ACTIVITY **3.1.7**

Sort Algorithms

INTRODUCTION

An *algorithm* is a sequence of instructions that accomplish a task. The word “algorithm” was derived from the Latin form of a 9th-century Persian mathematician’s name, Al-Khwarizmi. In today’s world, the word algorithm most often relates to a repeatable procedure that solves a problem using a computer.

Algorithms are a central concept to computer science. Some problems are so important, common, or both, that multiple algorithms have been created to solve them. Algorithms that solve the same problem can be compared in terms of speed, complexity, and use of computer resources (like memory and CPU time).

Two of the most common algorithms used in Computer Science are sorting and searching. Many algorithms have been developed to sort data; the algorithms perform different types of sorts and have different performance results. In this activity you will explore three common sort algorithms. In the next activity, you will explore two common search algorithms.

Materials

- Playing cards
- Computer with Android™ Studio
- Android™ tablet and USB cable, or a device emulator
- Free Backendless account per student

RESOURCES

Sorting Algorithms Supplemental Materials
Resources available online

Lesson 3.1 Reference Card for Backendless
Resources available online

Procedure

Imagine you have a collection of coins from various countries all over the world. If you need to find a particular coin from a country, how would you go about finding it? What would you do to make this task easier for you?

To search for an item efficiently, it is best to have the list of items sorted first. Multiple sorting algorithms are available. You will be introduced to three in this activity: **selection sort**, **insertion sort**, and **merge sort**.

selection sort

A sort algorithm that repeatedly scans for the smallest item in the list and swaps it with the element at the current index. The index is then incremented, and the process repeats until the last two elements are sorted.

insertion sort

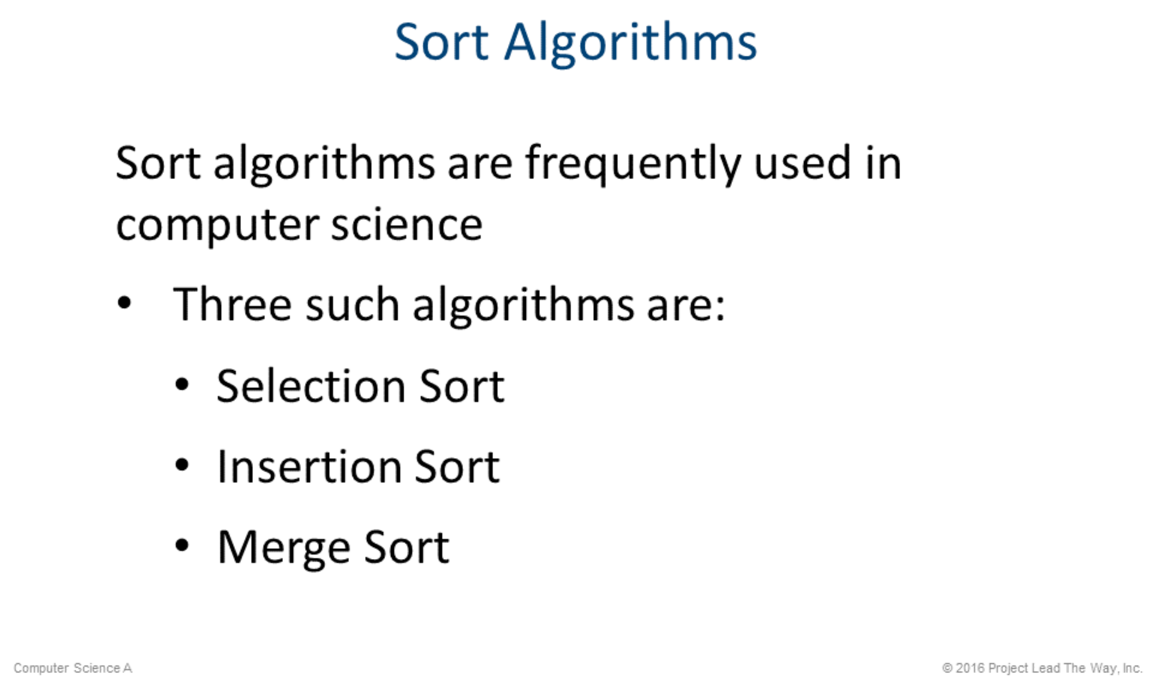
A sort algorithm that repeatedly compares and inserts items into a subset at the front of the list that is considered already sorted.

merge sort

A "divide-and-conquer" sort algorithm that recursively calls itself on the left sub-list and the right sub-list, until the sub-lists contain only one item. Then it starts to merge the sub-lists repeatedly into a sorted list, until all list items are merged and sorted into one big list.

Part I: Selection Sort

1. For an explanation of selection sort, view the slideshow.



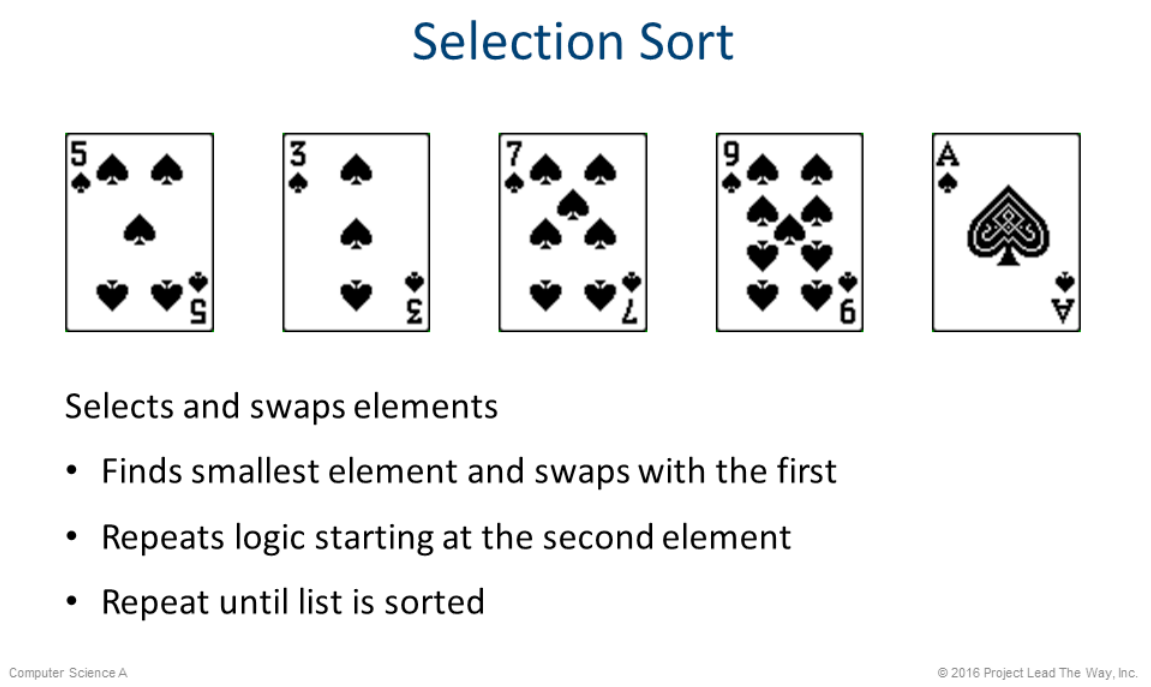
Sort Algorithms

Sort algorithms are frequently used in computer science

- Three such algorithms are:
  - Selection Sort
  - Insertion Sort
  - Merge Sort

Computer Science A

© 2016 Project Lead The Way, Inc.



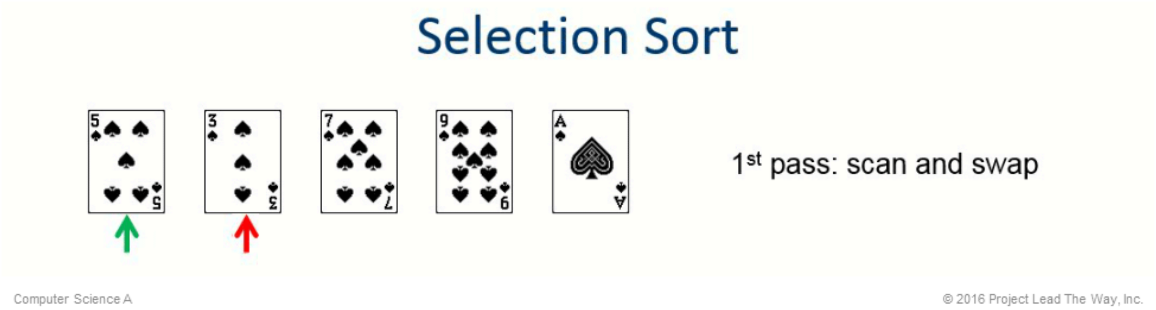
Selection Sort

Selects and swaps elements

- Finds smallest element and swaps with the first
- Repeats logic starting at the second element
- Repeat until list is sorted

Computer Science A

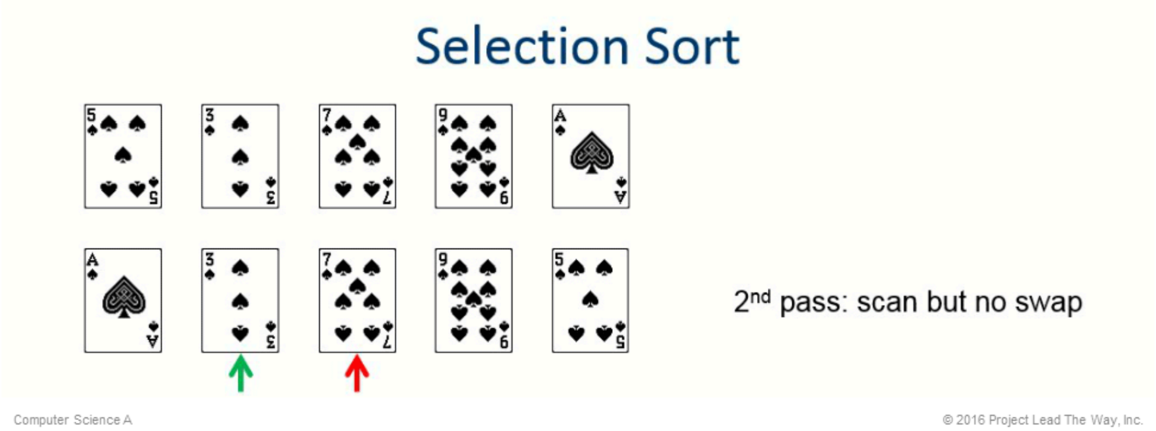
- The selection sort finds the smallest element in the list and swaps it with the first element.
- Then it repeats the same logic by searching through the list starting at the second element and swapping the smallest item with the second element.
- This repeats until the last two elements are sorted.



Selection Sort

1st pass: scan and swap

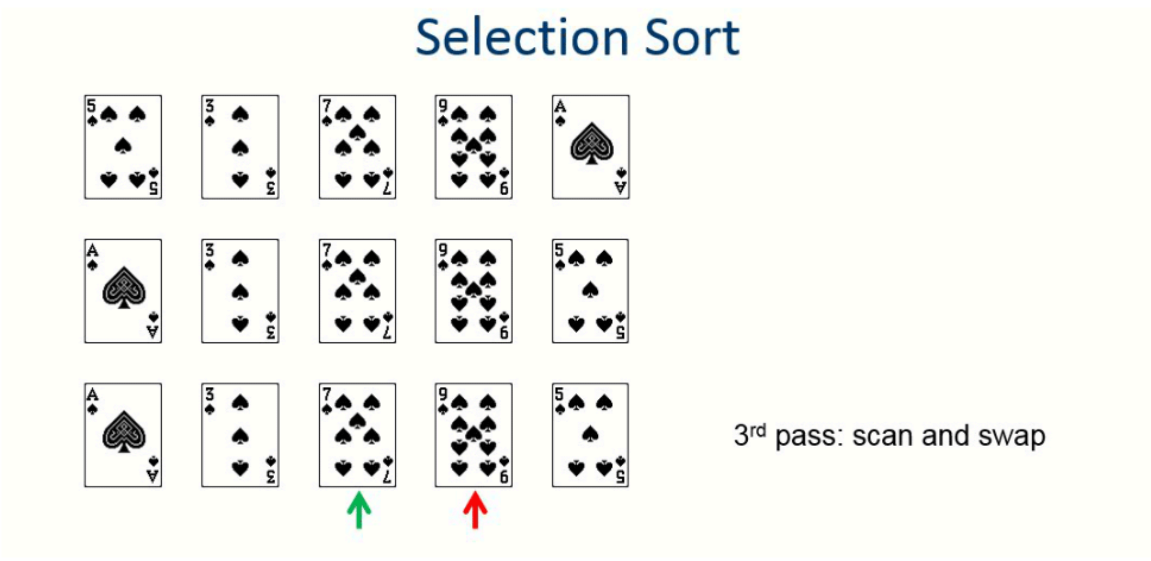
Computer Science A



Selection Sort

2nd pass: scan but no swap

Computer Science A

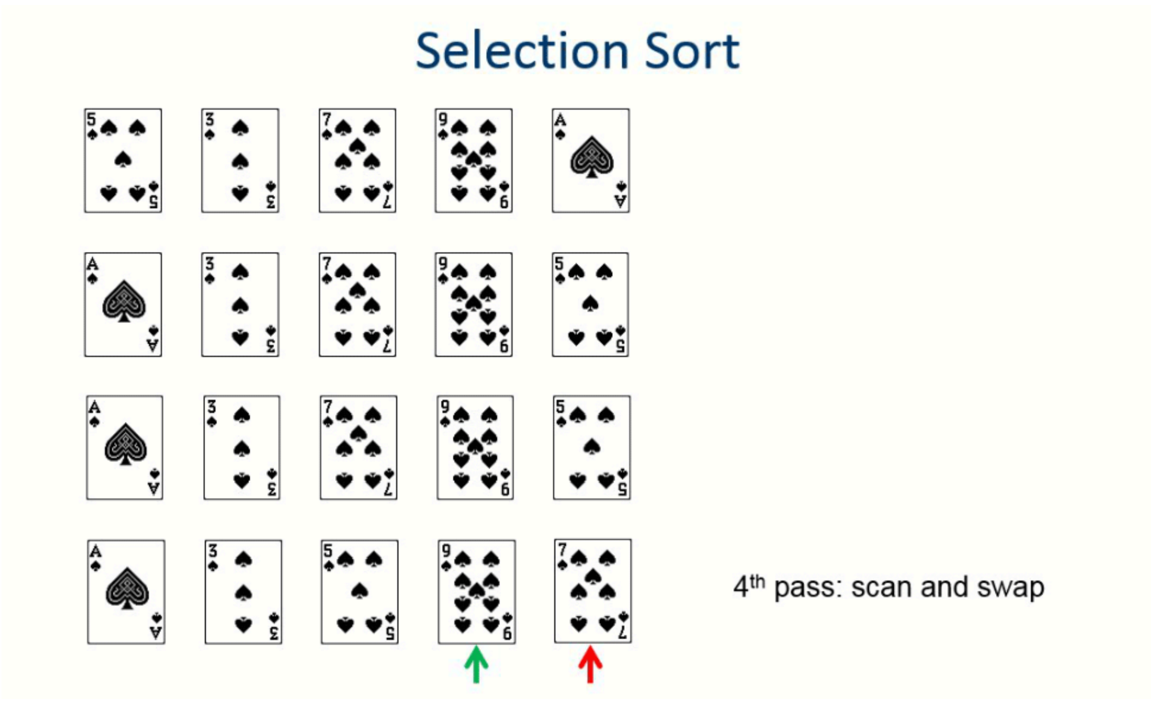


Selection Sort

3rd pass: scan and swap

Computer Science A

© 2016 Project Lead The Way, Inc.

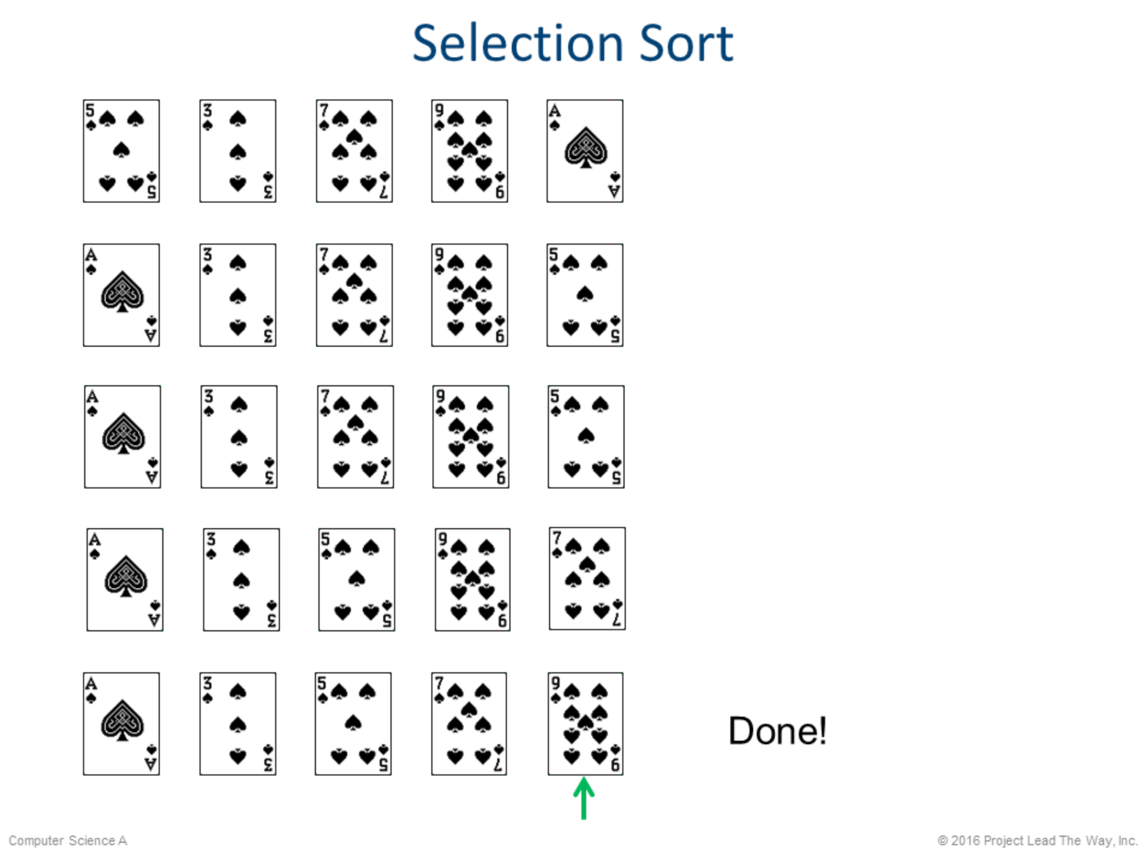


Selection Sort

4th pass: scan and swap

Computer Science A

© 2016 Project Lead The Way, Inc.

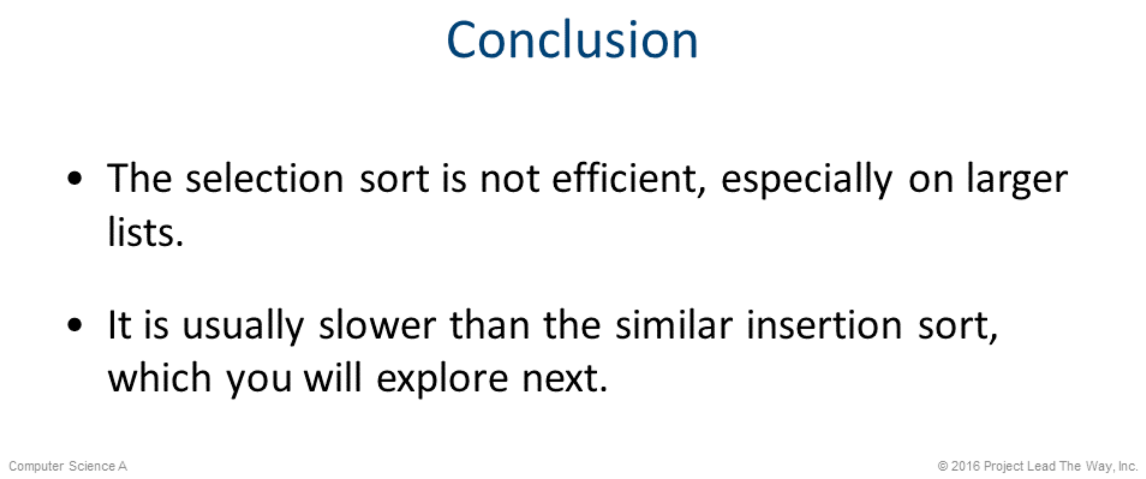


Selection Sort

Done!

Computer Science A

© 2016 Project Lead The Way, Inc.



Conclusion

- The selection sort is not efficient, especially on larger lists.
- It is usually slower than the similar insertion sort, which you will explore next.

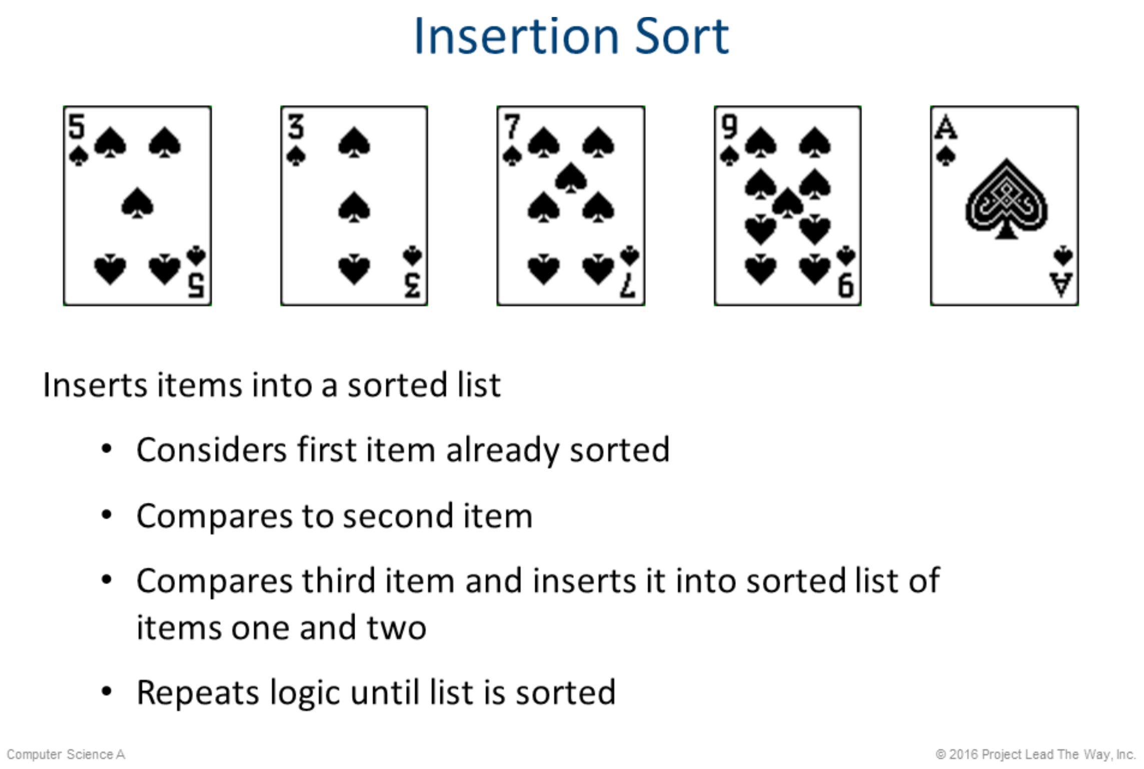
Computer Science A

© 2016 Project Lead The Way, Inc.

2 Work in a group to use the selection sort to sort through five playing cards, supplied by your teacher. In your notebook, draw the list of cards to illustrate how the sequence changes after each pass of the sort algorithm.

Part II: Insertion Sort

3 For an explanation of insertion, view the slideshow.



Insertion Sort

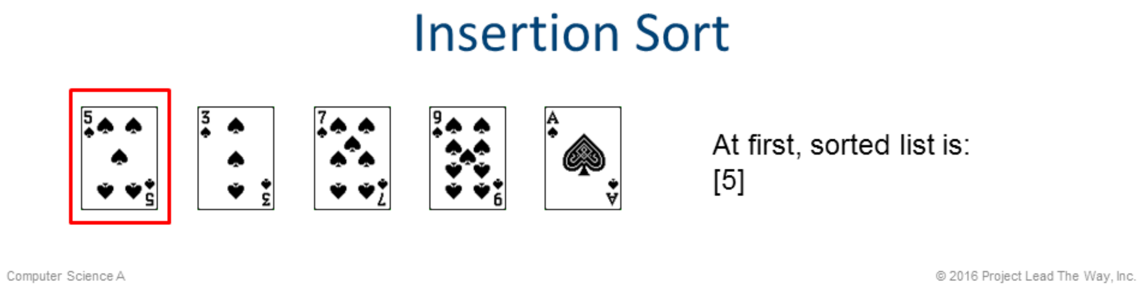
Inserts items into a sorted list

- Considers first item already sorted
- Compares to second item
- Compares third item and inserts it into sorted list of items one and two
- Repeats logic until list is sorted

Computer Science A © 2016 Project Lead The Way, Inc.

The insertion sort repeatedly compares and inserts items into a subset of the list that is considered already sorted.

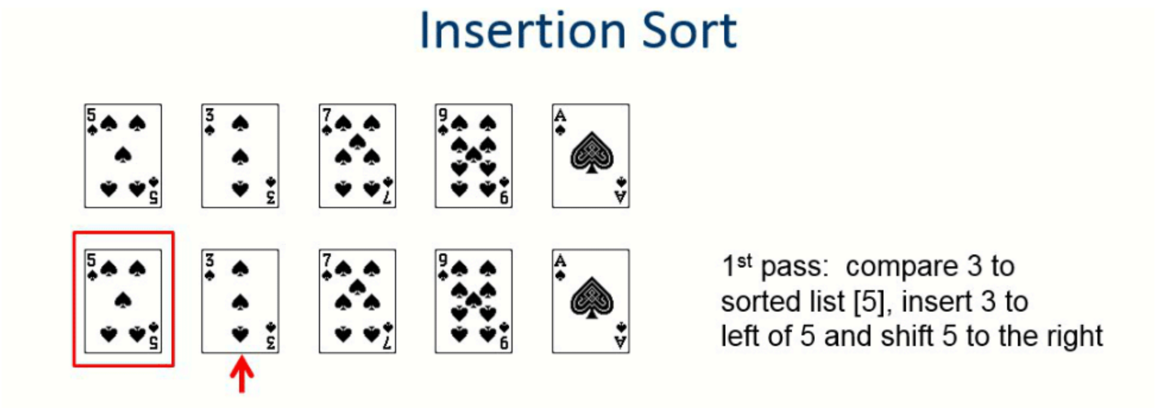
- Begins by considering the first item in the list to be the already “sorted list”.
- Compares the next item in the list to the first item and swaps them if needed. Now the first two items are considered sorted.
- Compares the third item to the items in the ‘sorted list’ and inserts it in the appropriate position. The first three items are considered sorted.
- Repeats this logic until the entire list is sorted.



Insertion Sort

At first, sorted list is:
[5]

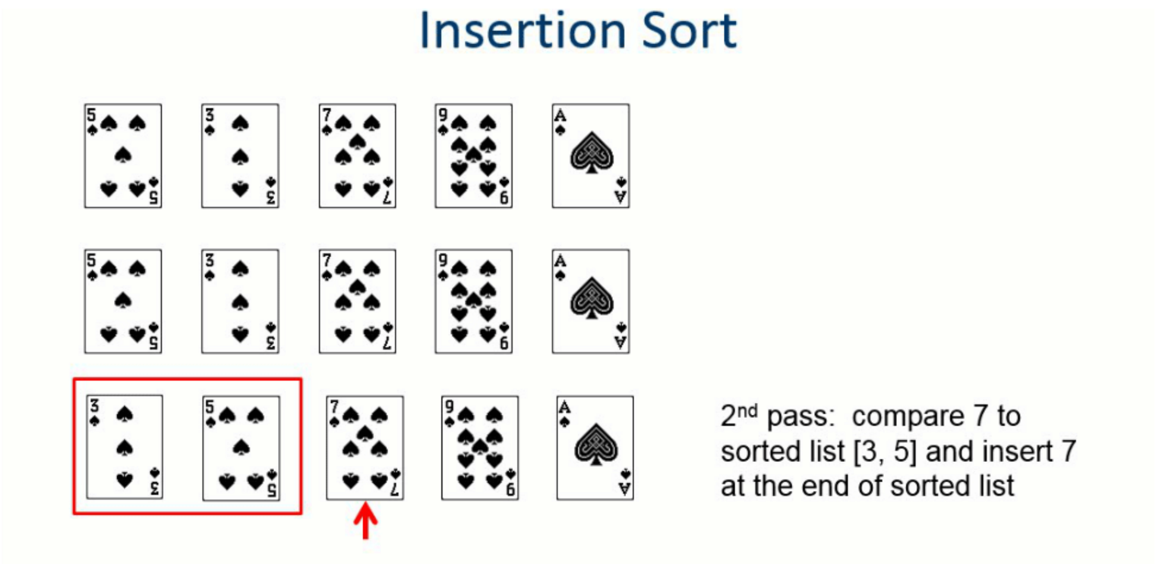
Computer Science A © 2016 Project Lead The Way, Inc.



Insertion Sort

1st pass: compare 3 to sorted list [5], insert 3 to left of 5 and shift 5 to the right

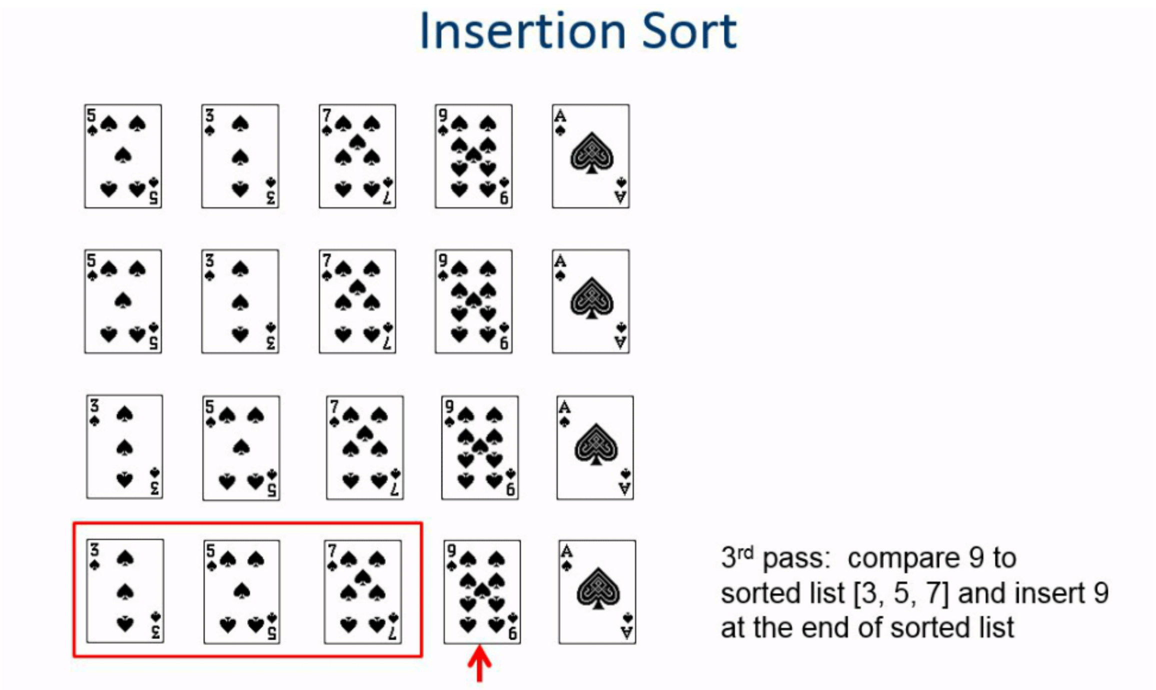
Computer Science A



Insertion Sort

2nd pass: compare 7 to sorted list [3, 5] and insert 7 at the end of sorted list

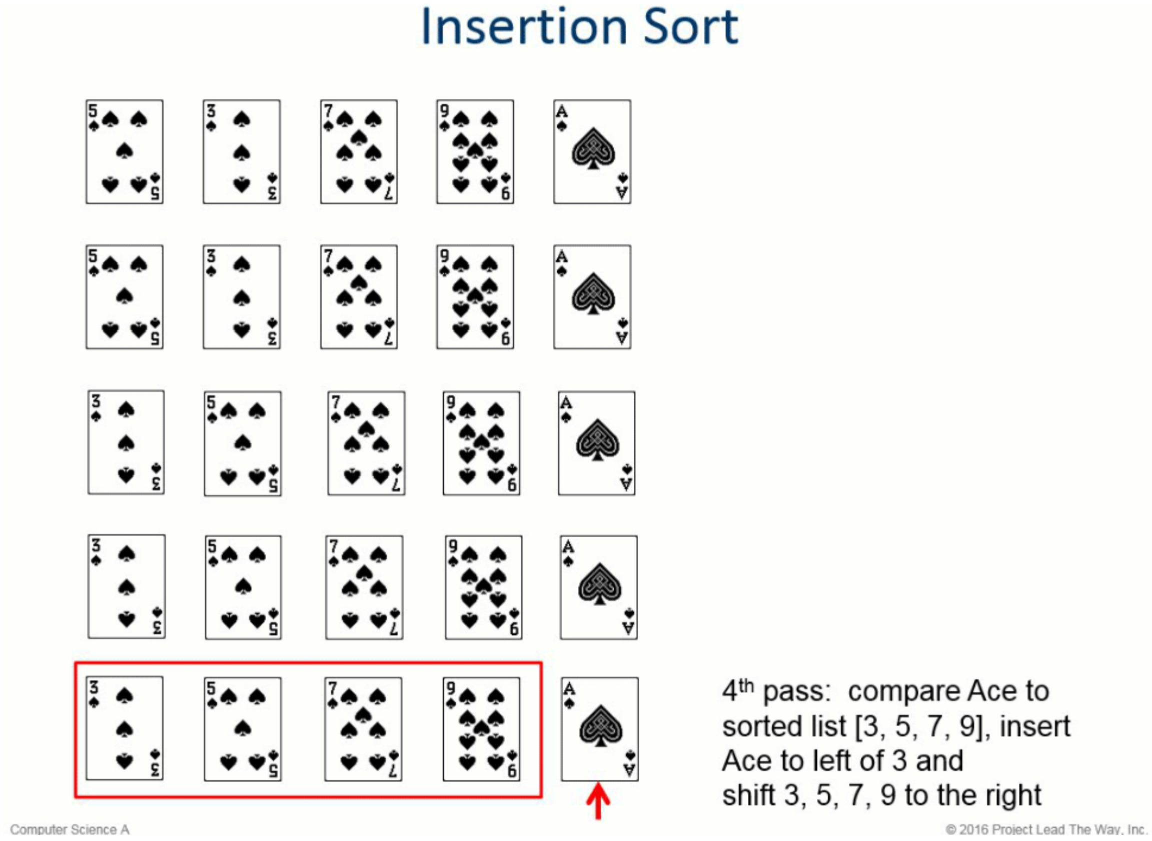
Computer Science A



Insertion Sort

3rd pass: compare 9 to sorted list [3, 5, 7] and insert 9 at the end of sorted list

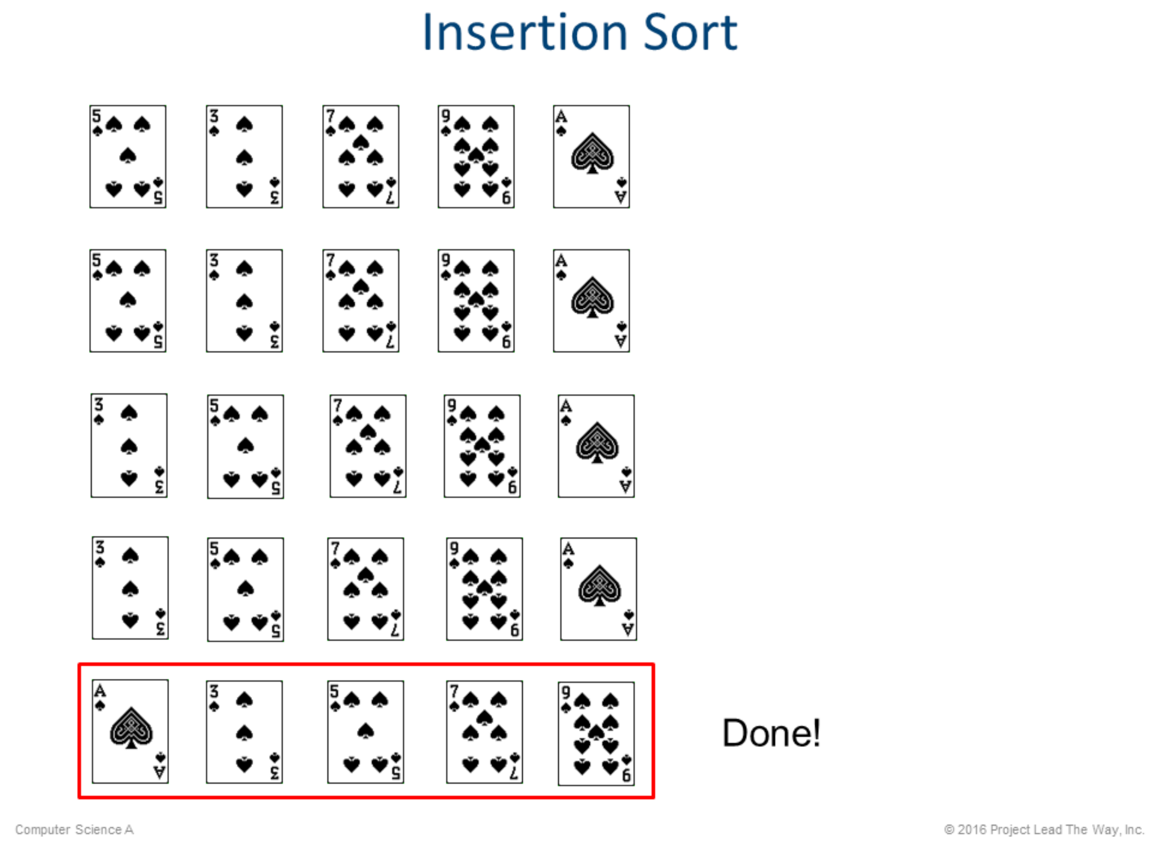
Computer Science A



Insertion Sort

4th pass: compare Ace to sorted list [3, 5, 7, 9], insert Ace to left of 3 and shift 3, 5, 7, 9 to the right

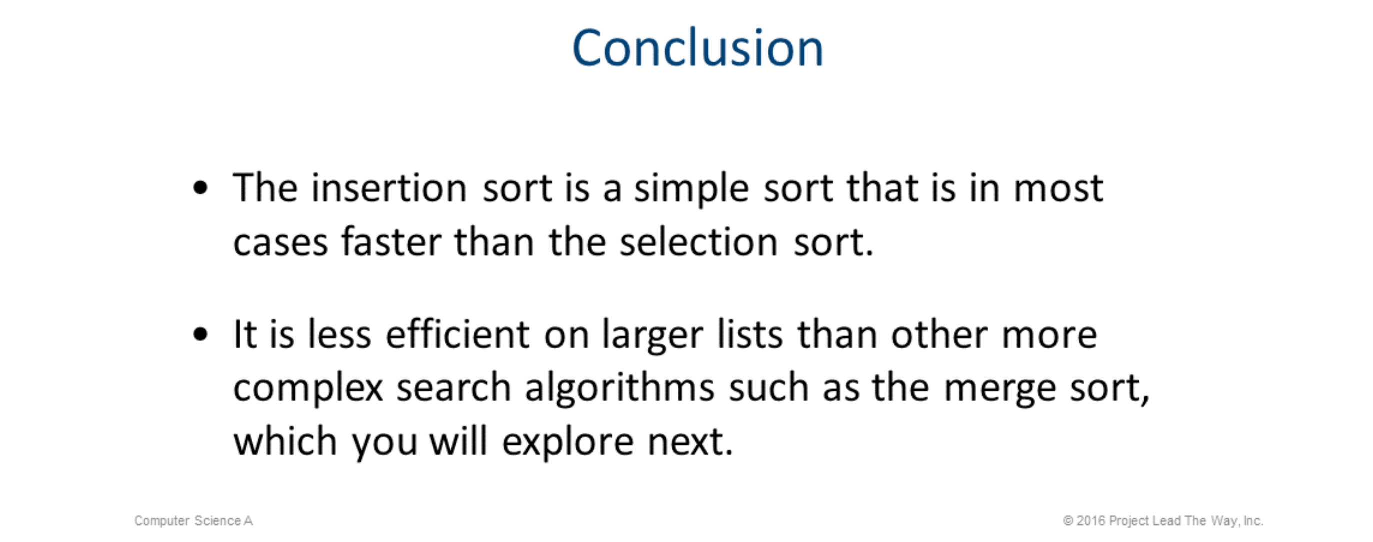
Computer Science A



Insertion Sort

Done!

Computer Science A



Conclusion

- The insertion sort is a simple sort that is in most cases faster than the selection sort.
- It is less efficient on larger lists than other more complex search algorithms such as the merge sort, which you will explore next.

Computer Science A

© 2016 Project Lead The Way, Inc.

4 Work in your group to use the insertion sort to sort through the same original list of five playing cards. In your notebook, draw the list of cards to illustrate how the sequence changes after each pass of the sort algorithm.

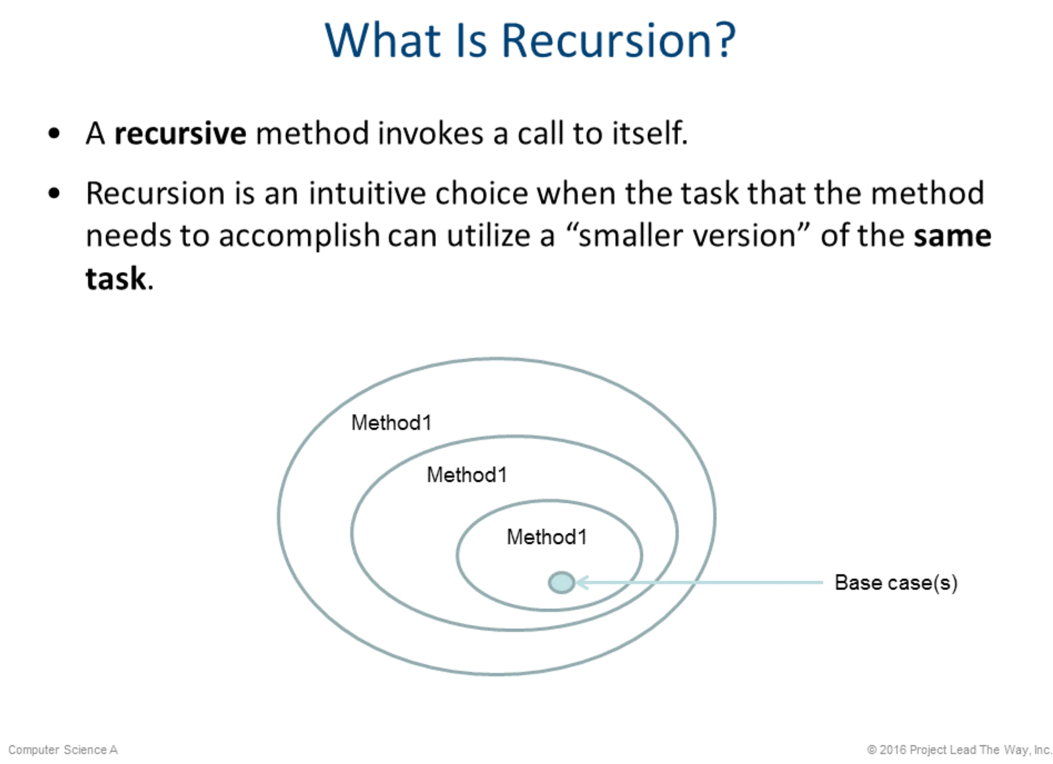
5 How does it compare to the selection sort?

Part III: Recursion

6 For an explanation of **recursion**, view the slideshow and then answer the following questions:

recursion

Recursion occurs when a method invokes a call to itself. Recursion is a natural choice when the task to be accomplished by the method can utilize "smaller versions" of the same task.

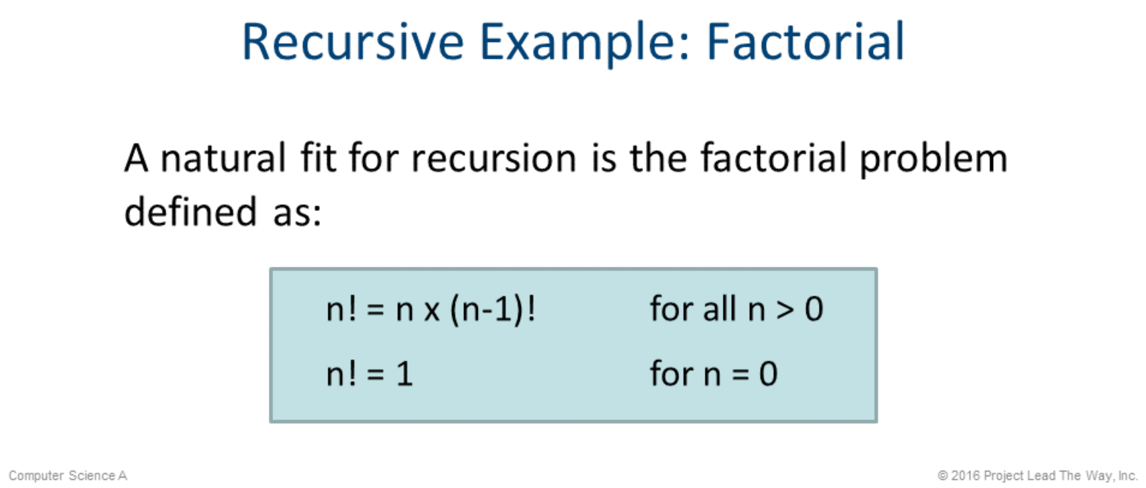


What Is Recursion?

- A **recursive** method invokes a call to itself.
- Recursion is an intuitive choice when the task that the method needs to accomplish can utilize a "smaller version" of the same **task**.

Computer Science A

© 2016 Project Lead The Way, Inc.

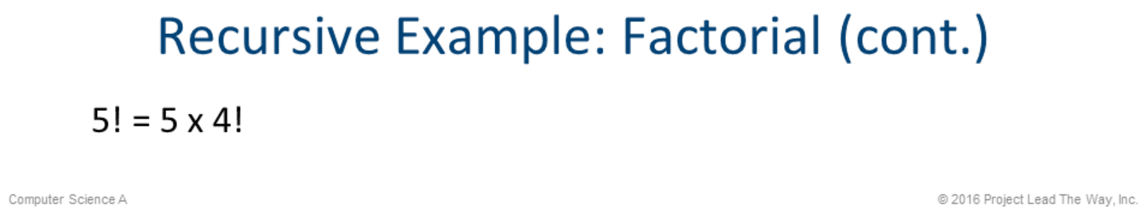


Recursive Example: Factorial

A natural fit for recursion is the factorial problem defined as:

n! = n x (n-1)!	for all n > 0
n! = 1	for n = 0

Computer Science A



Recursive Example: Factorial (cont.)

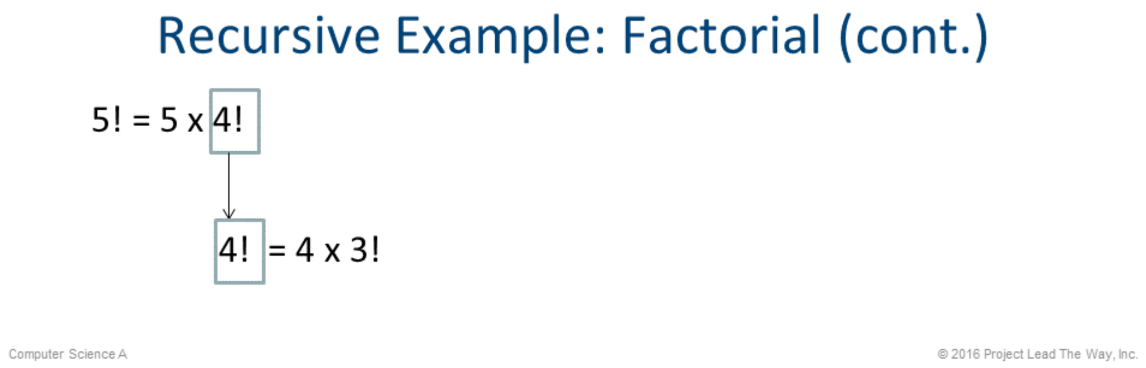
5! = 5 x 4!

Computer Science A

When you look at the recursive algorithm, you can conclude that the factorial problem can also be defined as “the product of a non-negative integer with all the positive integers that are below it”.

As the example above shows:

5! = 5 x (4 x (3 x (2 x (1 x 1)))) = 5 x 4 x 3 x 2 x 1



Recursive Example: Factorial (cont.)

5! = 5 x 4!

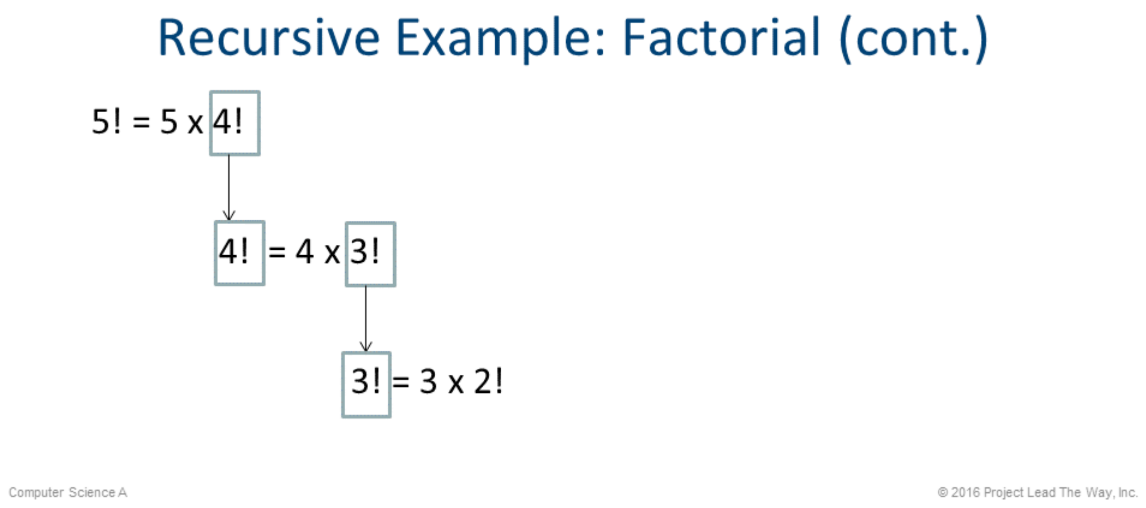
4! = 4 x 3!

Computer Science A

When you look at the recursive algorithm, you can conclude that the factorial problem can also be defined as “the product of a non-negative integer with all the positive integers that are below it”.

As the example above shows:

5! = 5 x (4 x (3 x (2 x (1 x 1)))) = 5 x 4 x 3 x 2 x 1



Recursive Example: Factorial (cont.)

$5! = 5 \times \boxed{4!}$

$\boxed{4!} = 4 \times \boxed{3!}$

$\boxed{3!} = 3 \times 2!$

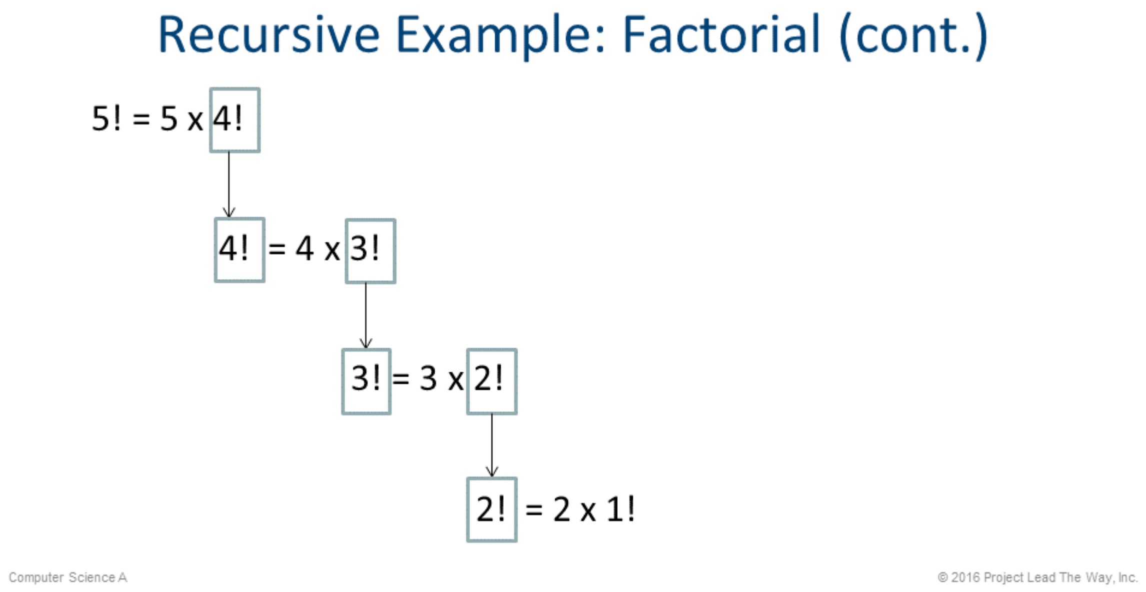
Computer Science A

© 2016 Project Lead The Way, Inc.

When you look at the recursive algorithm, you can conclude that the factorial problem can also be defined as “the product of a non-negative integer with all the positive integers that are below it”.

As the example above shows:

5! = 5 x (4 x (3 x (2 x (1 x 1)))) = 5 x 4 x 3 x 2 x 1



Recursive Example: Factorial (cont.)

$5! = 5 \times \boxed{4!}$

$\boxed{4!} = 4 \times \boxed{3!}$

$\boxed{3!} = 3 \times \boxed{2!}$

$\boxed{2!} = 2 \times 1!$

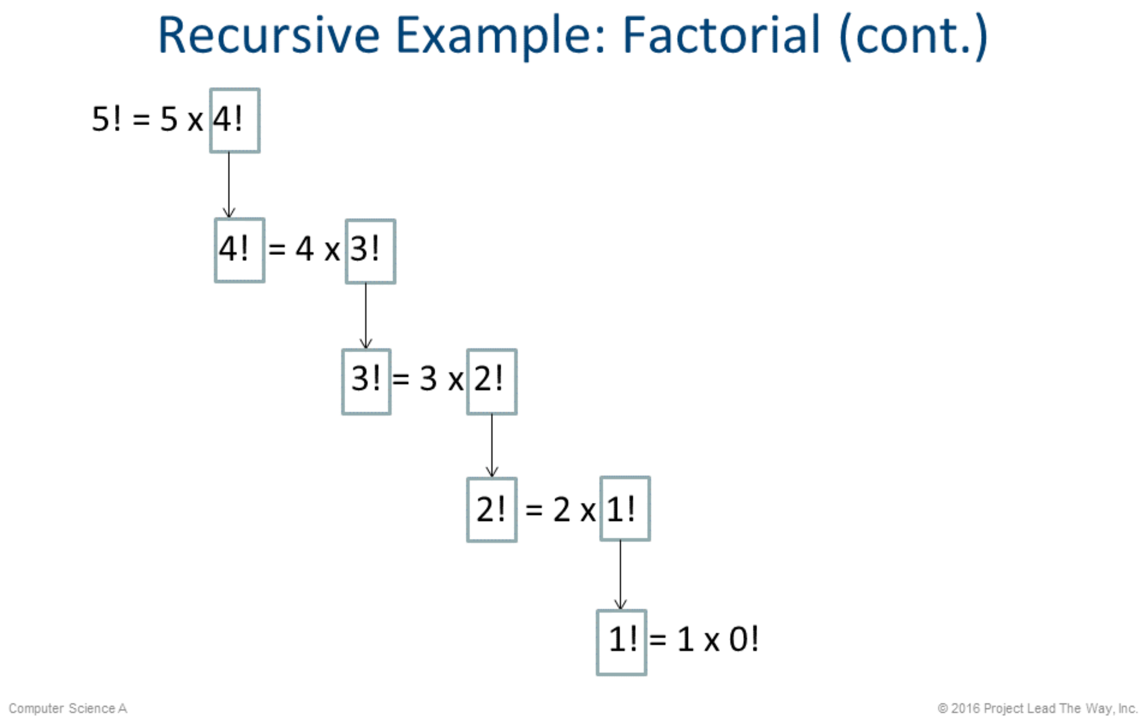
Computer Science A

© 2016 Project Lead The Way, Inc.

When you look at the recursive algorithm, you can conclude that the factorial problem can also be defined as “the product of a non-negative integer with all the positive integers that are below it”.

As the example above shows:

5! = 5 x (4 x (3 x (2 x (1 x 1)))) = 5 x 4 x 3 x 2 x 1



Recursive Example: Factorial (cont.)

5! = 5 x 4!

4! = 4 x 3!

3! = 3 x 2!

2! = 2 x 1!

1! = 1 x 0!

Computer Science A

© 2016 Project Lead The Way, Inc.

When you look at the recursive algorithm, you can conclude that the factorial problem can also be defined as "the product of a non-negative integer with all the positive integers that are below it".

As the example above shows:

5! = 5 x (4 x (3 x (2 x (1 x 1)))) = 5 x 4 x 3 x 2 x 1

a. Explain recursion in your own words and give an example.

b. How can you guarantee that a recursive method does not run indefinitely?

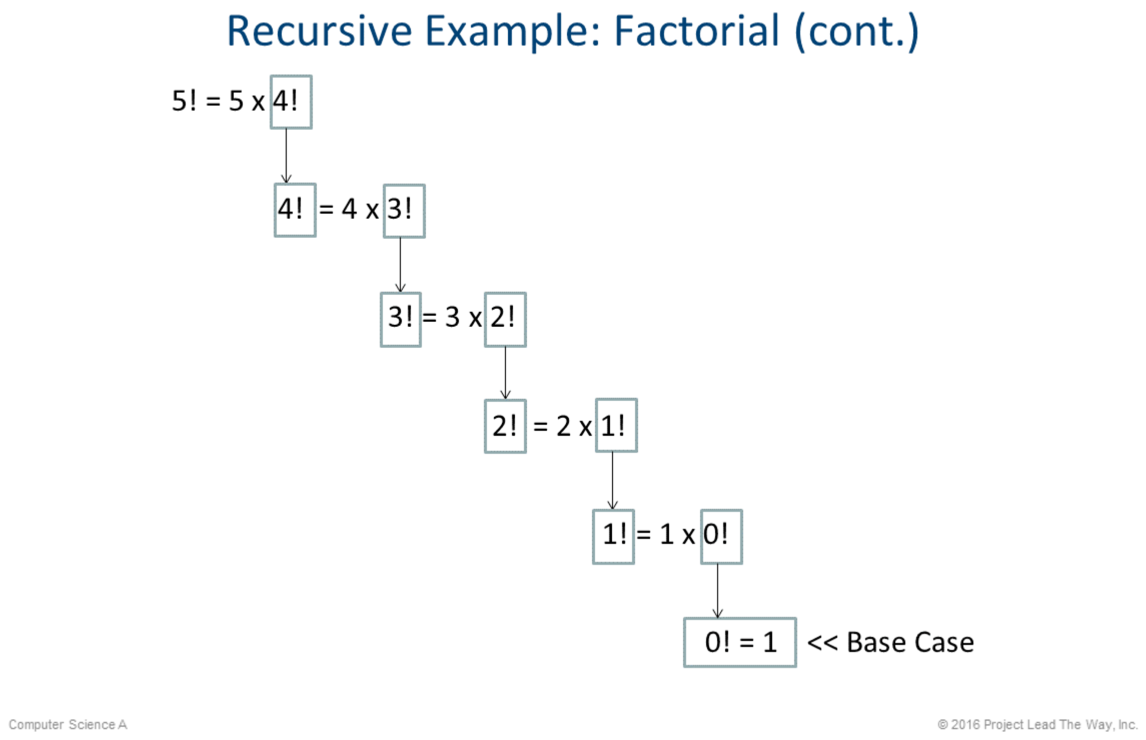
c. Recall the Fibonacci sequence is sequence numbers where each number is a sum of the previous two numbers. So, the first 6 numbers in a Fibonnaci sequence are 1, 1, 2, 3, 5, 8.

   Hand trace the following method showing the recursive calls, the values returned from each call, and the final value returned by `fib(6)`. It might help to write your answers in a tree map.

```
public int fib(int n) {

    if(n == 1 || n == 2)
        return 1;
    else
        return fib(n-1) + fib(n-2);
}
```

Part IV: Merge Sort

7 For an explanation of merge sort, view the slideshow.



Recursive Example: Factorial (cont.)

5! = 5 x 4!

4! = 4 x 3!

3! = 3 x 2!

2! = 2 x 1!

1! = 1 x 0!

0! = 1 << Base Case

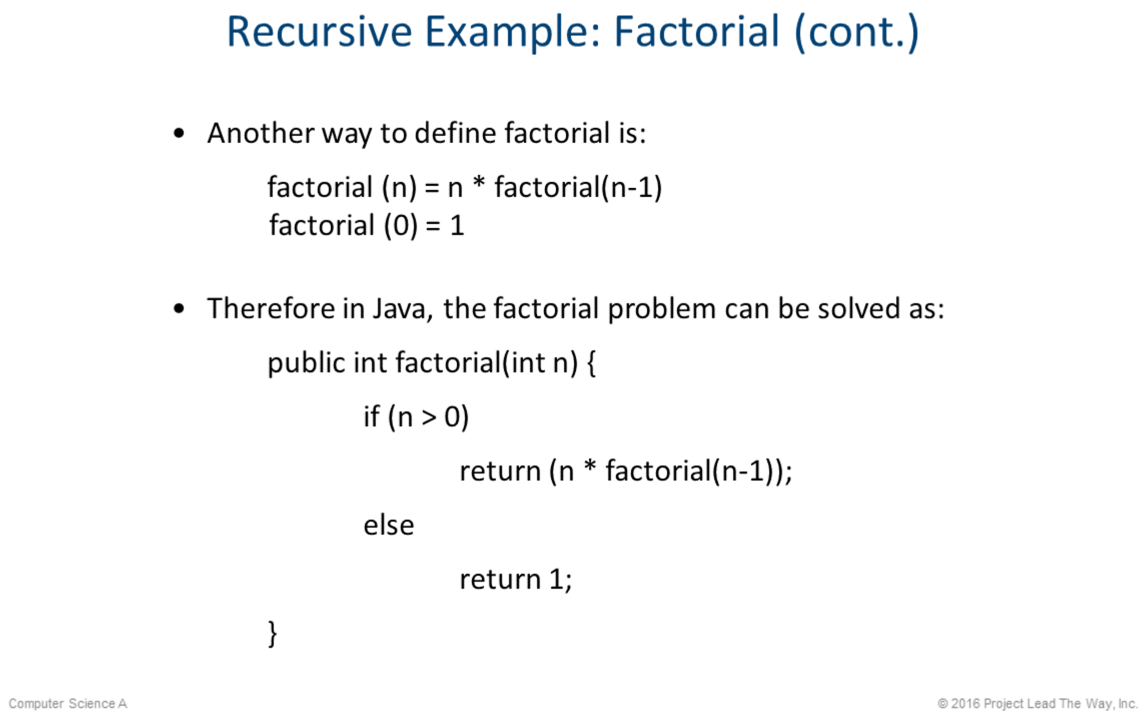
Computer Science A

© 2016 Project Lead The Way, Inc.

When you look at the recursive algorithm, you can conclude that the factorial problem can also be defined as "the product of a non-negative integer with all the positive integers that are below it".

As the example above shows:

5! = 5 x (4 x (3 x (2 x (1 x 1)))) = 5 x 4 x 3 x 2 x 1



Recursive Example: Factorial (cont.)

- Another way to define factorial is:

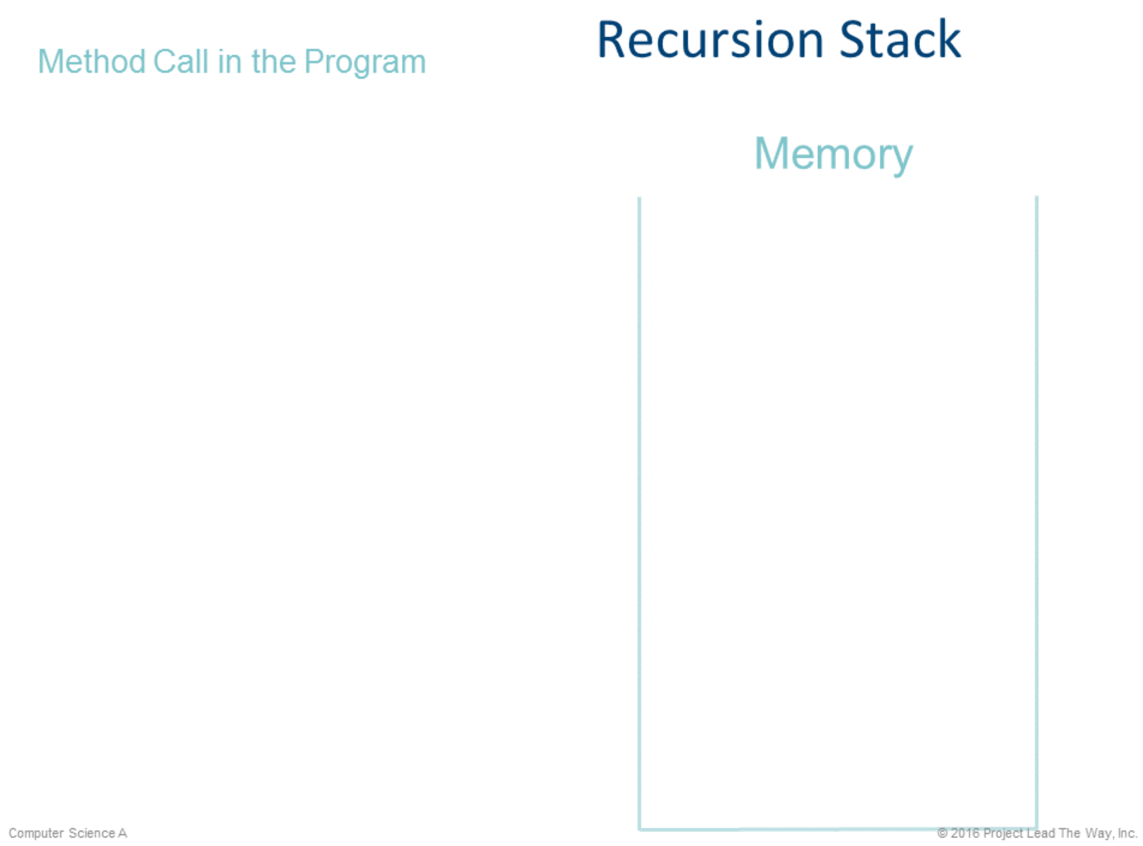
  factorial (n) = n * factorial(n-1)
  factorial (0) = 1

- Therefore in Java, the factorial problem can be solved as:

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

Computer Science A

© 2016 Project Lead The Way, Inc.



Recursion Stack

Method Call in the Program

Memory

Computer Science A

© 2016 Project Lead The Way, Inc.

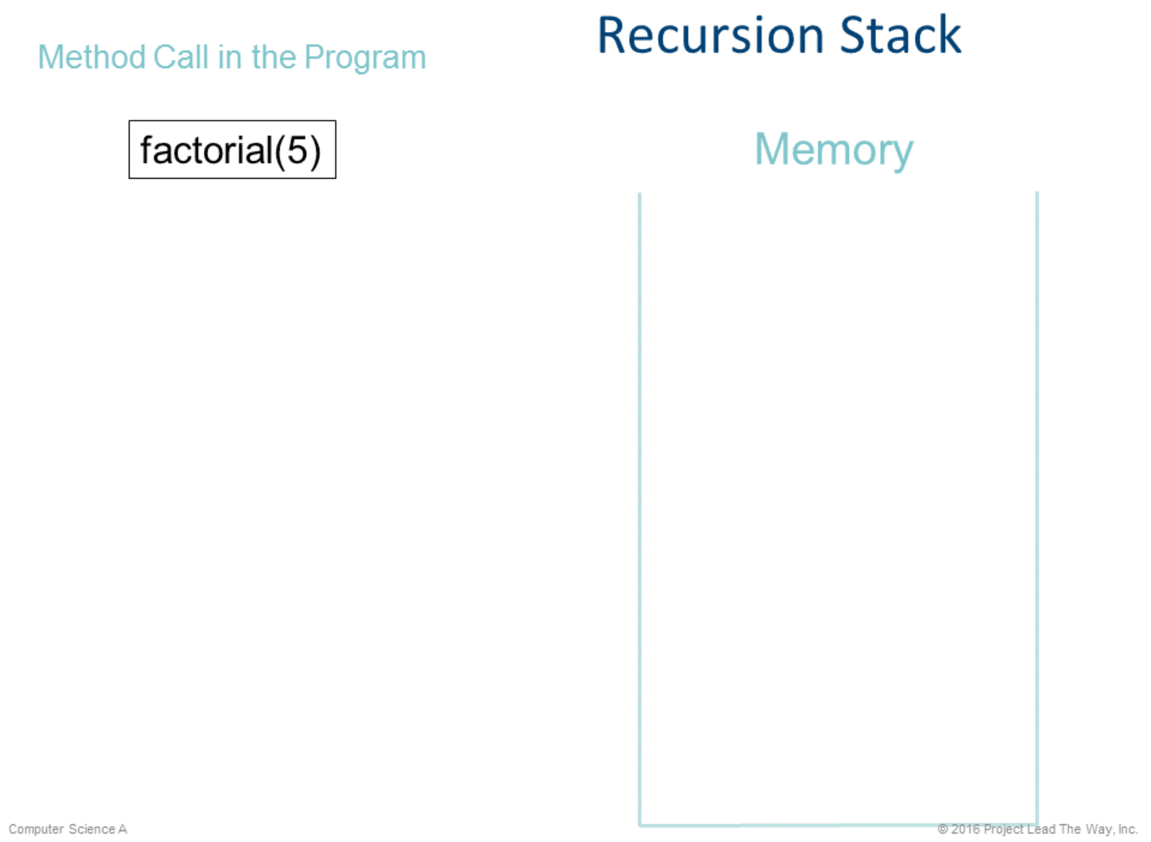
To keep track of recursion, it uses a structure called a “stack”, which is a memory structure similar to a stack of paper:

- You write information on a new sheet of paper and add it to the top of the stack.
- The top of the stack is the only sheet accessible at any time.
- When done with the top sheet of paper, you discard it to reveal the next sheet of paper in the stack.

Similarly, every time a recursive call is invoked within the method, the system “pushes” the recursive method to the stack by writing its information onto a new “stack frame” (an allocated space in memory) and adding that stack frame to the stack. The information that is written onto the stack frame includes a copy of the method definition and the parameter values.

When a recursive call completes its task without invoking a new recursive call (which means it is a base case), the system completes the computation in the stack frame and “pops” it out of the stack. Then it addresses the computation in the next stack frame, which is now on top of the stack, and pops that stack frame out of the stack. This continues until the last stack frame is completed and the stack is empty.

So the first recursive call is the last one to be computed! This is why a stack structure is referred to as a Last-in/First-out structure, or LIFO.



Recursion Stack

Method Call in the Program

factorial(5)

Memory

Computer Science A

© 2016 Project Lead The Way, Inc.

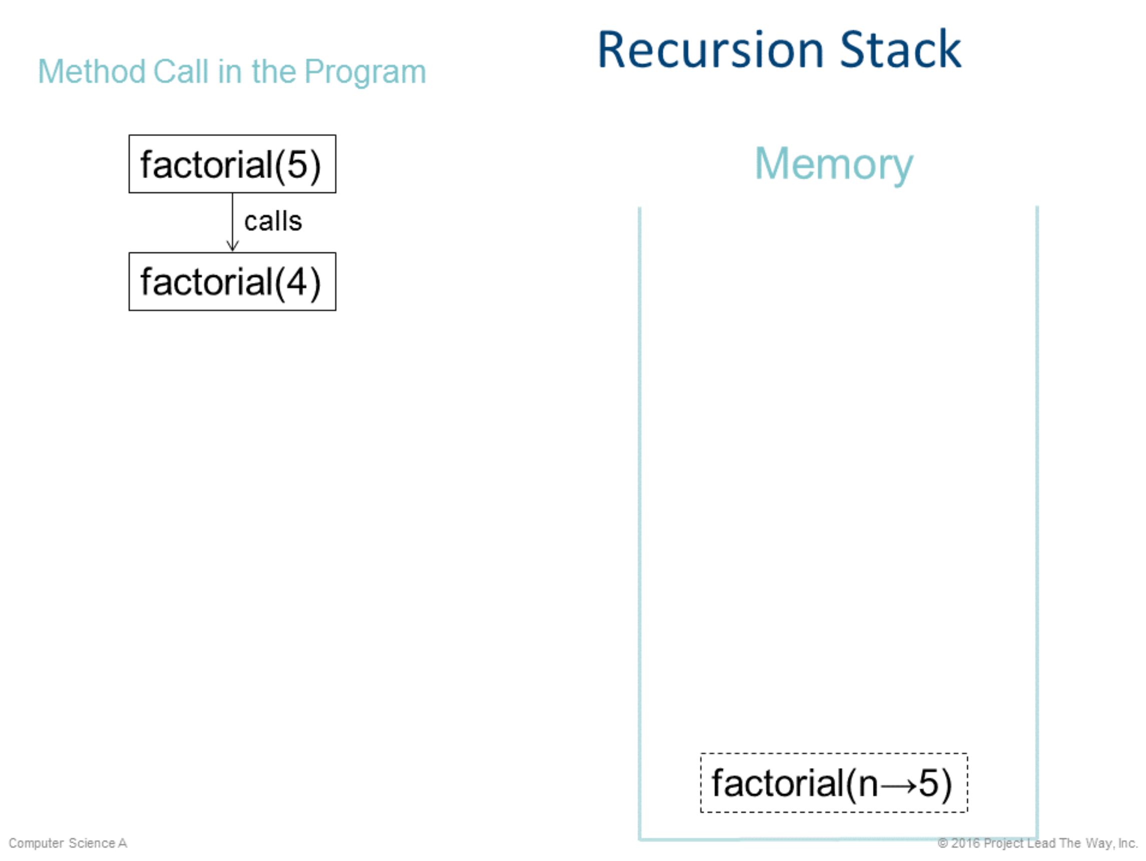
To keep track of recursion, it uses a structure called a “stack”, which is a memory structure similar to a stack of paper:

- You write information on a new sheet of paper and add it to the top of the stack.
- The top of the stack is the only sheet accessible at any time.
- When done with the top sheet of paper, you discard it to reveal the next sheet of paper in the stack.

Similarly, every time a recursive call is invoked within the method, the system “pushes” the recursive method to the stack by writing its information onto a new “stack frame” (an allocated space in memory) and adding that stack frame to the stack. The information that is written onto the stack frame includes a copy of the method definition and the parameter values.

When a recursive call completes its task without invoking a new recursive call (which means it is a base case), the system completes the computation in the stack frame and “pops” it out of the stack. Then it addresses the computation in the next stack frame, which is now on top of the stack, and pops that stack frame out of the stack. This continues until the last stack frame is completed and the stack is empty.

So the first recursive call is the last one to be computed! This is why a stack structure is referred to as a Last-in/First-out structure, or LIFO.



Recursion Stack

Method Call in the Program

factorial(5)

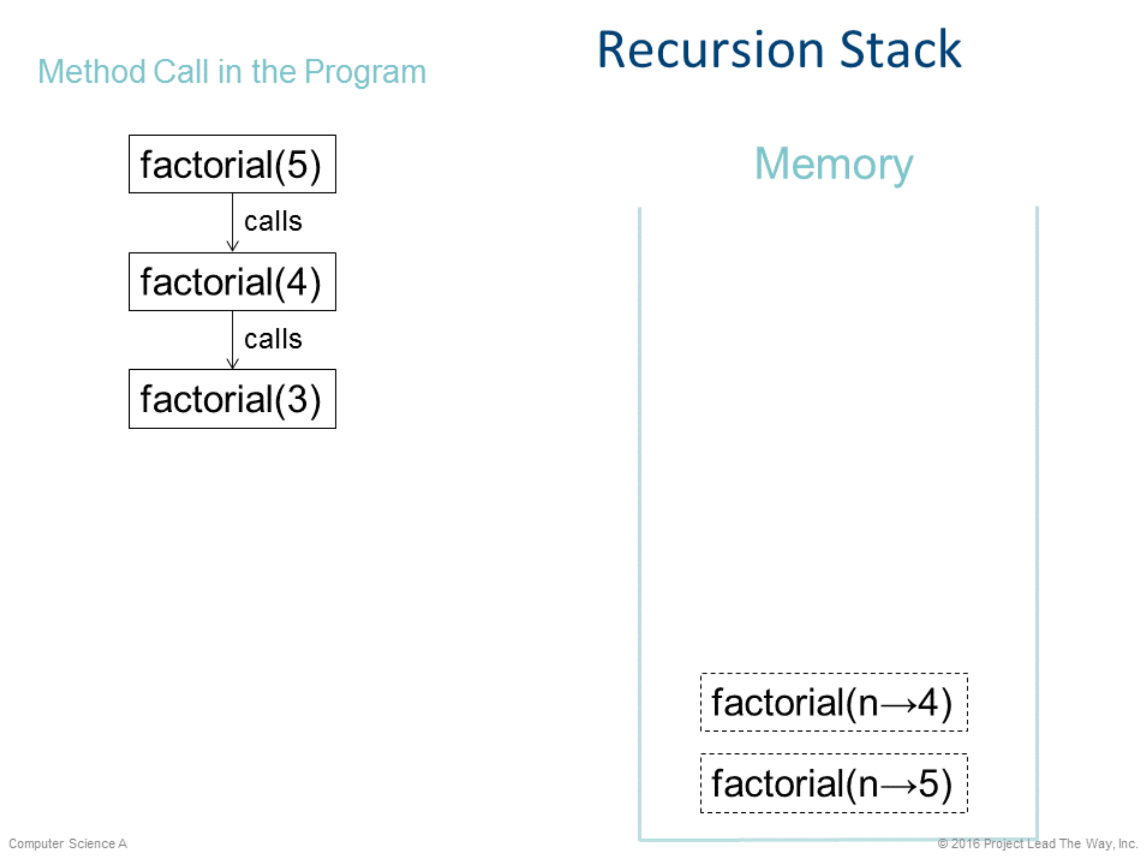
calls

factorial(4)

Memory

factorial(n→5)

Computer Science A



Recursion Stack

Method Call in the Program

factorial(5)

calls

factorial(4)

calls

factorial(3)

Memory

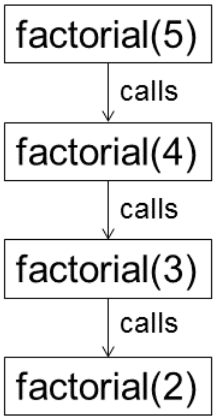
factorial(n→4)

factorial(n→5)

Computer Science A

Recursion Stack

Method Call in the Program


factorial(5)

calls

factorial(4)

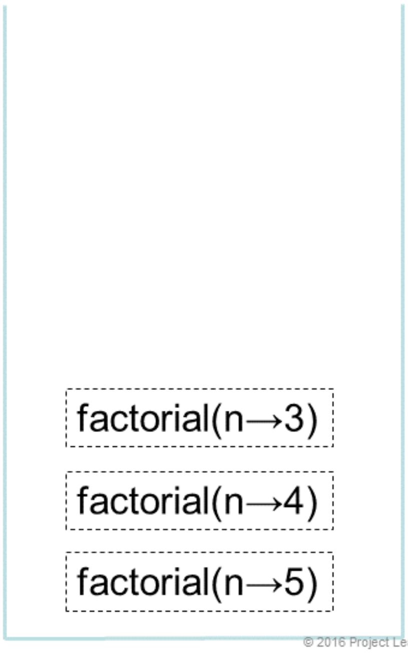
calls

factorial(3)

calls

factorial(2)

Memory


factorial(n→3)

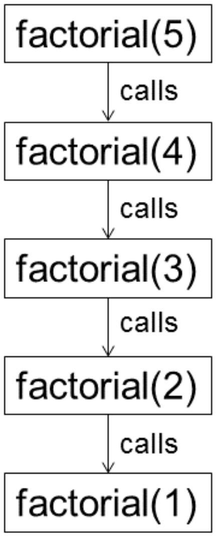
factorial(n→4)

factorial(n→5)

Computer Science A

Recursion Stack

Method Call in the Program


factorial(5)

calls

factorial(4)

calls

factorial(3)

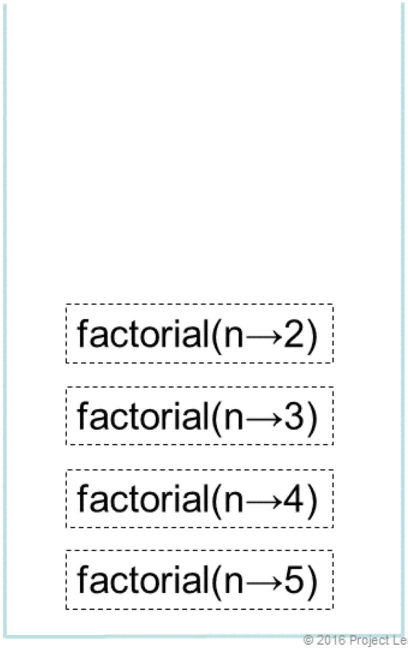
calls

factorial(2)

calls

factorial(1)

Memory

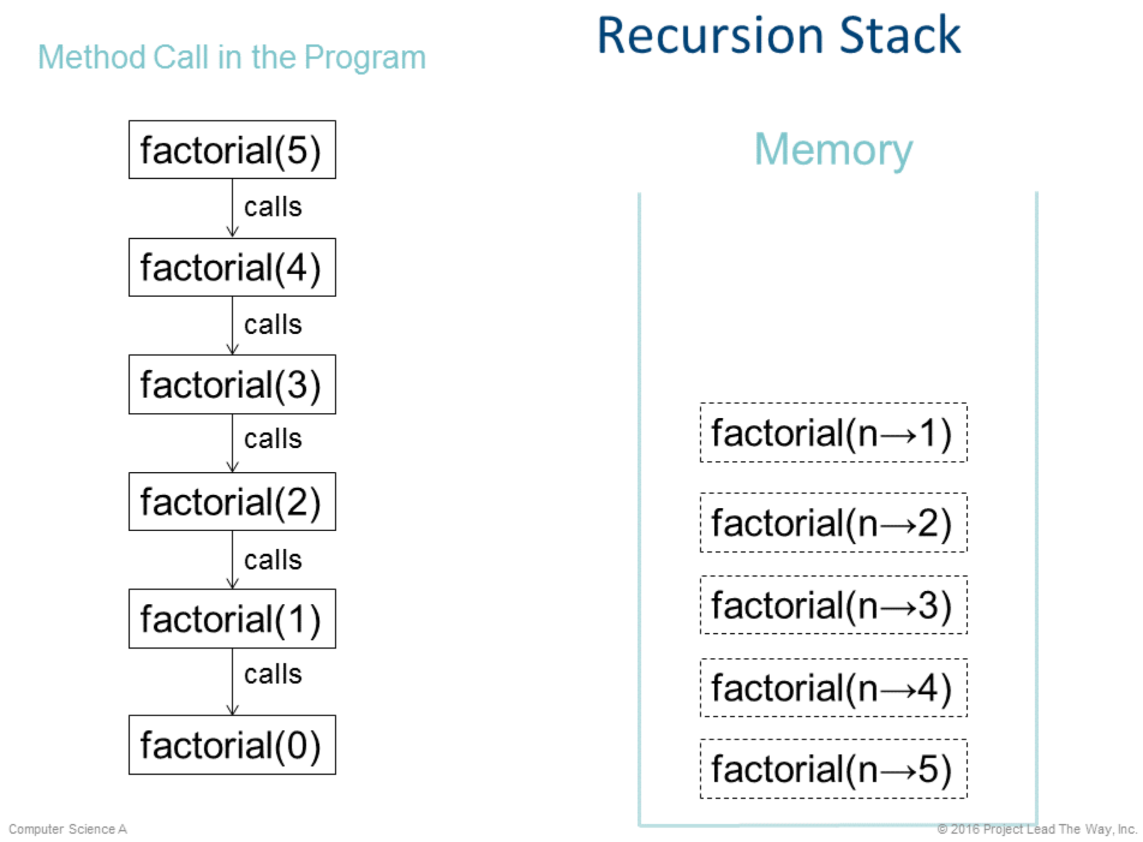

factorial(n→2)

factorial(n→3)

factorial(n→4)

factorial(n→5)

Computer Science A



Recursion Stack

Method Call in the Program

factorial(5)
calls
factorial(4)
calls
factorial(3)
calls
factorial(2)
calls
factorial(1)
calls
factorial(0)

Memory

factorial(n→1)
factorial(n→2)
factorial(n→3)
factorial(n→4)
factorial(n→5)

Computer Science A

© 2016 Project Lead The Way, Inc.

8 Work in your group to use merge sort to sort through the same original list of five playing cards. In your notebook, draw the list of cards to illustrate how the sequence changes after each pass of the sort algorithm.

9 How does it compare to the selection and insertion sorts?

Part V: Applying a Sort in TripTracker App

10 You are going to apply a sort method to your Trips list. But before you start writing the code, consider these questions:

a. Where will the method be called from?

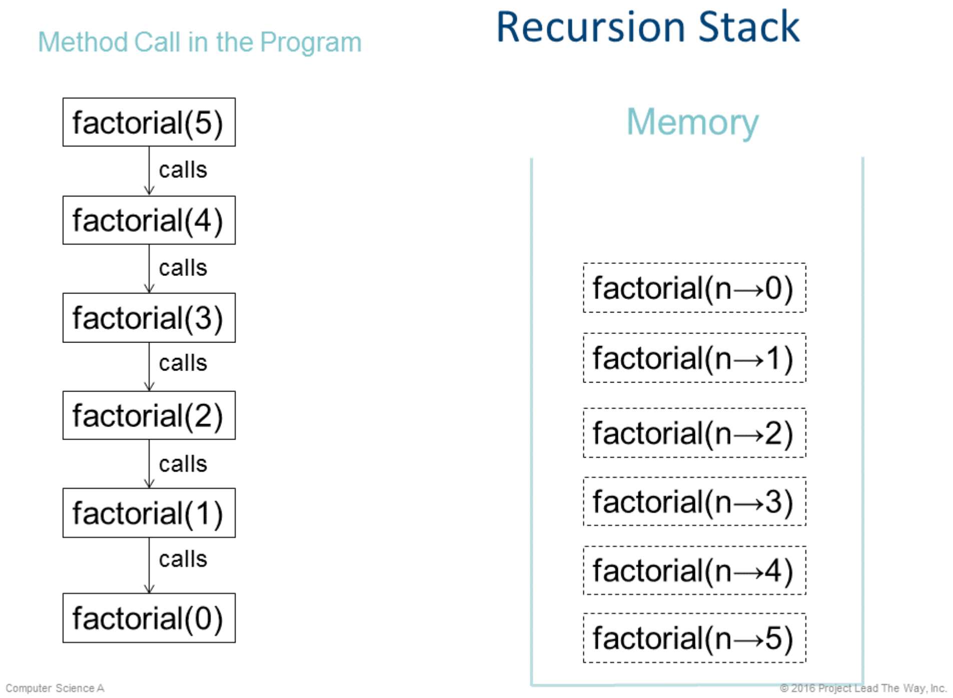
b. What data structure does it execute on?

c. Is it good to make this method accessible/reusable?

Take a few minutes to discuss these questions within your group.

Slides 34 and 35 in the *3.1.7 Sort Algorithms* presentation describe some considerations when choosing a sort solution in your app.

11. Compare and contrast the options with your group and then share with your class.



Method Call in the Program

Recursion Stack

factorial(5)

calls

factorial(4)

calls

factorial(3)

calls

factorial(2)

calls

factorial(1)

calls

factorial(0)

Memory

factorial(n→0)

factorial(n→1)

factorial(n→2)

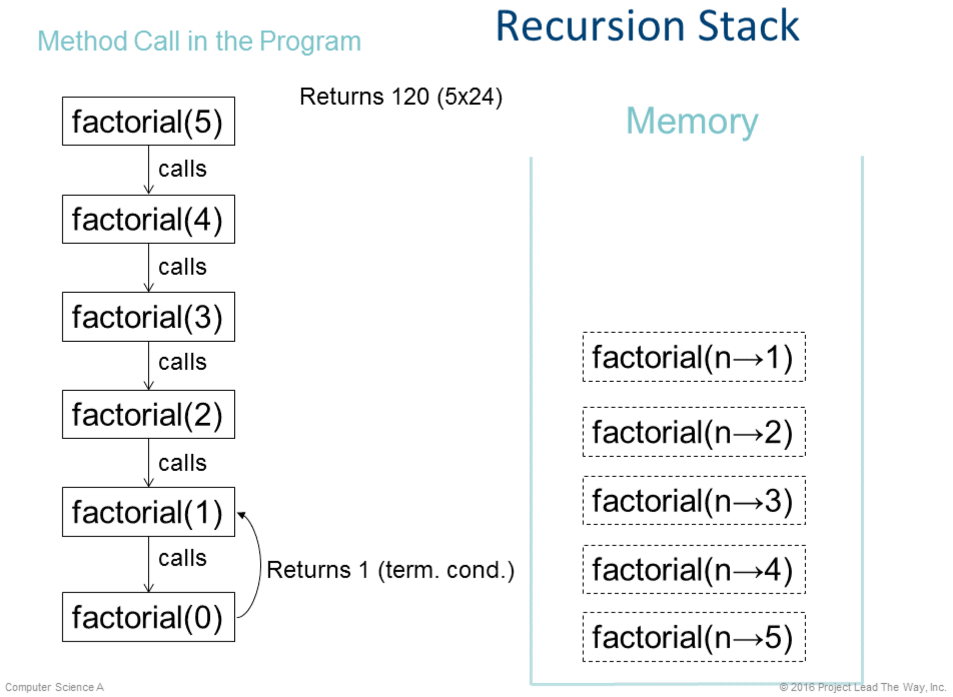
factorial(n→3)

factorial(n→4)

factorial(n→5)

Computer Science A

© 2016 Project Lead The Way, Inc.



Method Call in the Program

Recursion Stack

Returns 120 (5x24)

factorial(5)

calls

factorial(4)

calls

factorial(3)

calls

factorial(2)

calls

factorial(1)

calls

factorial(0)

Returns 1 (term. cond.)

Memory

factorial(n→1)

factorial(n→2)

factorial(n→3)

factorial(n→4)

factorial(n→5)

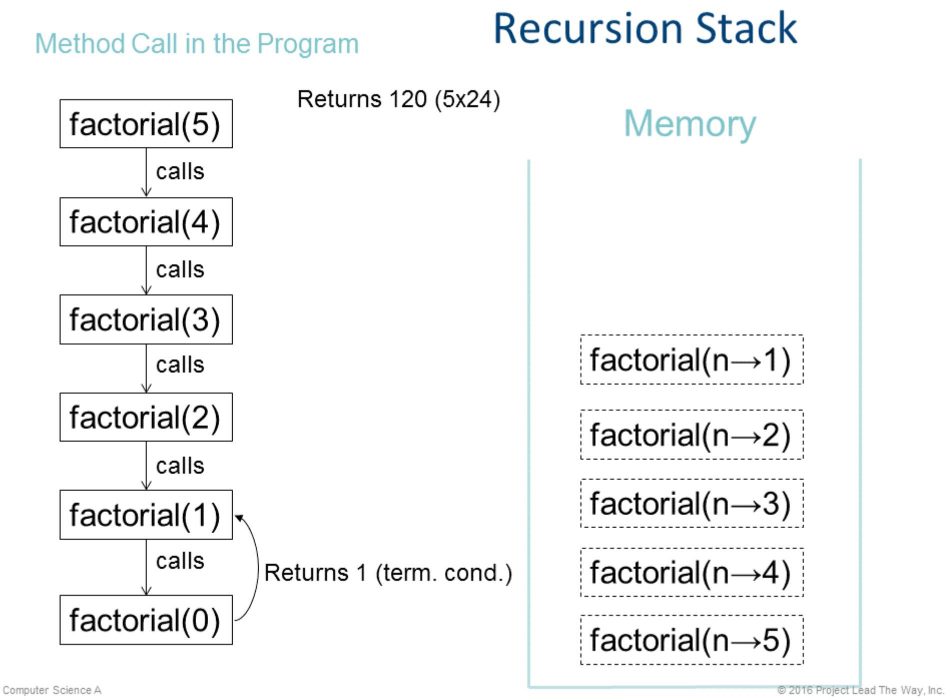
Computer Science A

© 2016 Project Lead The Way, Inc.

12. Open your TripTracker app in Android Studio. If you were unable to complete *Activity 3.1.6 Public vs. Private Trips*, obtain and import `3.1.6TripTracker_Solution` as directed by your teacher.

13. Run the app and add a few trips to your list with a variety of names so you have at least five or six trips defined.

14 In this activity, you will walk through Option 3 identified in slide 35 of the presentation, to enable flexibility and reusability of the sort method with any class in the future. For an introduction to the `Comparable` interface, refer to slides 36–37 in the *3.1.7 Sort Algorithms* presentation.



Method Call in the Program

Recursion Stack

Returns 120 (5x24)

Memory

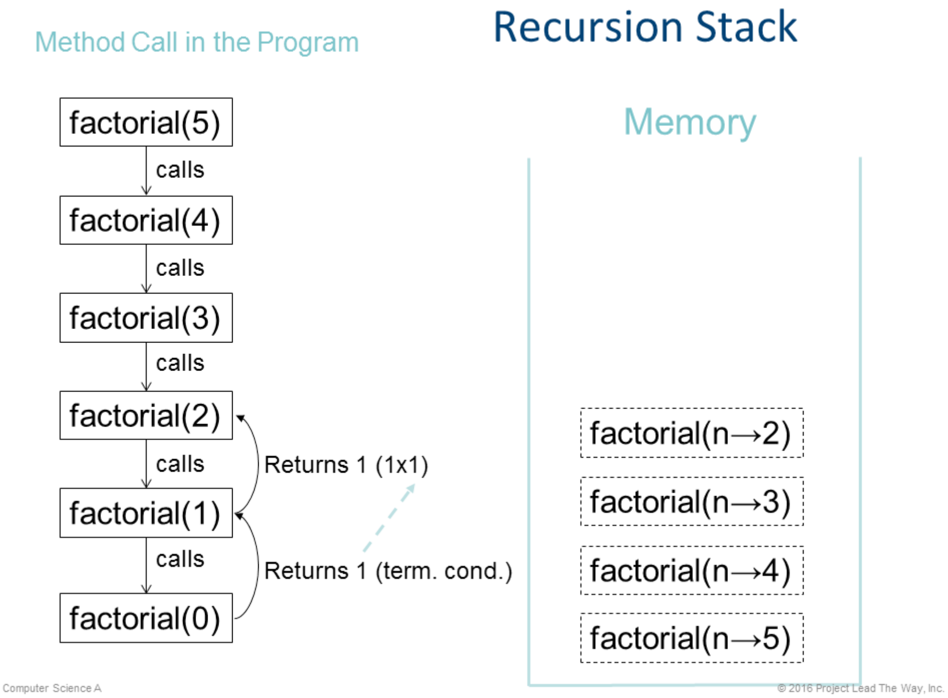
factorial(5)
calls
factorial(4)
calls
factorial(3)
calls
factorial(2)
calls
factorial(1)
calls
factorial(0)

Returns 1 (term. cond.)

factorial(n→1)
factorial(n→2)
factorial(n→3)
factorial(n→4)
factorial(n→5)

Computer Science A

© 2016 Project Lead The Way, Inc.



Method Call in the Program

Recursion Stack

Memory

factorial(5)
calls
factorial(4)
calls
factorial(3)
calls
factorial(2)
calls
factorial(1)
calls
factorial(0)

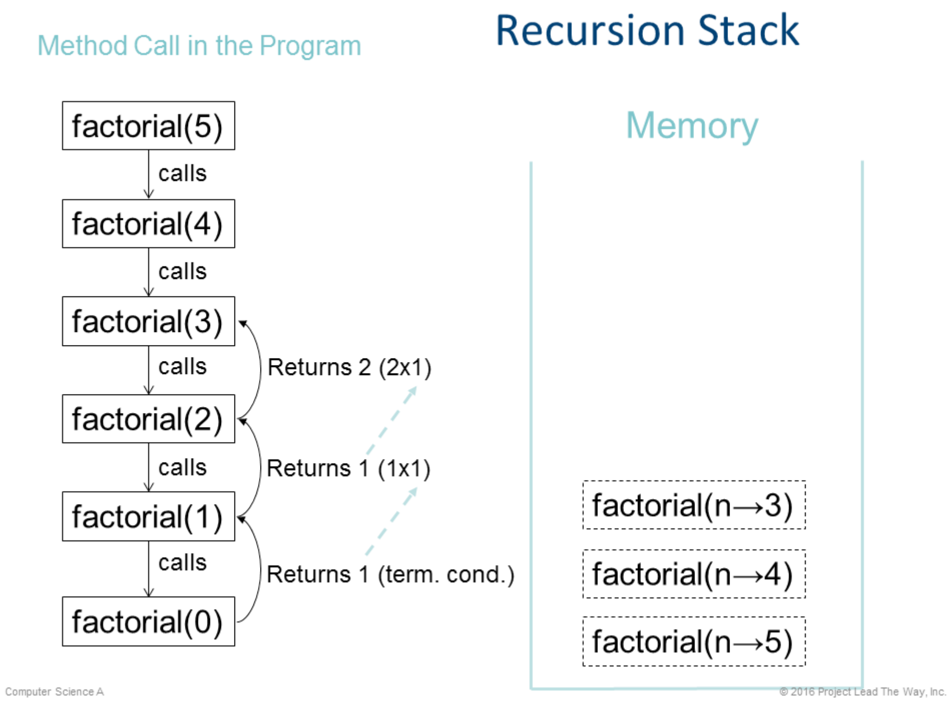
Returns 1 (1x1)

Returns 1 (term. cond.)

factorial(n→2)
factorial(n→3)
factorial(n→4)
factorial(n→5)

Computer Science A

© 2016 Project Lead The Way, Inc.



15. Because you will be comparing `Trip` objects in the sort method, modify the `Trip` class so that it:

    a. Implements `Comparable`. Notice, you can *implement* more than one class, but you can only *extend* once.

    b. Overrides the `compareTo(Object o)` method so that it compares the trips using their trip names and returns an integer based on the semantics explained on slide 37.

    Hint: You will have to cast the o *object to a* `Trip` *object.*

NOTE

It is good practice to use a try/catch block to handle exceptions.

16. Create a new Java class called `ArrayListSorter`.

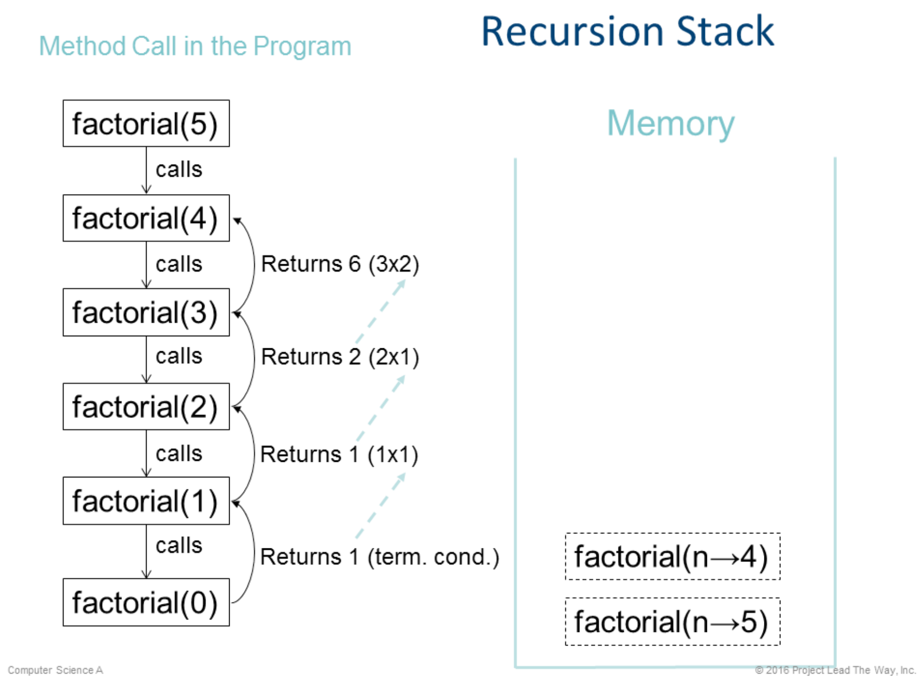
17. The following code shows the insertion sort algorithm. Copy the `insertionSort` method into the `ArrayListSorter` class.

```
public static void insertionSort(ArrayList list) {

    for (int j = 1; j < list.size(); j++) {
       Comparable temp = (Comparable)list.get(j);
       int possibleIndex = j;

       while (possibleIndex > 0 && temp.compareTo(list.
       get(possibleIndex - 1)) < 0) {
           list.set(possibleIndex, list.get(possibleIndex - 1));
           possibleIndex--;
      }
      list.set(possibleIndex, temp);

   }
}
```

18 For an explanation of the code above, refer to slide 38 in the *3.1.7 Sort Algorithms* presentation. Then answer the following questions:

a. What happens if you use the keyword `this` for accessing the data structure rather than sending it in as a parameter (list)?

b. Why use `Comparable` for `temp’s` data type, rather than `Trip`?



Recursion Stack

Computer Science A

© 2016 Project Lead The Way, Inc.

19 In `refreshTripList`, sort your `mTrips` array using your new insertion sort method.

With this sort, you have made other code in this method unnecessary or *obsolete*. What code was made obsolete when you sorted mTrips?

20 Test your app to make sure that the trips are sorted by name in ascending order.

CHALLENGES

□ Modify the insertion sort so that you can choose to sort the list in ascending or descending order.

□ Write your own selection sort method in `ArrayListSorter` and test your code.

□ Write your own merge sort method in `ArrayListSorter` and test your code.

CONCLUSION

1. Now that you have learned all three sort algorithms, explain how the algorithms might work better in certain circumstances, and why some algorithms might be preferred over others. Keep in mind the size and original state of the list.
2. Explain how implementation of the sort algorithm in Trip Tracker maintained abstraction and encapsulation.

Sorting Algorithms Supplemental Material

This document contains pseudocode for the following sort algorithms discussed in this activity:

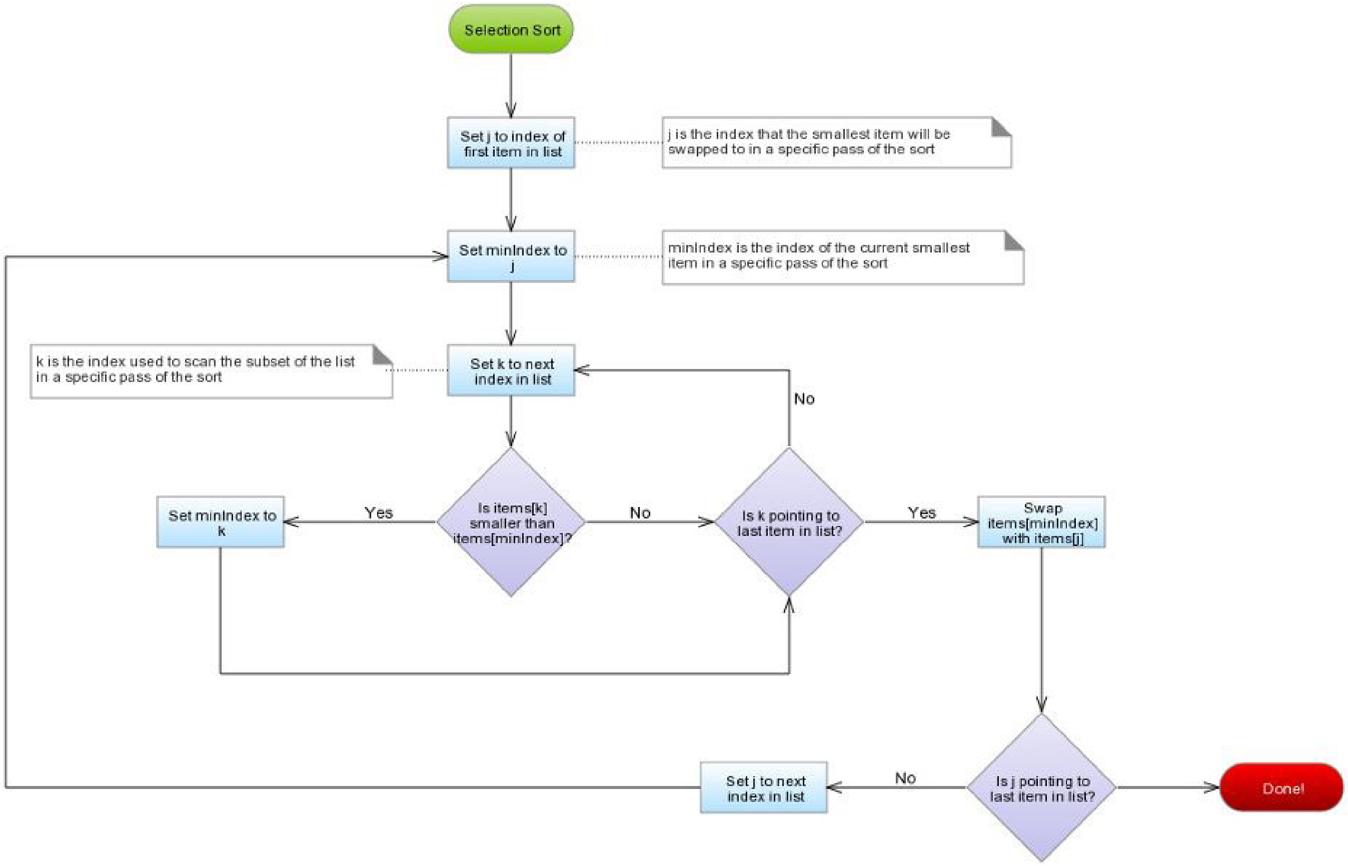
- Selection Sort
- Insertion Sort
- Merge Sort

For each algorithm, the document also provides a flowchart. The flowcharts are a visual representation of the algorithm.

You can also find fun and engaging videos on the Internet that illustrate the sort of algorithms discussed in this activity. Good search phrases might be “sort algorithm illustration” or “sort algorithm animation.”

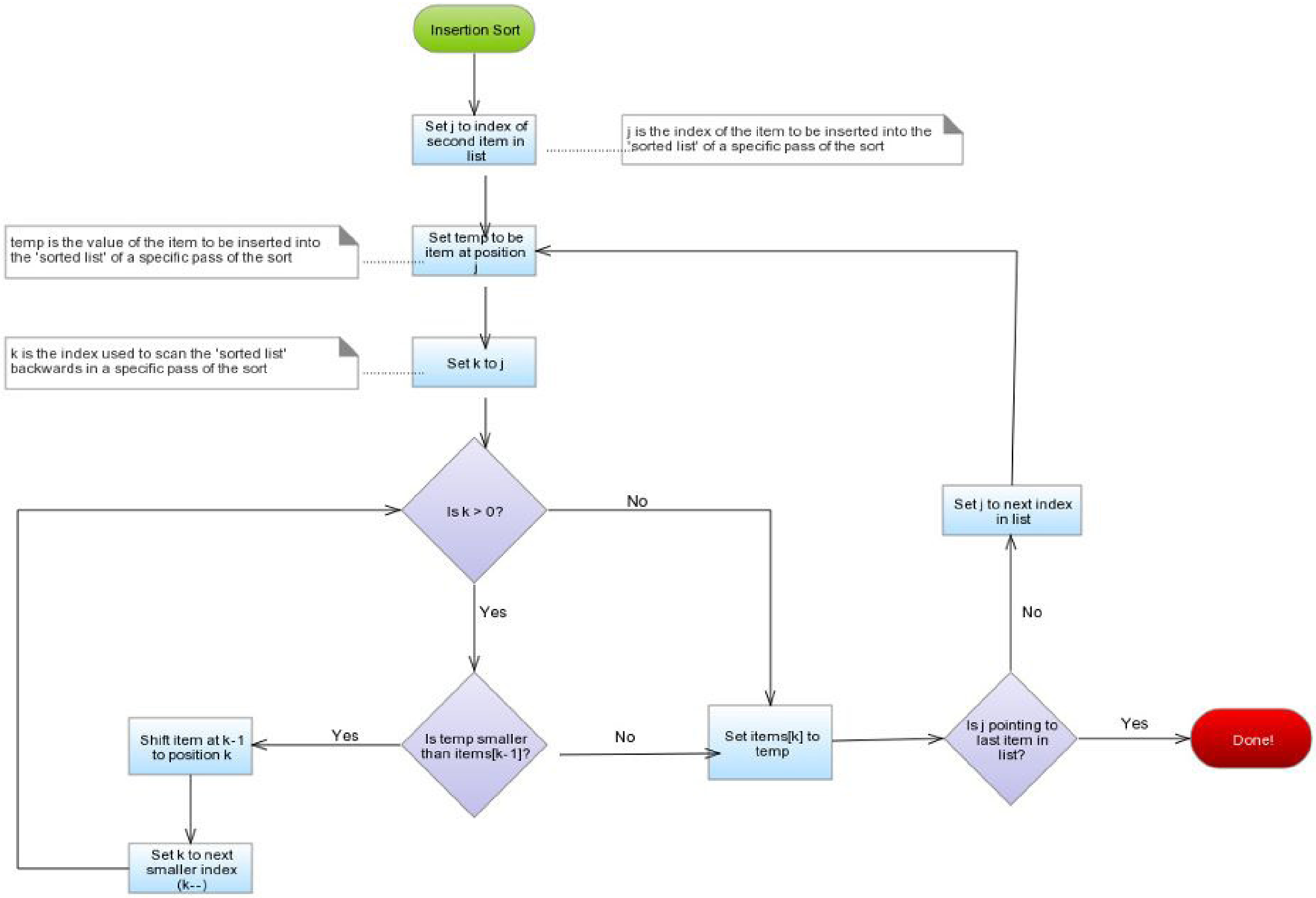
Selection Sort

```
//assume N is the number of items in the list
Procedure selectionSort(items):
      For j = 0 to N-1 do:
      minIndex = j
      For k = j + 1 to N-1 do:
              If items[k]<items[minIndex]
      minIndex = k
              End-If
          End-For
      temp = items[j]
      items[j] = items[minIndex]
      items[minIndex] = temp
      End-For
      End-Procedure
```


Selection Sort
Set j to index of first item in list
j is the index that the smallest item will be swapped to in a specific pass of the sort
Set minIndex to j
minIndex is the index of the current smallest item in a specific pass of the sort
k is the index used to scan the subset of the list in a specific pass of the sort
Set k to next index in list
No
Set minIndex to k
Yes
Is items[k] smaller than items[minIndex]?
No
Is k pointing to last item in list?
Yes
Swap items[minIndex] with items[j]
Set j to next index in list
No
Is j pointing to last item in list?
Done!

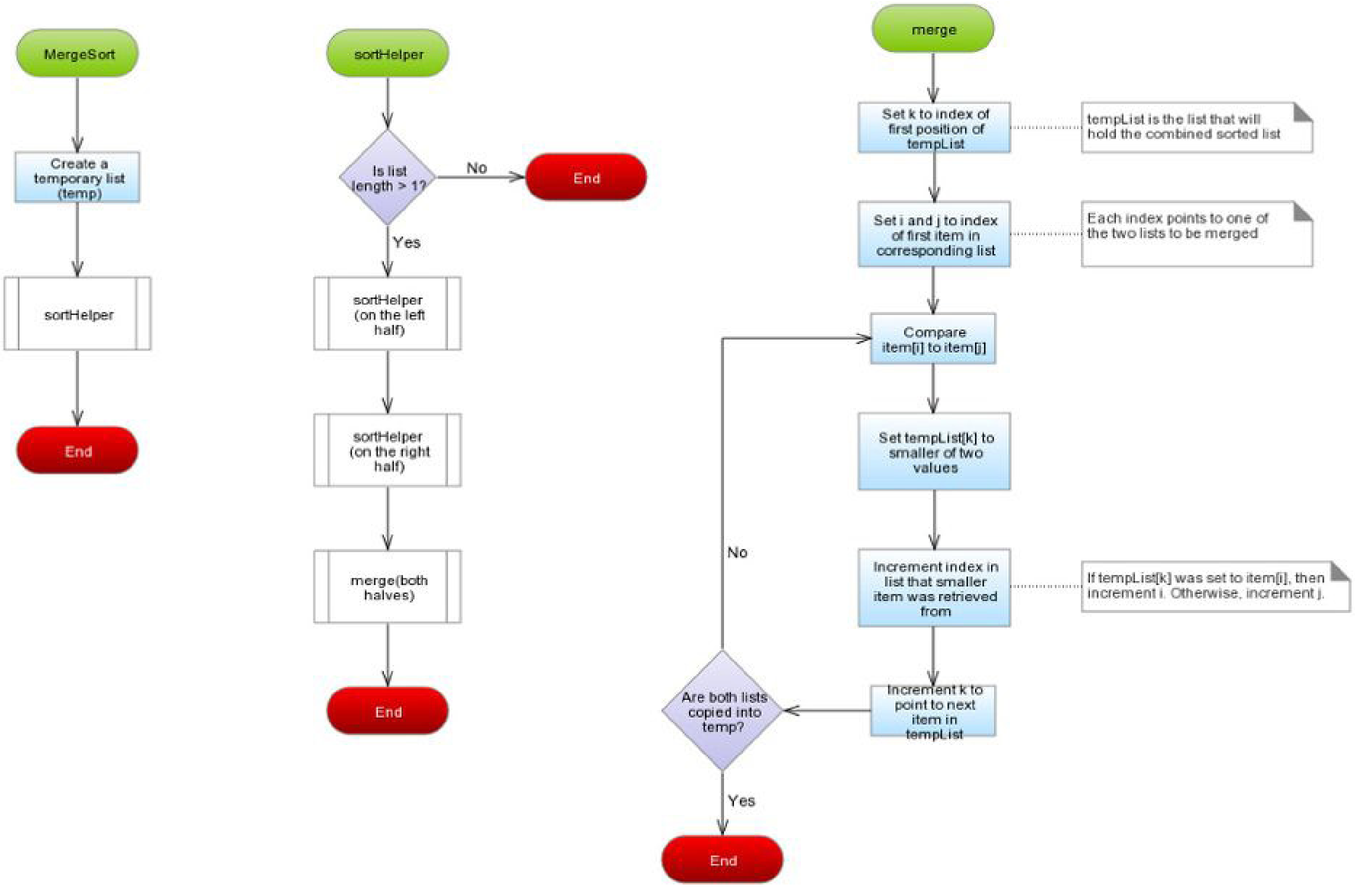
Insertion Sort

```
//assume N is the number of items in the list
Procedure insertionSort(items):
      For j = 1 to N-1
      k = j
          Do while (k>\0) and (items[k]<\items(k - 1)
      temp = items[k]
      items[k] = items[k – 1]
      items[k– 1] = temp
      k = k - 1
          End-Do
      End-For
End-Procedure
```


Insertion Sort
Set j to index of second item in list
j is the index of the item to be inserted into the 'sorted list' of a specific pass of the sort
Set temp to be item at position j
temp is the value of the item to be inserted into the 'sorted list' of a specific pass of the sort
Set k to j
k is the index used to scan the 'sorted list' backwards in a specific pass of the sort
Is k > 0?
No
Yes
Set j to next index in list
No
Shift item at k-1 to position k
Yes
Is temp smaller than items[k-1]?
No
Set items[k] to temp
Is j pointing to last item in list?
Yes
Done!
Set k to next smaller index (k--)

Merge Sort

```
//temp is a temporary list that will be used during the sort
Procedure mergeSort(items):
        If (length(items) > 1)
            L1 = List of elements in first half of items
            L2 = List of elements in second half of items
            mergeSort(L1)
            mergeSort(L2)
            merge(L1, L2)
        End-If
End-Procedure

//i and j traverse through the two lists
//k is the index to insert an item into the merged list
Procedure merge(L1, L2):
        If length(L1) < length(L2)
            N = length(L1)
        Else
            N = length(L2)
        End-If

        i = 0
        j = 0
        L = new list to contain the merged list
        For k = 0 to N-1
            If L1(i) < L2(j)
                L[k] = L1(i)
                i++
            Else
                L[k] = L2(j)
                j++
            End-If
        End-For

        //add any items left in L1
        While i < length(L1)
            L[k] = L1[i]
            i++
            k++
        End-While

        //add any items left in L2
        While j < length(L2)
            L[k] = L2[j]
            j++
            k++
        End-While

End-Procedure
```


MergeSort
Create a temporary list (temp)
sortHelper
End
sortHelper
Is list length > 1?
No
End
Yes
sortHelper (on the left half)
sortHelper (on the right half)
merge(both halves)
End
merge
Set k to index of first position of tempList
tempList is the list that will hold the combined sorted list
Set i and j to index of first item in corresponding list
Each index points to one of the two lists to be merged
Compare item[i] to item[j]
Set tempList[k] to smaller of two values
Increment index in list that smaller item was retrieved from
If tempList[k] was set to item[i], then increment i. Otherwise, increment j.
Increment k to point to next item in tempList
Are both lists copied into temp?
No
Yes
End