)
{
  int fact;
  if (n > 1)
    fact = 2 * 3;
  else
    fact = 1;
  return fact;
}
```

```
public static int factorial(int 2)
{
  int fact;
  if (n > 1)
    fact = 1 * 2;
  else
    fact = 1;
  return 2;
}
```

```
public static int factorial(int 3)
{
  int fact;
  if (n > 1)
    fact = 2 * 3;
  else
    fact = 1;
  return 6;
}
```

Execution Trace (decomposition)

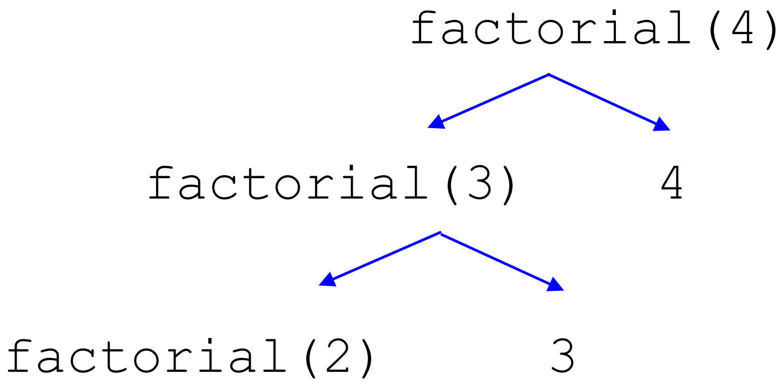
```
public static int factorial(int n)
{
  int fact;
  if (n > 1) // recursive case (decomposition)
    fact = factorial(n – 1) * n; (composition)
  else // base case
    fact = 1;
  return fact;
}
```

factorial(4)

factorial(3)   4

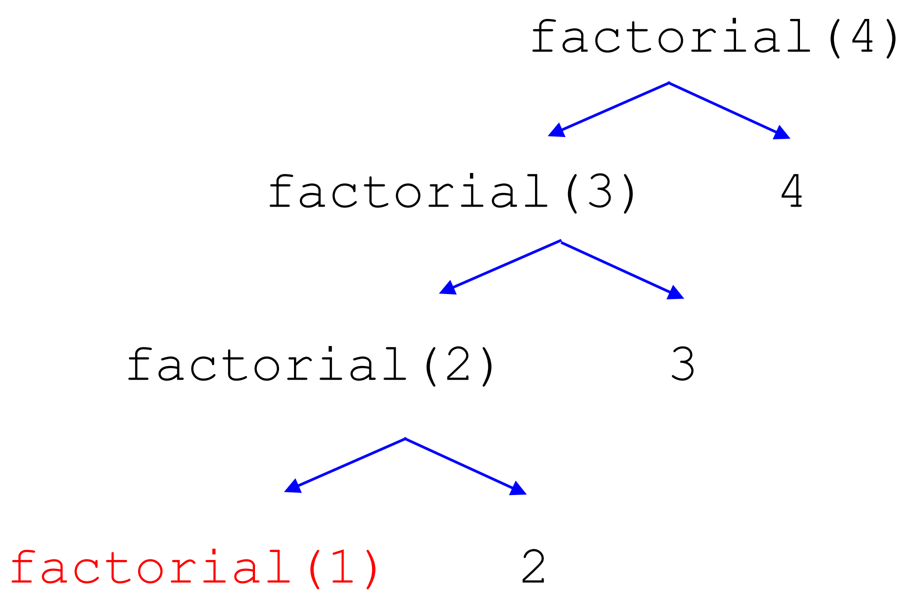
Execution Trace (decomposition)

```
public static int factorial(int n)
{
  int fact;
  if (n > 1) // recursive case (decomposition)
    fact = factorial(n – 1) * n; (composition)
  else // base case
    fact = 1;
  return fact;
}
```



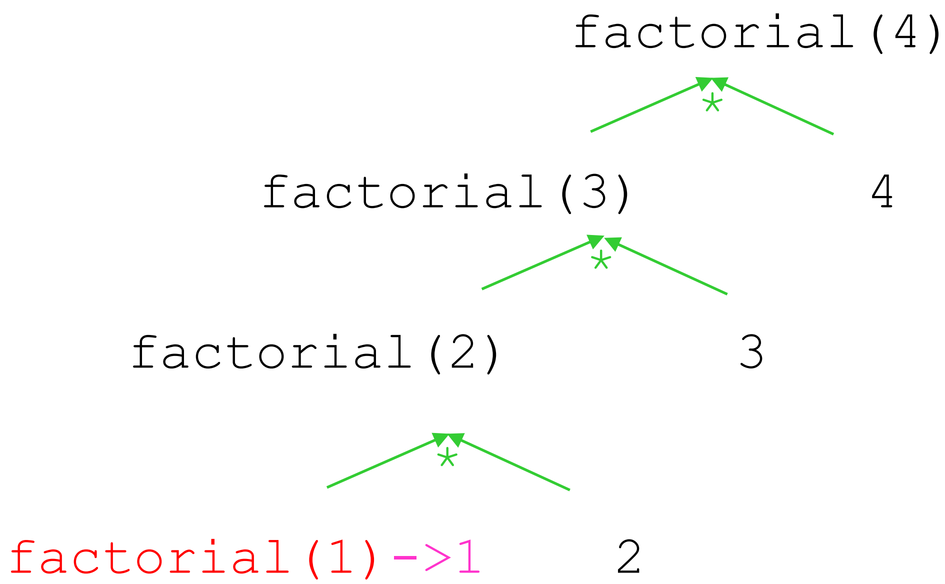
Execution Trace (decomposition)

```
public static int factorial(int n)
{
  int fact;
  if (n > 1) // recursive case (decomposition)
    fact = factorial(n – 1) * n; (composition)
  else // base case
    fact = 1;
  return fact;
}
```



Execution Trace (composition)

```
public static int factorial(int n)
{
  int fact;
  if (n > 1) // recursive case (decomposition)
    fact = factorial(n – 1) * n; (composition)
  else // base case
    fact = 1;
  return fact;
}
```



Execution Trace (composition)

```
public static int factorial(int n)
{
  int fact;
  if (n > 1) // recursive case (decomposition)
    fact = factorial(n – 1) * n; (composition)
  else // base case
    fact = 1;
  return fact;
}
```

```
                    factorial(4)
                         *
          factorial(3)            4
                   *
factorial(2)->2         3
```

Execution Trace (composition)

```
public static int factorial(int n)
{
  int fact;
  if (n > 1) // recursive case (decomposition)
    fact = factorial(n – 1) * n; (composition)
  else // base case
    fact = 1;
  return fact;
}
```

factorial(4)

*

factorial(3)->6   4

Execution Trace (composition)

```
public static int factorial(int n)
{
  int fact;
  if (n > 1) // recursive case (decomposition)
    fact = factorial(n - 1) * n; (composition)
  else // base case
    fact = 1;
  return fact;
}
```

```
factorial(4)->24
```

Improved `factorial` Method

```
public static int factorial(int n)
{
   int fact=1;   // base case value

   if (n > 1)    // recursive case (decomposition)
     fact = factorial(n - 1) * n; // composition
   // else do nothing; base case

   return fact;
}
```

Remember:Key to Successful Recursion

- if-else statement (or some other branching statement)
- Some branches: recursive call
  - "smaller" arguments or solve "smaller" versions of the same task (*decomposition*)
  - Combine the results (*composition*) [if necessary]
- Other branches: no recursive calls
  - stopping cases or base cases

Template

```
… method(…)
{
    if ( … )// base case
    {
    }
    else // decomposition & composition
    {
    }
    return … ; // if not void method
}
```

Template (only one base case)

```
… method(…)
{
   … result = …    ;//base case

   if ( … ) // not base case
   { //decomposition & composition
      result = …
   }

   return result;
}
```

Warning: Infinite Recursion May Cause a Stack Overflow Error

- Infinite Recursion
  - Problem not getting smaller (no/bad decomposition)
  - Base case exists, but not reachable (bad base case and/or decomposition)
  - No base case
- *Stack:* keeps track of recursive calls by JVM (OS)
  - Method begins: add data onto the stack
  - Method ends: remove data from the stack
- Recursion never stops; stack eventually runs out of space
  - **Stack overflow error**

Binary Search Algorithm

- Searching a list for a particular value
  - *sequential* and *binary* are two common algorithms
- *Sequential search* (aka *linear search*):
  - Not very efficient
  - Easy to understand and program
- *Binary search*:
  - more efficient than sequential
  - but *the list must be sorted first!*

Why Is It Called "Binary" Search?

Compare sequential and binary search algorithms:
How many elements are eliminated from the list each time a value is read from the list and it is not the "target" value?

<u>Sequential search:</u> *only one item*

<u>Binary search:</u> *half the list*!

That is why it is called *binary* -

each unsuccessful test for the target value

reduces the remaining search list by 1/2.

Binary Search Method

- `public find(target)` calls `private search(target, first, last)`
- returns the index of the entry if the target value is found or -1 if it is not found
- Compare it to the pseudocode for the "name in the phone book" problem

```
private int search(int target, int first, int last)
 {
    int location = -1; // not found

    if (first <= last) // range is not empty
    {
       int mid = (first + last)/2;

       if (target == a[mid])
          location = mid;
       else if (target < a[mid]) // first half
          location = search(target, first, mid - 1);
       else //(target > a[mid]) second half
          location = search(target, mid + 1, last);
    }

    return location;
 }
```

Where is the composition?

- If no items
  - not found (-1)
- Else if target is in the middle
  - middle location
- Else
  - location found by search(first half) or search(second half)

Binary Search Example

`target` is `33`

The array **`a`** looks like this:

Indices	0	1	2	3	4	5	6	7	8	9
Contents	5	7	9	13	32	33	42	54	56	88

`mid = (0 + 9) / 2` (which is `4`)

`33 > a[mid]` (that is, `33 > a[4]`)

So, if `33` is in the array, then `33` is one of:

					5	6	7	8	9
					33	42	54	56	88

Eliminated half of the remaining elements from consideration because array elements are sorted.

Binary Search Example

`target` is `33`

The array **`a`** looks like this:

	0	1	2	3	4	5	6	7	8	9
Indexes	0	1	2	3	4	5	6	7	8	9
Contents	5	7	9	13	32	33	42	54	56	88

`mid = (5 + 9) / 2` (which is `7`)

`33 < a[mid]` (that is, `33 < a[7]`)

So, if `33` is in the array, then `33` is one of:

					5	6			
					33	42			

Eliminate half of the remaining elements

`mid = (5 + 6) / 2` (which is `5`)

`33 == a[mid]`

So we found `33` at index `5`:

					5				
					33				

Binary vs. Sequential Search

- Binary Search
  - $\log_2 N + 1$ comparisons (worst case)
- Sequential/Linear Search
  - N comparisons (worst case)
- Binary Search is faster ***but***
  - array is assumed to be sorted beforehand
- Faster searching algorithms for "non-sorted arrays"
  - More sophisticated *data structures* than arrays
  - Later courses

Recursive Versus Iterative Methods

All recursive algorithms/methods can be rewritten without recursion.

- *Iterative* methods use loops instead of recursion
- Iterative methods generally run faster and use less memory--less overhead in keeping track of method calls

So When Should You Use Recursion?

- Solutions/algorithms for some problems are inherently recursive
  - iterative implementation could be more complicated
- When efficiency is less important
  - it might make the code easier to understand
- *Bottom line is about:*
  - *Algorithm design*
  - *Tradeoff between readability and efficiency*


The End