CS 162FZ: Introduction to Computer Science II

Lecture 06

Recursion II

Dr. Chun-Yang Zhang

Recursion: Factorial

Factorial(n) - The Iterative Way

Do you recall the iterative method to get the factorial of a number?

The method took a positive integer value n and multiplied all the numbers from 1 to n together.

By definition, the factorial of zero equals 1.

When n = 0, iterativeFactorial(0) = 1

When n = 1, iterativeFactorial(1) = 1

When n = 2, iterativeFactorial(2) = 2 * 1

When n = 3, iterativeFactorial(3) = 3 * 2 * 1 etc…

:

Recursion: Factorial

Factorial(n) - The Iterative Way:

```
public static int iterativeFactorial(int n)
 {
        int product = 1;

        for(int j = 1; j < n; j++)
         {
               product = product * j;
         }
        return product;
 }
```

Recursion: Factorial

Factorial(n) – The Recursive Way:

In order to implement Factorial(n) recursively we need to do two things:

1. Determine the base case(s) – when should the method **not** call itself.
2. Determine the reduction (recursive) step.

Let us first determine the base case(s).
For factorial there are two base cases. By definition 0! and 1! are both equal to 1 and these will be our base cases.

Recursion: Factorial

Factorial(n) – The Recursive Way:

```
public static int recursiveFactorial(int n) {
        // Base Case
        if (n <= 1)
         {
              return 1;
         }
        ....................
}
```

A pattern emerges, namely, **n! = n * (n - 1)!** This pattern will help to code the recursive steps of the factorial program.

Once the base cases [...] reduction (recursive) st[...] examples of factorial.

o 2! = 2 * 1.

o 3! = 3 * 2 * 1 **or** 3 * 2!

o 4! = 4 * 3* 2 * 1 **or** 4 * 3!

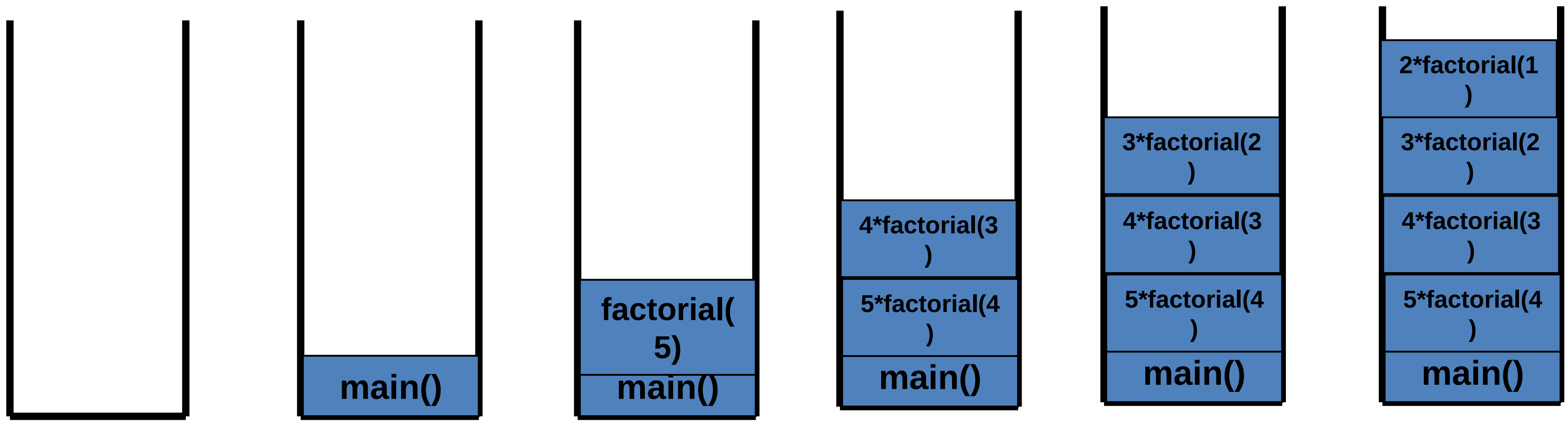
o 5! = 5*4*3*2*1 **or** 5*4!   //See that it is **n*(n-1)! which is (5) * (5-1)**!

Recursion: Factorial

Factorial(n) – The Recursive Way:

```
public static int recursiveFactorial(int n)
 {
// Base Case
      if (n <= 1) // Base Case
       {
            return 1;
       }
      else
      {
            return (n * recursiveFactorial(n – 1));

       }
 }
```

Stacks and Recursive Factorial in Action



Time: 0
Empty Stack

Time 1:
Push: main()

Time 2:
Push: fact(5)

Time 3:
Push: fact(4)

Time 4:
Push: fact(3)

Times 5:
Push fact(2)

!n<=1

So we call factorial(4)

return 5*factorial(5-1)

Inside factorial(4):

print (n); → 4

!n<=1

4*factorial(4-1)

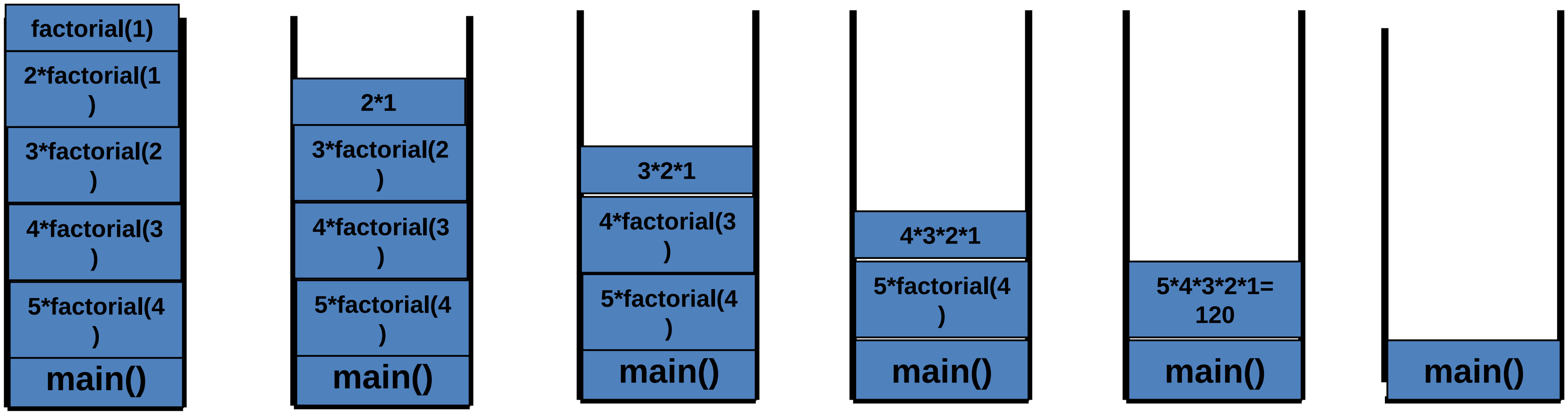
Inside factorial(3):

print (n); → 3

!n<=1

3*factorial(3-1)

Stacks & Recursive Factorial in Action



Time: 5 (cont)

Time 6:
Pop factorial(1)

Time 7:
Pop factorial(2)

Time 8:
Pop factorial(3)

Time 9:
Pop factorial(4)

Time 10:
Pop factorial(5)

Inside factorial(1):

print (n); → 1

n<=1

return 1

Replace factorial(1) with result of 1 and multiply with 2 from factorial(2)

2*factorial(1)=2*1

Because we return 1

Replace factorial(3) with result of 6 and multiply by 4 call from factorial(4)

4*factorial(3)=4*6

Because we return 6

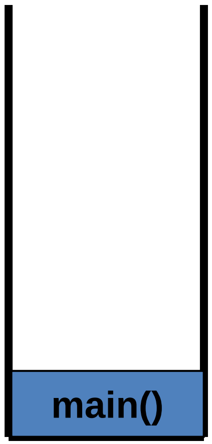
We continue until

we return

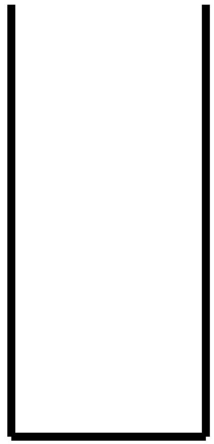
result (120) of factorial(5)

back to main

Stacks and Recursive Factorial in Action



Time 11: Pop Main()



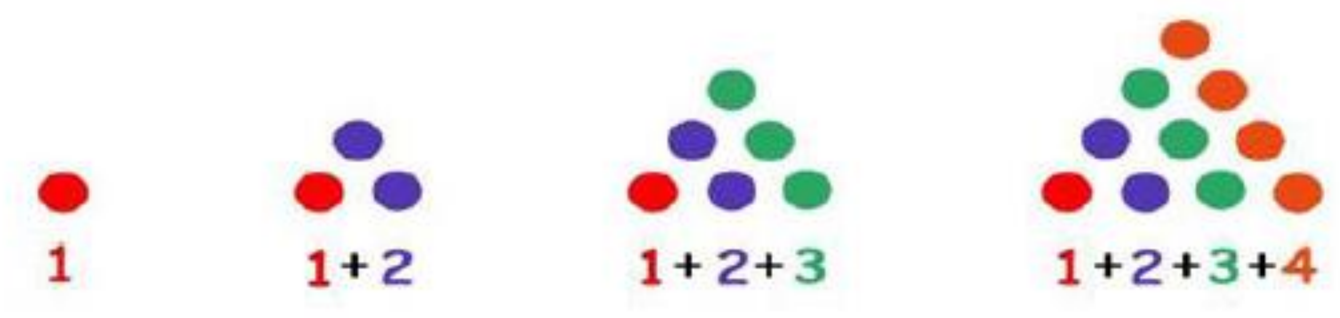
Time 12: Empty Stack – Terminate JVM

Inside main():

print (result); → 120 and exit main()

Recursion: Triangular Numbers

The concept of Triangular Numbers is shown below in the figure. The first triangular number is 1, the second is 3, the 3rd is 6 and so on…



Tri(1)=1
Tri(2) = 1 +2
Tri(3) =1 +2 +3

Recursion: Triangular Numbers

Triangular Number Iteratively

Initially let us begin by looking at an iterative method of computing

```
public static int triangular(int n)
 {
      int triSum=0;
      for(int i=1; i<=n; i++)
       {
            triSum = i + triSum;
       }

      return triSum;
 }
```

Recursion: Triangular Numbers

Triangular Number Iteratively

Initially let us begin by looking at an iterative method of computing

```
public static int triangular(int n)
 {
        int triSum=0;
        for(int i=1; i<=n; i++)
         {
              triSum = i + triSum;
         }

        return triSum;
 }
```

Recursion: Triangular Numbers

Triangular Numbers Recursively

- With Triangular Numbers we are just adding numbers up!
- The 5th (n = 5) triangular number is just 1 + 2 + 3 + 4 + 5.
- If we calculate it using the formula below we can confirm that T(5) = (5*(5+1)) / 2 = (5 * 6) / 2 = 15.

$$T_n = \sum_{k=1}^{n} k = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

Recursion: Triangular Numbers

Triangular Numbers Recursively

- Firstly we need to determine the base case(s) for triangular numbers.

- The first triangular number T(1) = 1. This will be our base case.

Recursion: Triangular Numbers

Triangular Numbers Recursively

•Once the base cases are determined we need to determine the reduction (recursive) step.

•To help with this let us look at a few examples of triangular numbers.

- o T(2) = T(1) + 2.
- o T(3) = T(2) + 3
- o T(4) = T(3) + 4
- •T(5) =T(4) + 5

// T(n)= T(n-1) +n ,

where *n =5* *note that 5-1 OR n-1 = 4 so its T(n-1)*

A pattern emerges, namely, **T(n) = T(n – 1) + n.** This pattern will help to code the recursive steps of the triangular numbers program.

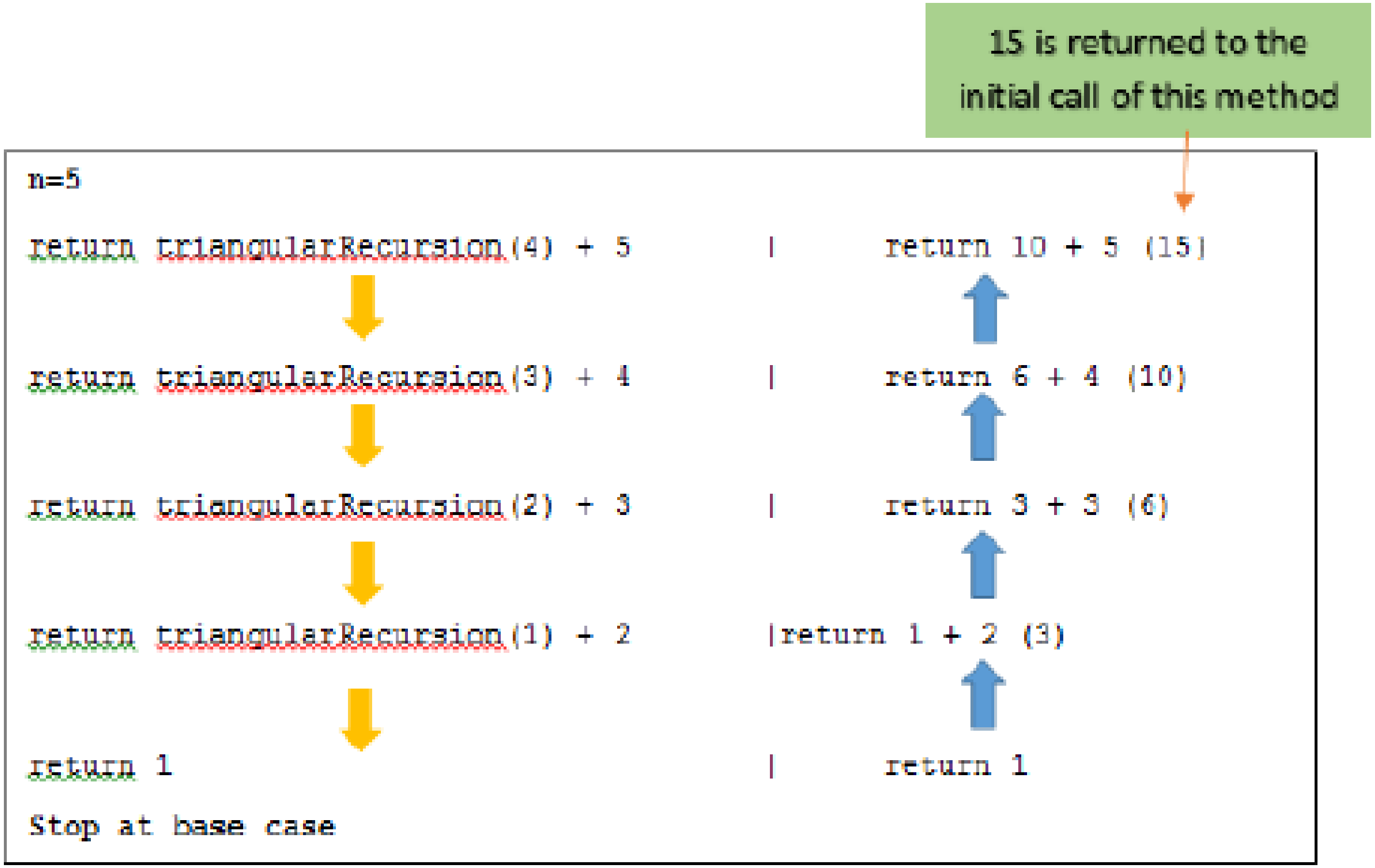
Recursion: Triangular Numbers

Triangular Numbers Recursively

```
public static int triangularRecursion(int n)
 {
      if(n == 1) // Base Case
       {
           return 1;
       }
      else
       {
           return (triangularRecursion(n - 1) + n);
       }
 }
```

Recursion: Triangular Numbers

Triangular Numbers Recursively



Recursion: Fibonacci Numbers

Fibonacci Numbers

- Another example of where we can use a recursive function is calculating the Fibonacci numbers.
- The Fibonacci numbers are 0, 1, 1, 2, 3, 5, 8, 13, 21..
- Fibonacci sequence, **every number after the first two is the sum of the two preceding ones:**
- The following equation is a mathematical description of the Fibonacci Numbers:

$$f_n = \begin{cases} f_0 = 0, & n = 0 \\ f_1 = 1, & n = 1 \\ f_n = f_{n-1} + f_{n-2}, & n \geq 2 \end{cases}$$

Recursion: Fibonacci Numbers

Iterative Fibonacci

If we look at the Fibonacci Numbers from an iterative perspective, the algorithm may look something like the following:

- To calculate any Fibonacci Number “n” we need to know the Fibonacci Number “n – 1” and the Fibonacci Number “n – 2”.
- For our iterative version we start at the 1st number (n = 0).
- So if we are looking for the 5th Fibonacci Number – we start off by looking at **0th and 1st** Fibonacci numbers and **adding them together to get the next.**
- We then look at the 1st and 2nd Fibonacci numbers and then 2nd and 3rd etc....

Recursion: Fibonacci Numbers

Iterative Fibonacci

```
public static int FibIter(int n)
{
  int prev1 = 0, prev2 = 1;
  int savePrev1 = 0;
  for(int i=0; i<n; i++)
   {
        savePrev1=prev1;
        prev1=prev2;
        prev2=savePrev1+prev2;
   }
  return prev1;
}
```

n=0
savePrev1=prev1=0
prev1=prev2=1
prev2=savePrev1+prev2=0+1=1
n=1
savePrev1=prev1=1
prev1=prev2=1
prev2=savePrev1+prev2=1+1=2
n=2
savePrev1=prev1=1
prev1=prev2=2
prev2=savePrev1+prev2=1+2=3
n=3
savePrev1=prev1=2
prev1=prev2=3
prev2=savePrev1+prev2=2+3=5

n=3
savePrev1=prev1=3
prev1=prev2=5
prev2=savePrev1+prev2=3+5=8

Recursion: Fibonacci Numbers

Recursive Fibonacci

The **base cases** in Fibonacci Numbers are the first (0th) and second numbers in the series – 0 and 1. The Fibonacci sequences begins as follows:

0,1,1,2,3,5,8,13,21,34,55......

- To calculate the 3rd Fibonacci Number we must add the 2nd and the 1st Fibonacci numbers together.
- To calculate the 4th Fibonacci Number we must add the 3rd and the 2nd Fibonacci numbers together.

From this, the general formula can be specified as:

$$f_n = \begin{cases} f_0 = 0, & n = 0 \\ f_1 = 1, & n = 1 \\ f_n = f_{n-1} + f_{n-2}, & n \geq 2 \end{cases}$$

Base Case (Termination)

Recursive step – notice how f_n calls itself again twice.

Recursion: Fibonacci Numbers

Recursive Fibonacci

From this, the general formula can be specified as:

$$f_n = \begin{cases} f_0 = 0, & n = 0 \\ f_1 = 1, & n = 1 \\ f_n = f_{n-1} + f_{n-2}, & n \geq 2 \end{cases}$$

Base Case (Termination)

Recursive step – notice how f_n calls itself again twice.

Remember to get a Fibonacci Number *n* we must add the 2nd and the 1st numbers together. So If n= 2 fib(2) = fib(n-1) + fib(n-2) = fib(2)=fib(1)+fib(0) = 1+0 = 1

Recursion: Fibonacci Numbers

Recursive Fibonacci

Java code for recursive fibonacci is as follows:

```
public static int fib(int n)
{
        if (n == 0)
        {
                return 0;
        }
        else if (n == 1)
        {
                return 1;
        }
        else
        {
                return fib(n - 1) + fib(n - 2);
        }
}
```

$$f_n = \begin{cases} f_0 = 0, & n = 0 \\ f_1 = 1, & n = 1 \\ f_n = f_{n-1} + f_{n-2}, & n \geq 2 \end{cases}$$

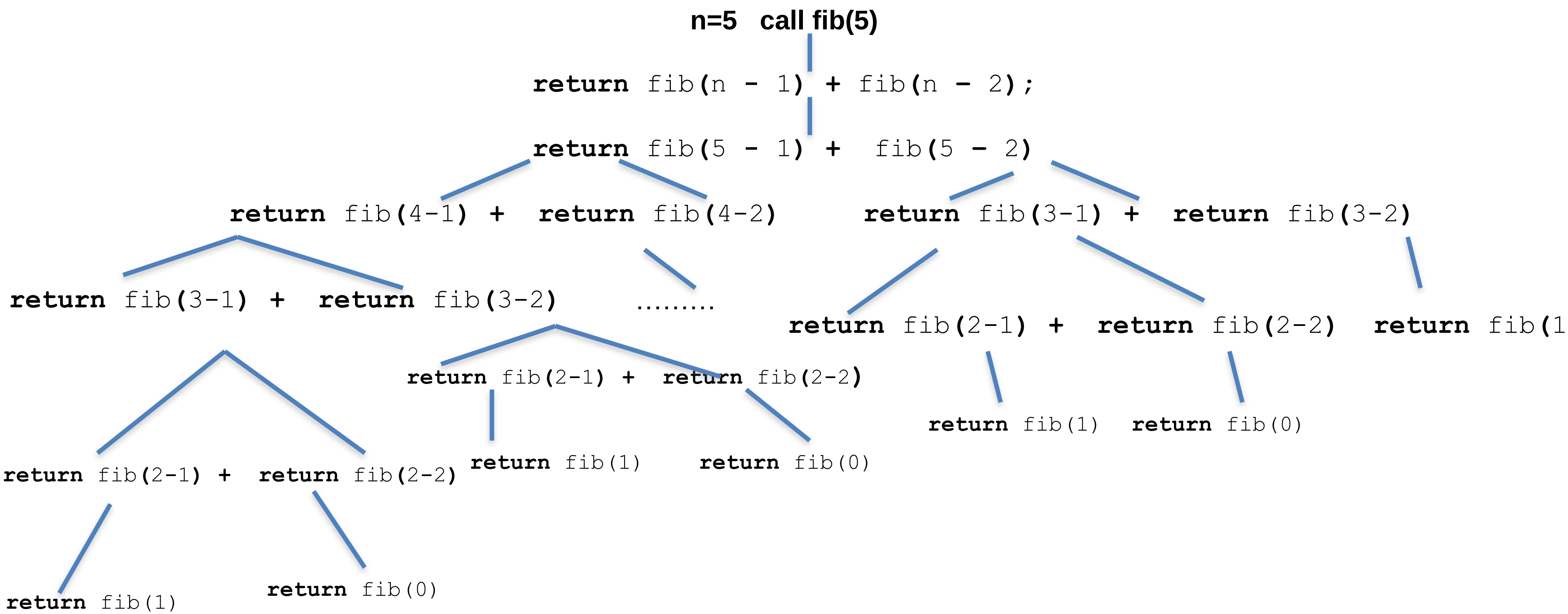
Note on Iterative and Recursive Fibonacci Code

- Both solutions calculate the same solution to the same problem. However, both approaches are VERY different.

Recursion: Fibonacci Numbers

Recursive Fibonacci

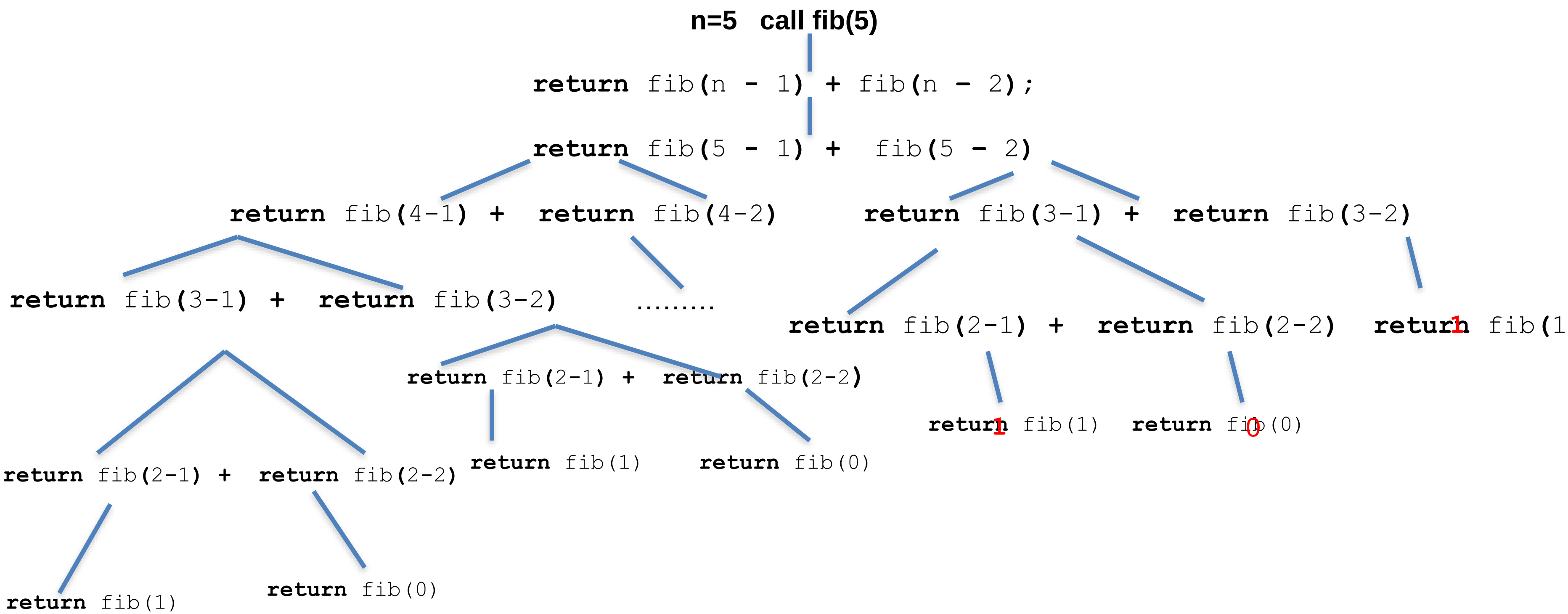
Execution of recursive fibonacci is as follows:



Recursion: Fibonacci Numbers

Recursive Fibonacci

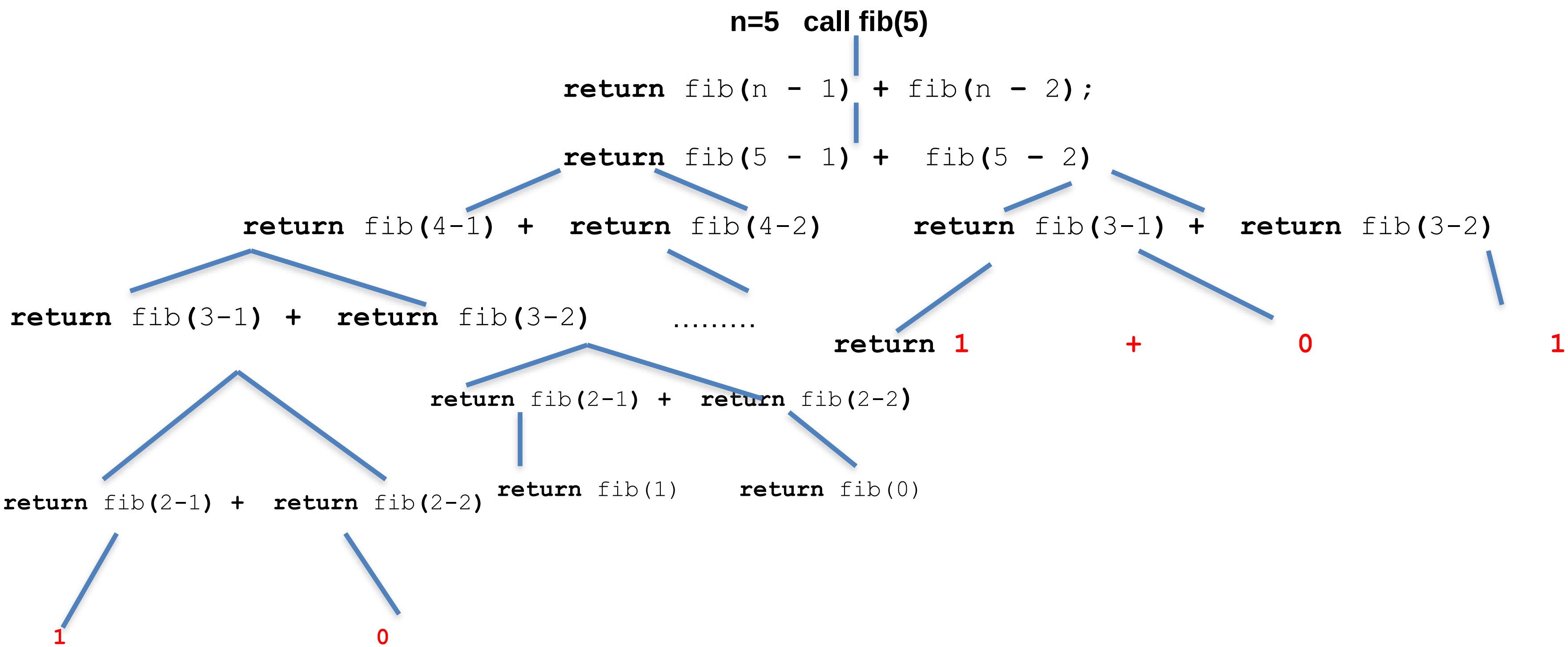
Execution of recursive fibonacci is as follows:



Recursion: Fibonacci Numbers

Recursive Fibonacci

Execution of recursive fibonacci is as follows:



Recursion: Fibonacci Numbers

Recursive Fibonacci

Execution of recursive fibonacci is as follows:

n=5 call fib(5)

```
return fib(n - 1) + fib(n - 2);

return fib(5 - 1) +  fib(5 - 2)

return fib(4-1) +  return fib(4-2)      return fib(3-1) +  return fib(3-2)

return fib(3-1) +  return fib(3-2)     .........     return 1     +     0     1

return fib(2-1) +  return fib(2-2)

1     0

return 1     +     0
```

Recursion: Fibonacci Numbers

Recursive Fibonacci

Execution of recursive fibonacci is as follows:

n=5 call fib(5)

```
return fib(n - 1) + fib(n - 2);

return fib(5 - 1) +  fib(5 - 2)

return fib(4-1) +  return fib(4-2)      return 1    +    1

return 1              +  return fib(3-2)      1

                              1
```

Recursion: Fibonacci Numbers

Recursive Fibonacci

Execution of recursive fibonacci is as follows:

n=5 call fib(5)

```
return fib(n - 1) + fib(n - 2);
              3   +   2
```

Recursion: Fibonacci Numbers

Recursive Fibonacci

Execution of recursive fibonacci is as follows:

n=5 call fib(5)

`return` 5

Recursion: Functional Definitions

Recursive Functional Definitions

- Very often we are given problems which are formulated as recursive function definitions – just like we have seen with the Fibonacci sequence.
- They can represent some other pattern or sequence in a scientific situation, scientific problem or environmental situation.
- The definition of the problem is usually written mathematically.
- One such example is:

$$a_n = \begin{cases} a_1 = 4, & n = 1 \\ a_n = 5a_{n-1} + 10, & n \geq 2 \end{cases}$$

Recursion: Functional Definitions

Recursive Functional Definitions

To calculate the values of a(n) where n=4

a1 = 4		
a2 = 5a1 + 10	= 5(4) + 10	= 30
a3 = 5a2 + 10	=5(30) + 10	= 160
a4 = 5a3 + 10	=5(160) + 10	= 810

- Can you see the recursion?
- Can you see the function call to itself?
- Let us examine how an is calculated.
- We know that the first value is a1=4.

Recursion: Functional Definitions

Recursive Functional Definitions

Base Case (Termination Condition)

Recursive Call

$$a_n = \begin{cases} a_1 = 4, & n = 1 \\ a_n = 5a_{n-1} + 10, & n \geq 2 \end{cases}$$

The first values for this sequence are: 4, 30, 160, 810, 4060, ……. A Java recursive method to complete this action is:

```
public static int aFunc(int n)
 {
       if (n == 1)
        {
             return 4;
        }
       else
        {
             return (5 * aFunc(n-1) + 10);
        }
 }
```

Recursion: Functional Definitions

Recursive Functional Definitions

Let us see in more detail what is happening in this method. Let us take a value of n = 5;

n=5

return 5a(4) + 10	n=5	5(810)+10 = 4060
return 5a(3) + 10	n=4	5(160)+10 = 810
return 5a(2) + 10	n=3	5(30)+10 = 160
return 5a(1) + 10	n=2	5(4)+10 = 30
return 4	n=1	

Recursion: Functional Definitions

Writing the Recursive Function Iteratively

Let us look at writing the previous function in an iterative manner.

$$a_n = \begin{cases} a_1 = 4, & n = 1 \\ a_n = 5a_{n-1} + 10, & n \geq 2 \end{cases}$$

Figure 8 - Recursive Function

A similar approach is needed as to that used in the. A for loop can be used here again. The program needs to keep track of the current value and the previous values. One version of an iterative solution is:

```
public static int A (int n)
{
        int baseCase = 4;

        int current = baseCase;
        int runningTotal = current;
        // Start the loop at i = 2
        // (this is the first recursive case)
        for (int i = 2; i <= n; i++)
        {
                runningTotal = 5*current + 10;
                current = runningTotal;
        }
        return runningTotal;
}
```

Recursion: Functional Definitions

Writing the Recursive Function Iteratively

Let us look at writing the previous function in an iterative manner.

Stepping through with n = 4 we can see what happens in this iterative method:

```
n=4

baseCase=4;
current=4;

runningTotal=4;

i=2
      runningTotal = 5(current)+10;  //runningTotal=30
      current = runningTotal; // current = 30
i=3
      runningTotal = 5(current)+10  // runningTotal = 160
      current = runningTotal; // current = 160
i=4
      runningTotal = 5(current)+10 = 30  //runningTotal = 810
      current = runningTotal; // current=810
```

Recursion with Arrays

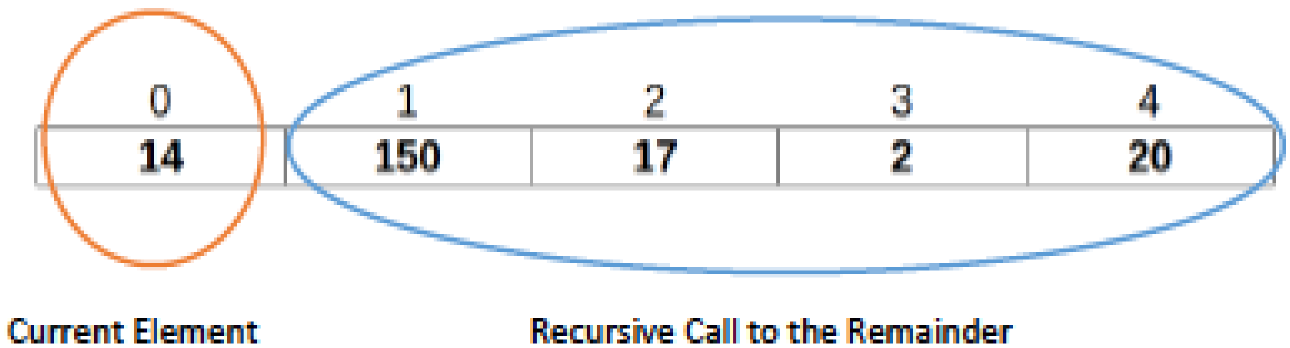
- Recursion with Arrays
- Recursion can also be used to find the maximum and minimum values stored in an array. Let us look at an example of finding the largest value of an array. Consider the following array:

0	1	2	3	4
14	**150**	**17**	**2**	**20**

Recursion with Arrays

- In order to consider solving this recursively you have to consider the array in the same way as the string of characters in the palindrome recursive program.
- First the base case(s) and recursive step need to be determined. Approach the problem of looking for the maximum value by considering the current element and then look at making a recursive call to the 'remainder' of the array.
- Assume the current element is element 0:

Recursion with Arrays



- What might the base case be?
- We will assume that we are going to work through the array from the current element until the end.
- Our base case will be when we are looking at the last element in the array – we know at this point that we will have traversed all the elements in the array.

Recursion with Arrays

- What will our recursive step be?
- We will compare each element with the current stored largest element. If the element we are currently looking at is larger than the current largest stored element we need to update this value to the now largest value.
- We will continue this until the end of the array.

Recursion with Arrays

- A sample recursive solution to find the largest element in an array is:

```
public static int maxArray(int [] array, int start)
{
        if(start==array.length-1)
         {
                //base case - single element array
                return array[start];
         }
        else
         {
                //compare current element and remainder of array
               return (Math.max(array[start], maxArray(array, start+1)));
         }
}
```

Current Element   Remainder of Array

Recursion with Arrays

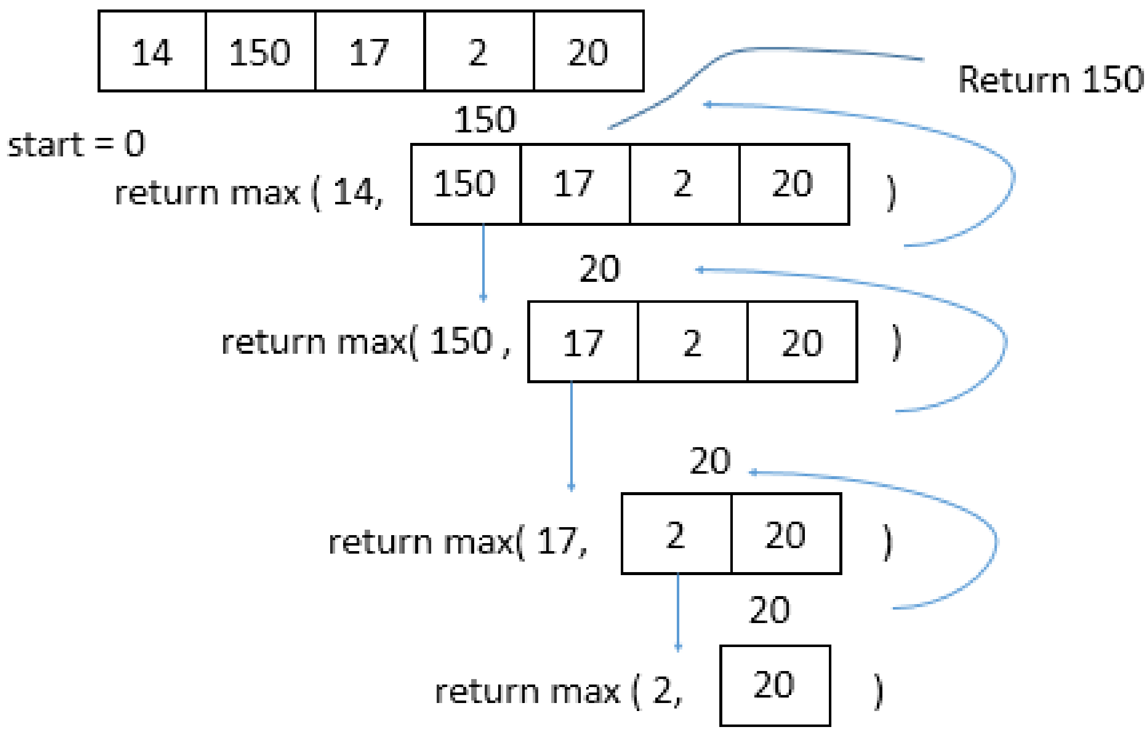
- the example we used `Math.max` to get the maximum of two numbers.
- We could just as easily use an `if else` statement. The `java.lang.Math.max(int a, int b)` returns the greater of two int values. Another example of using `Math.max` is: :

```
import java.lang.*;

public class MathDemo
 {
   public static void main(String[] args)
    {
      // get two integer numbers
      int x = 60984;
      int y = 1000;

      // print the larger number using Math.max(<...>)
      System.out.println(Math.max(x, y));
    }
}
```

Recursion with Arrays

- Let us look at how our recursive function works.

0	1	2	3	4
14	**150**	**17**	**2**	**20**



Recursion with Arrays

We can also code this solution in an iterative manner as follows. A `for loop` can be used to iterate (or visit) every element in the array in order.

```
public static int maxArray(int [] array)
{
        int max=Integer.MIN_VALUE;

        for (int i=0;i<array.length;i++){
                if(array[i]>max){
                        max=array[i];
                }
        }

        return max;
}
```

Recursion with Arrays

In this example we are using `int max=Integer.MIN_VALUE`.

- Using Integer.MIN_VALUE allows the integer variable max to contain the smallest integer number usable in Java - which is -2,147,483,648 or -2^{31}.
- If the variable was initialised without assigning it an integer number, it would be set to the default 0 and there may be numbers smaller than that in the array.
- Using Integer.MIN_VALUE means there cannot be smaller integer numbers but only numbers equal to -2^{31} or greater.
- As well as Integer.MIN_VALUE we also have Integer.MAX_VALUE = $(2^{31}-1)$ or 2,147,483,647.

Recursion with Arrays

- Arrays lend themselves very well to recursion and iteration.
- An array as a data structure can naturally be reduced into smaller units – as we saw when we took the first element and then considered the remainder of the array.

Pros and Cons of Recursion

Cons

•Recursion repeatedly invokes the method which incurs a cost in terms of memory and processing time.
•Each recursive call causes another copy of the method (and all its variables) to be created.
•This copying of methods consumes considerable memory space.

Pros

•Sometimes it's easier to find a recursive solution if we just make a slight change to the original problem.
•Occasionally a recursive solution runs much slower than its iterative counterpart.
•However for the most part it is just marginally slower.
•In many cases it is easier to understand and code up the recursive solution than the iterative one.