Lab 6 Recursion—Part I

Goal

In this lab you will design and implement recursive algorithms. The primary focus in this lab will be algorithms that make a single recursive call. Tail recursion will be explored. An improved recursive algorithm for computing Fibonacci numbers will be developed.

Resources

- Chapter 7: Recursion
- `Lab6Graphs.pdf` – Full size versions of the blank graphs for this lab

Java Files

- *`Count.java`*
- *`RecursiveFactorial.java`*
- *`RecursiveFibonacci.java`*
- *`RecursiveStringReplace.java`*
- *`TestFactorial.java`*
- *`TestFibonacci.java`*
- *`TestReplace.java`*
- *`TimeFibonacci.java`*

Warning: There is a lot of material in the introduction and pre-lab for this lab. Make sure to start early to give yourself enough time to complete them before the lab.

Introduction

Recursion is an important technique that shows up in diverse areas of computer science, such as the definition of computer languages, the semantics of programs, the analysis of algorithms, data structure definitions, and algorithms. Some of the better algorithms are either recursive or based on recursive algorithms. Also, recursive algorithms are often very elegant and show an economy of expression.

Since some students struggle with recursion, this lab starts out with some simple applications to improve your familiarity. The improved version of Fibonacci at the end of the lab is a more complicated example that shows the true power of recursion. Applying yourself to this lab and the one after should get you comfortable with recursion. This will be especially useful when merge sort and quick sort are discussed and then again later when trees are presented. For students who want even more practice with recursion and functional programming, experiment with the language Lisp. It is one of the oldest computer languages and is still in use. Its original core was strictly recursive, though iterative constructs were quickly added. Programming in Lisp or Scheme (one of its derivatives) is an interesting way to practice recursive programming.

Recursion is closely related to iteration. With the appropriate data structures, any iterative algorithm can easily be turned into a recursive algorithm. Similarly, any recursive algorithm can be made iterative. Tail recursive algorithms can be easily converted into an iterative algorithm. (Good Lisp compilers will convert tail recursive code into a loop.) Other recursive algorithms can be converted into an iterative algorithm by using a stack. (See Lab 10 Explicit Stacks in Recursion for an example.) In some cases, finding an equivalent nonrecursive algorithm that does not use a stack can be quite challenging.

The basic idea of recursion is to solve a problem by first solving one or more smaller problems. Once this has been done, the solutions of the smaller problems are combined to form the solution to the original problem. The repetitive nature of recursion comes into play in that to solve the smaller problems, you first solve even smaller problems. You cannot defer the solution indefinitely. Eventually, some very small problem must be solved directly.

Recursive Design

There are five parts to designing a recursive algorithm.

Identify the problem: What are the name and arguments of the original problem to be solved?

Identify the smaller problems: What are the smaller problems that will be used to solve the original problem?

Identify how the answers are composed: Once the solutions to the smaller problems are in hand, how are they combined to get the answer to the original problem?

Identify the base cases: What are the smallest problems that must be solved directly? What are their solutions?

Compose the recursive definition: Combine the parts into a complete definition.

A recursive design for computing factorial will be used to illustrate the process. The standard recursive definition of factorial is well known (see the resources), so we will make a slight that will reduce the number of recursive calls made.

Identify the problem:
Factorial(n)

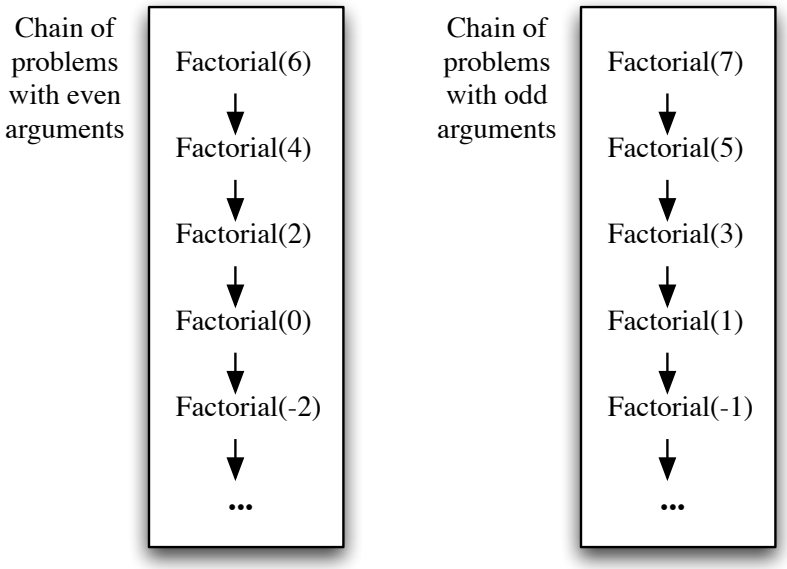
Identify the smaller problems:
(Reduce the problem size by two instead of one.)
Factorial(n – 2)

Identify how the answers are composed:
Factorial(n) = n * (n –1) * Factorial(n – 2)

Identify the base cases:
Certainly Factorial(1) is 1. But is this enough? Consider Factorial(6). Applying the recursion gives

Factorial(6) = 6*5* Factorial(4)
Factorial(6) = 6*5*4*3* Factorial(2)
Factorial(6) = 6*5*4*3*2*1* Factorial(0)
Factorial(6) = 6*5*4*3*2*1*0*(–1) Factorial(–2)
Factorial(6) = 6*5*4*3*2*1*0*(–1) *(–2) *(–3) Factorial(–4)
...

Clearly, this recursion has two chains of problems, odds and evens.



Both chains must have a base case. An appropriate question is "Where in the even chain should the recursion stop?" Looking at the last two expansions in the recursion, you see that the resulting product will be zero, which is not the correct result. No negative n is suitable for the base case. This just leaves the question of whether the base case for the even chain should be Factorial(2) or Factorial(0). If Factorial(0) is the base case, what is its value? The value that makes the recursive definition work is 1.

Factorial(0) = 1
Factorial(1) = 1

Compose the recursive definition:

Factorial(n) =	1	if n=0 or n=1
	n * (n–1) * Factorial(n–2)	if n > 1

From here you can write down the code:

```
int factorial(int n)
{
    int result;
    if( n < 2)
       result = 1;
    else
       result = n *(n–1)*factorial(n–2);
    return result;
}
```

An alternate version can be created by recognizing that the base cases are simple and can be folded into the initialization.

```
int factorial(int n)
{
    int result = 1;
    if( n >= 2)
       result = n *(n–1)*factorial(n–2);
    return result;
}
```

Recursion with Structures

A list is a data structure that is an ordered collection of items. It will be discussed in more detail in Chapter 12. It is interesting because this was the basic data structure that Lisp used originally and it was recursively defined. It was basically a linked structure composed of "cons" pairs. The first part of the cons is a list item value and the second is a link to the rest of the list (another cons pair). From this definition, there are two obvious methods that can be used on a list. The first method (originally "car" in lisp) returns the first item of the list. The rest method (originally "cdr" in lisp) returns the rest of list not including the first item. These correspond to the two parts of the cons cell. To build a list we can use the cons method that builds a cons pair out of an item and a list. The base case of the recursive definition is an empy list represented by nil.

Identify the problem:

Reverse(L)

Identify the smaller problems:

To reduce the size of the problem, some function of the arguments must decrease. In this case, the list L must be reduced in size. Consider the following instance of reverse:

Reverse({ 1 2 3 4 5 }) = { 5 4 3 2 1 }

We can reduce the problem by removing the first item and reversing what is left.

Reverse({ 2 3 4 5 }) = { 5 4 3 2 }

This can clearly be used to get the solution to the original problem. So in this case the smaller problem is

Reverse(rest(L))

Identify how the answers are composed:

From the example, it is clear that the first item in L must be pasted onto the end of the solution to the smaller problem.

Reverse(L) = append(Reverse(rest(L)), first(L))

Identify the base cases:

An empty list is the smallest possible list and is represented by nil. Reversing an empty list results in an empty list.

Reverse(nil) = nil

Compose the recursive definition:

Reverse(L) =	nil	if L is nil
	append(Reverse(rest(L)), first(L))	if L is not nil

Code is but a short step away.

```
<T> List<T> reverse(List<T> ls)
{
    List<T> result = new ArrayList<T>();
    if( ls.size() > 0)
    {
        result =reverse( ls.subList(1,ls.size() ));
        result.add(ls.get(0));
    }
    return result;
}
```

The recursive definition used the operations append, first, and rest. In the code, the type of the argument is the built-in `List` class from Java. The implementation must use appropriate methods from `List` to accomplish the three operations. Since only a single character is being appended, the `add()` method can be used to place it at the end of the list. Since the first item is in location 0 of the list, get`(0)` will extract the first value of the list. The `subList()` method allows one to get a range of items from the list. By starting at 1, everything except the first value will be in the sublist.

There is one more thing to note about this implementation. It is functional in the sense that its argument is unchanged by the method. It returns a new list, which is the reverse of the argument. While this works well for the abstract definition of a list, is it equally good for an array representation of a list? Each of the operations (append, first, and rest) can be implemented, but they will not be very efficient. Often for an array, it is desired that the array be changed in place instead of creating a new array with the result.

The following redesign of the problem has the constraint that the items to be reversed are stored in an array and it is desired that the reverse be done in place.

Identify the problem:

Reverse(A)

Looking ahead, there is a problem. The arguments must decrease in some fashion, yet the array will remain a constant size. What is decreasing is not the array, but the portion of the array that the recursion is working on. An auxiliary method that does the actual recursion is required.

ReverseAux(A, start, end)

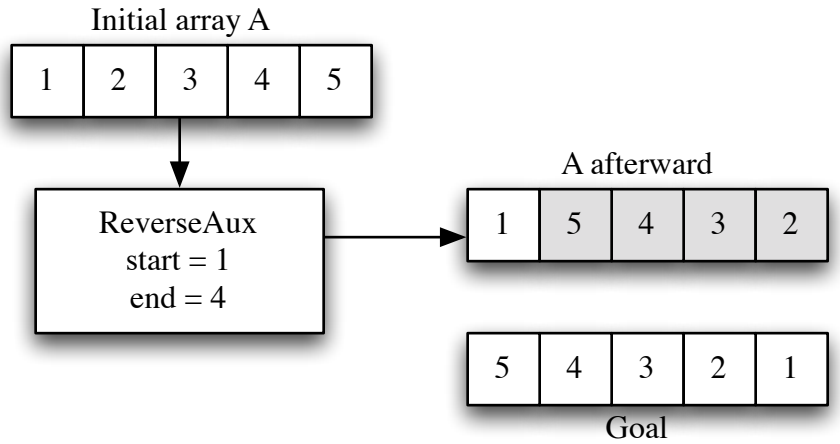
Identify the smaller problems:

The portion of the array that is considered must be reduced in size. Consider the following instance of reverse:

Initially A is [1 2 3 4 5].
After ReverseAux ([1 2 3 4 5], 0, 4), A is [5 4 3 2 1].

As opposed to Reverse, ReverseAux does not return anything but has a side effect.

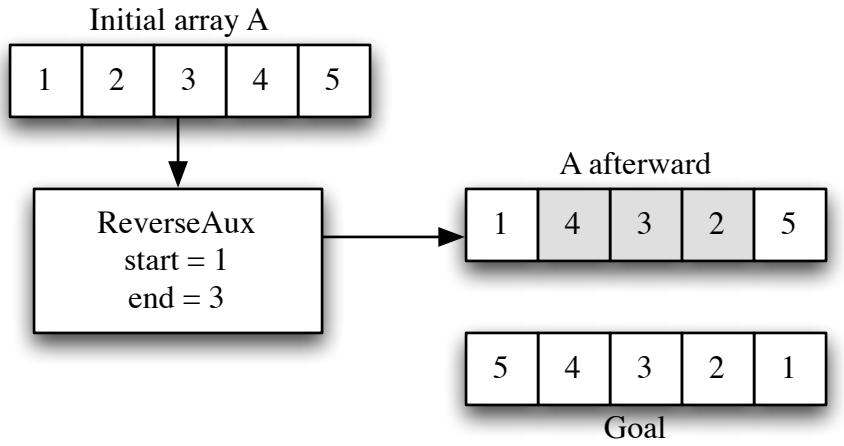
Suppose the same reduction in problem size is used as before (reverse the rest of the array).



How can you get from { 1 5 4 3 2 } to { 5 4 3 2 1 }? While it is possible, it requires that every data value be moved. Trying to use ReverseAux ({ 1 2 3 4 5 }, 0, 3) results in the same kind of difficulty.

One solution is to reduce the portion of the array being worked on by moving both ends inward.

ReverseAux(A, start+1, end–1)



To get the desired result, all that remains is to swap the first and last entries.

Identify how the answers are composed:

ReverseAux(A, start, end) is

1. ReverseAux(A, start+1, end–1)
2. swap(A[start], A[end]);

Identify the base cases:
Since the reduction is by two, we have two chains of recursive method calls, one each for arrays with odd and even numbers of values to be reversed. If start is the same as end, there is one value to be reversed in place. If start is less than end there is more than one value to be reversed. What if start is greater than end? It is convenient to let this represent the situation where there are no values to be reversed.

If the portion of array to be reversed is empty or contains a single entry, the reverse is the same as the original and nothing needs to be done.

ReverseAux(A, x, y) where x >= y
is
1. Do nothing.

Compose the recursive definition:

Reverse(A) = ReverseAux(A, 0, A.length - 1)

ReverseAux(A, start, end) is

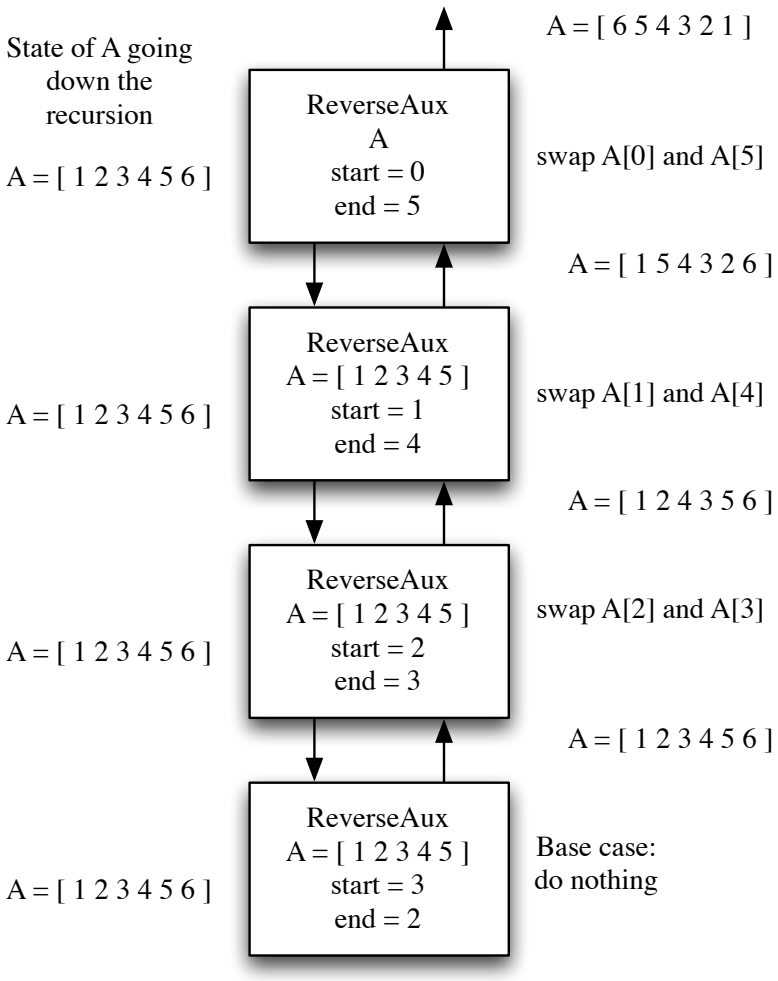
1. Do Nothing. if start >= end

or

1. ReverseAux(A, start+1, end–1) if start < end
2. swap(A[start], A[end])

Tail Recursion

The composition work that a recursive algorithm does can either be performed before or after the recursive call. Here is a trace of the calls that ReverseAux does on a list with six entries.



In this case the swap is done after the recursive call and all the work is done on the way back up the chain.

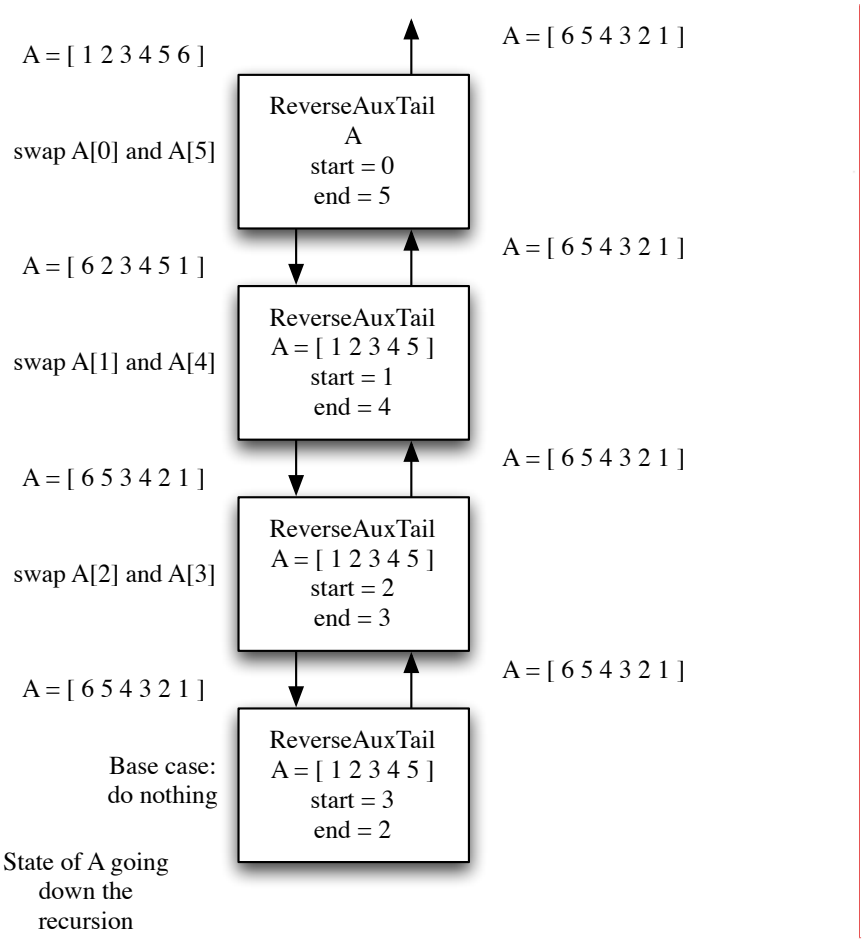
In a tail recursion, all the work is done on the way down the chain. Suppose that the definition is modified to become

```
Reverse(A) =  ReverseAuxTail(A, 0, A.length - 1)

ReverseAuxTail (A, start, end)   is

        1. Do Nothing.                          if start >= end
or
        1. swap(A[start], A[end])               if start < end
        2. ReverseAuxTail (A, start+1, end–1)
```

Here is a trace of the calls that ReverseAuxTail does on a list with six entries.



The tail recursive method is composing the answer as it goes down the chain of recursive calls. Once it reaches the base case, all the work has been done and it can return immediately. Also notice that once the next method in the chain of recursive calls is invoked, the variables start and end are no longer needed. This means that you can use an iteration that just has one copy of start and end.

In the version that is not tail recursive, start and end cannot be discarded until after the swap is completed. Therefore, it must have multiple pairs of start and end, one for each recursive call. These values will be stored on the system stack.

Often a tail recursive method will have an argument whose purpose is to store a partial solution as it is being composed. This can be illustrated by revisiting reverse on a list. Remember that the composition step was

Reverse(L) = append(Reverse(rest(L)), first(L))

In this case, the rest operation will be performed on the way down the chain of recursive calls, but the append is done on the way back up. Unlike reverse with an array, it is not simply a matter of moving statements around. The solution is to add another variable, which will hold the partial solution.

Identify the problem:

Reverse(L)
ReverseAuxTail(L, partial)

Identify the smaller problems:

Again, the size of the list will be reduced by using the rest operation. Consider the following instance of reverse:

ReverseAuxTail ({ 1 2 3 4 5 }, $partial_0$) = { 5 4 3 2 1 }

It is not clear what partial is yet, but the next call will be

ReverseAuxTail ({ 2 3 4 5 }, $partial_1$) = { 5 4 3 2 1}

Remember that the result of the final recursive call will be the final solution, so all tail recursive calls will return the solution. Continuing on,

ReverseAuxTail ({ 3 4 5 }, $partial_2$) = { 5 4 3 2 1}
ReverseAuxTail ({ 4 5 }, $partial_3$) = { 5 4 3 2 1}
ReverseAuxTail ({ 5 }, $partial_4$) = { 5 4 3 2 1}
ReverseAuxTail ({ }, $partial_5$) = { 5 4 3 2 1}

Each entry must be added to the partial solution. Looking at the second to last call, the value 5 must be prepended to the front of the partial solution. (In Lisp, we can easily create a new list with a value at the front using the cons method. If fact, adding a value at the end of a list is much more complicated.) The smaller problem is therefore:

ReverseAuxTail (tail(L), prepend(first(L), partial))

Identify how the answers are composed:

In a tail recursion, all the work in determining how to compose the final solution from the smaller problem is done in identifying the smaller problem.

ReverseAuxTail (L, partial) =
    ReverseAuxTail (tail(L), prepend(first(L), partial))

Identify the base cases:

An empty list is still the smallest possible list. It will be represented by nil. In this case, though nil is not returned. When the base case is reached the partial solution is now a complete solution and will be returned.

ReverseAuxTail (nil, partial) = partial

Compose the recursive definition:

There is one remaining piece of business. What should the initial partial solution be? Since each of the values will be prepended to it one by one, the only possible choice is nil (an empty list).

Reverse(L) = ReverseAuxTail (L, nil)

ReverseAuxTail (L, partial) =

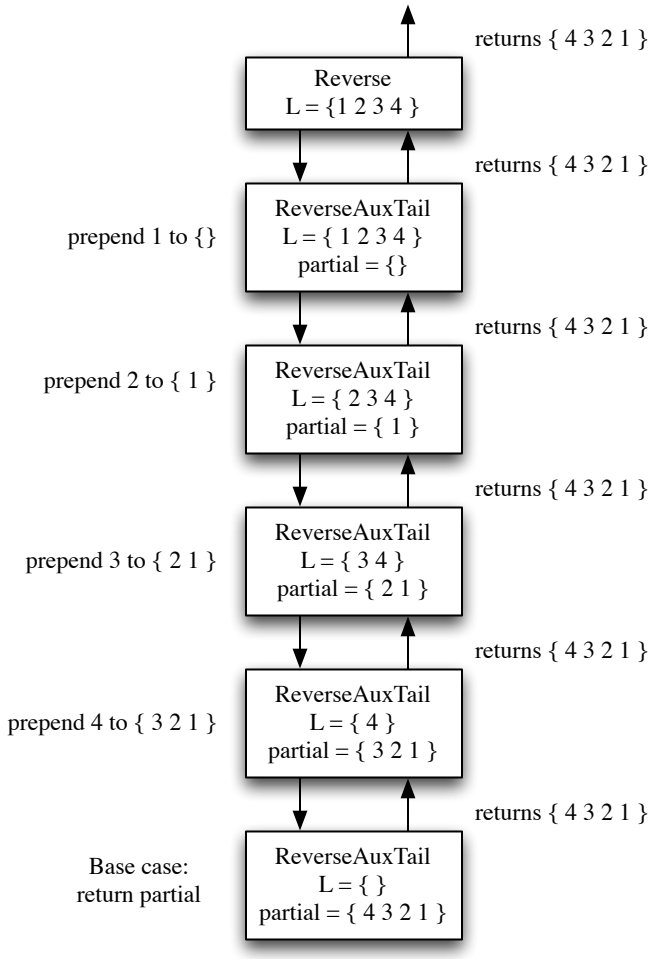
partial	if L is nil
ReverseAuxTail (rest(L), prepend(first(L), partial))	if L is not nil

Here is the code.

```
<T> List<T> reverse(List<T> ls)
{
    return reverseAuxTail(ls, new ArrayList<T>());
}

<T> List<T> reverseAuxTail(List<T> ls, List<T> partial)
{
    if ( ls.size() == 0)
        return partial;
    else
    {
        partial.add(0, ls.get(0));
        return reverseAuxTail(ls.subList(1,ls.size()), partial);
    }
}
```

The following diagram shows a trace of the method.



Double Recursion

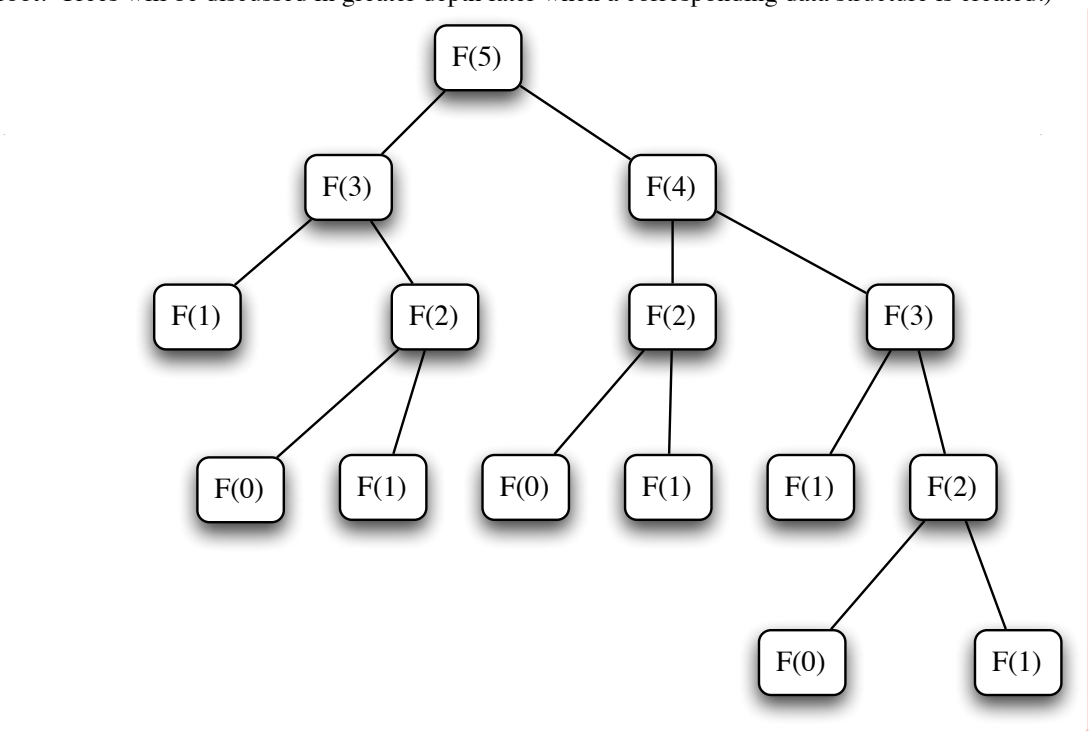
All of the recursive algorithms presented so far have a simple chain of recursive calls. Each recursive call in turn makes a single invocation of a smaller recursive problem. A number of recursive algorithms can make two or more recursive invocations. The classic example of a double recursion is the standard recursive definition of the sequence of Fibonacci numbers

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

Each is the sum of the previous two numbers in the sequence. The recursive definition is

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-2) + F(n-1) & \text{if } n > 1 \end{cases}$$

Here is the pattern of recursive calls made for the F(5) tree. (A tree is a mathematical structure composed of vertices and edges. Each call is a vertex in the tree. Edges represent a method invocation. The root of the tree is at the top of the diagram and it grows down. The height of the tree is the length of the longest path from the root. Trees will be discussed in greater depth later when a corresponding data structure is created.)

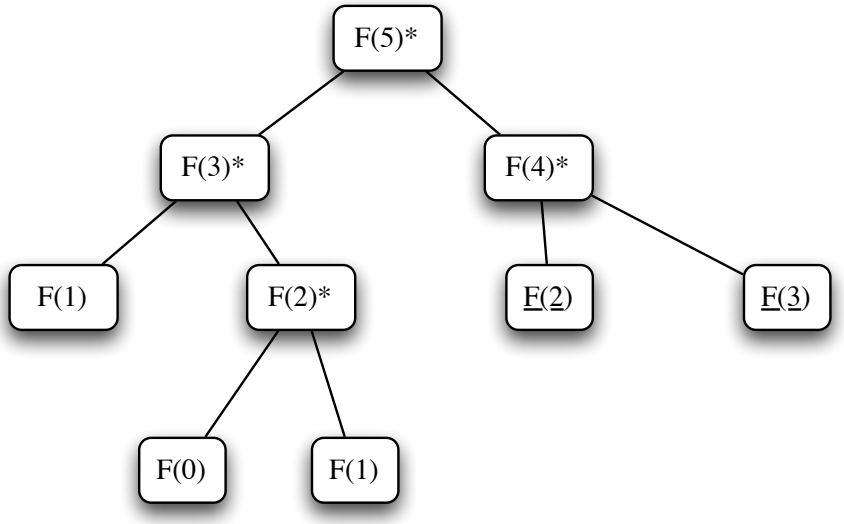


The problem with this kind of recursion is that the number of recursive calls made can grow exponentially as the tree of problems gets taller. Though it takes a bit of analysis to show, the number of invocations for this recursive definition of Fibonacci numbers is exponential in *n*.

One way of dealing with the exponential growth is to guarantee that the height of the tree grows slowly as the problem size increases. To accomplish this requires that the size of the problem reduce quickly as you go down a branch in the tree. The merge sort algorithm, which will be presented later, guarantees this by solving two subproblems each of which is half the size of the original. Halving the size of the subproblems limits the height of the tree to $\log_2 n$.

The other way of dealing with the problem is to look for many instances of the same smaller problem. In those cases we can try two approaches. Memoization stores the results to problems when they are encountered for the first time. The next time a problem is seen, the result is just retrieved. The pattern for a memoized Fibonacci is

shown next. An asterisk indicates the first evaluation. An underline indicates second evaluations. Base cases are just evaluated normally.



The other technique for dealing with this problem is to iteratively evaluate from small problems to larger problems. Note that for the Fibonacci sequence each number only depends on the previous two values, so you do not need to keep all the values. This results in the standard iterative algorithm for computing Fibonacci numbers.

Timing Programs

The different algorithms for computing Fibonacci numbers will be timed in this lab. This introduces a number of complications. To find the time, the following chunk of code will be used.

```
Calendar start = Calendar.getInstance();

// The Code being timed goes here

Calendar end  = Calendar.getInstance();
long diff     = end.getTime().getTime() -
                start.getTime().getTime();

System.out.println("Time to compute ... was "
             + diff + " milliseconds.");
```

Each time the `getInstance()` method is invoked, the current system time will be retrieved. That time is the number of milliseconds from some fixed date. One consequence of this is that the difference may be off by as much as a millisecond. Any time of that order is not to be trusted. With the speed of today's computers, some algorithms may complete well within that time frame. To address this issue, the code that is being timed may be inside a `for` loop that will execute the code multiple times. The reported time will be divided by the number of times the loop executed. This, of course, will introduce an extra bit of time for the loop overhead, but it is assumed that this will be small and can therefore be ignored.

Our timing difficulties are further complicated by the fact that the code being timed may not have been running the whole time. The Java Runtime Environment (JRE) is not the only program being executed. As the load on the computer changes, the amount of time the program gets will change as well. Running the same timing code with the same parameters will not give you the same result. You hope that the results are within 10%, but there is no guarantee. Another complicating factor is that the JRE is threaded. (Multiple tasks can be running each in their own thread within the JRE.) Some development environments will have threads running that will compete with your program's thread for time.

Another issue is that as computers get faster, the time required for the execution of an implementation of an algorithm will decrease. This presents some problems in the instructions for the labs. An appropriate number of times to execute a loop today may be insufficient tomorrow. Two strategies have been used to ameliorate these problems.

The first strategy guarantees that enough iterations are done to get a reasonable execution time (usually on the order of a minute or so). The code is timed once for a fixed number of iterations. That time is then used to determine the number of iterations to use for the subsequent tests.

The second strategy addresses how to plot the times in a graph. Instead of plotting the actual time, a ratio is plotted instead. The ratio will be the actual time divided by a baseline time (usually the time for the smallest input). While the times themselves will vary from computer to computer, the ratios should be fairly stable.

Pre-Lab Visualization

Count

To start, consider a very simple recursion that does not do anything but displays all the integer values from 1 to n. Each call of the recursion will be for a different value of count and has the responsibility of printing that value. The original call will be for the largest value to be printed.

First consider the problem of counting from 1 up to n. The smallest value that should be printed is 1. When doing the recursive design, think about whether the display should be done on the way down the recursion chain or on the way back up.

 **Identify the problem**:

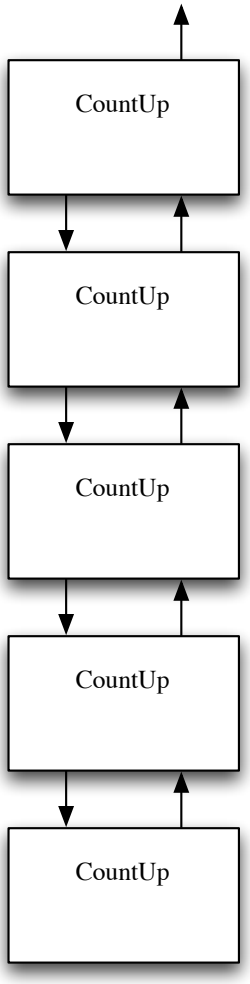
 **Identify the smaller problems**:

 **Identify how the answers are composed**:

 **Identify the base cases**:

 **Compose the recursive definition**:

Show the operation of your definition on the number 4 in the following diagram. Inside the boxes, show the values of the arguments passed into the method. On the left-hand side, show the operations done before the recursive call by the method. On the right-hand side, show operations done after the recursive call.



Now consider the problem of counting down. In this definition, the same values will be displayed, but in decreasing order. This should be very similar to the design of counting up.

 **Identify the problem**:

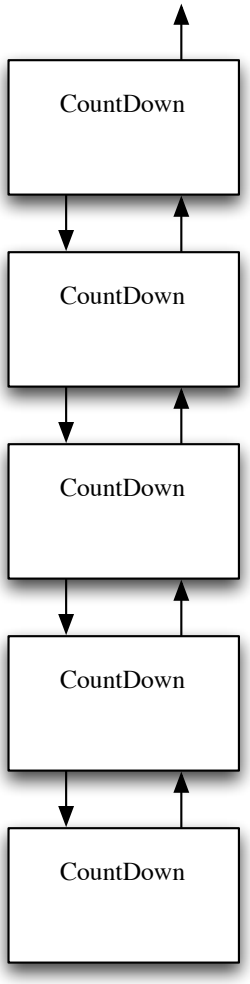
 **Identify the smaller problems**:

 **Identify how the answers are composed**:

 **Identify the base cases**:

 **Compose the recursive definition**:

Show the operation of your definition on the number 4 in the following diagram. Inside the boxes, show the values of the arguments passed into the method. On the left-hand side, show the operations done before the recursive call by the method. On the right-hand side, show operations done after the recursive call.



String Replace

Consider the problem of taking a `String` object and replacing every `'a'` in the string with a `'b'`. In general, the actual characters will be parameters of the replace method. The first problem you run into is that a `String` is immutable. (Once a `String` object has been created, it cannot be changed.) . So unlike an array, where the replace operation can be done in place, an approach more akin to the recursive reverse on a list is needed.

Examine the methods of the `String` class and show how you would implement each of the following operations.

 **First**:

 **Rest**:

 **Append/Prepend**:

Using those opeations and consulting reverse as a model, complete the recursive design for replace on a string.

 **Identify the problem**:

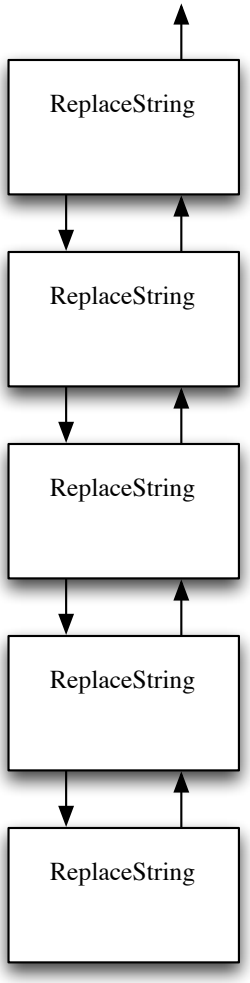
 **Identify the smaller problems**:

 **Identify how the answers are composed**:

 **Identify the base cases**:

 **Compose the recursive definition**:

Show the operation of your definition on the string `"abcb"` with `'b'` replaced by `'e'` in the following diagram. Inside the boxes, show the values of the arguments passed into the method. On the left-hand side, show the operations done before the recursive call. On the right-hand side, show operations done after the recursive call and indicate what value is returned.



Tail Recursive Factorial

As is common with tail recursive designs, an extra variable for the partial solution needs to be added. Factorial will call a helper method that will do the actual recursion. Think about what factorial is computing in conjunction with the value of n that is available as one goes down the recursion.

 **Identify the problem**:

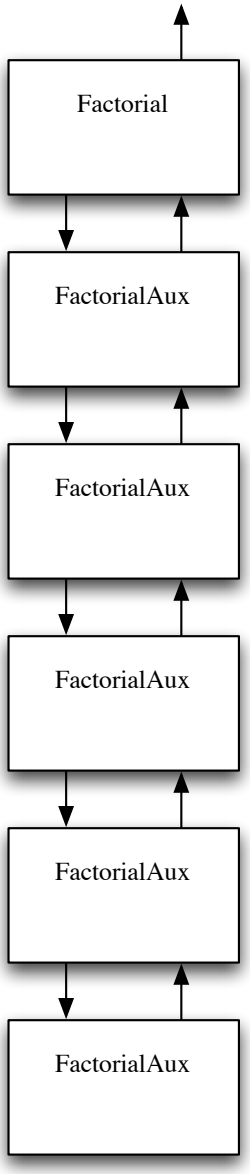
 **Identify the smaller problems**:

 **Identify how the answers are composed**:

 **Identify the base cases**:

 **Compose the recursive definition**:

Show the operation of your definition on the number 4 in the following diagram. In the boxes, show the values of the arguments passed into the method. On the left-hand side, show the operations done before the recursive call. On the right-hand side, show operations done after the recursive call and indicate what value is returned.



Improving Fibonacci

As was mentioned in the introduction, one way of controlling a double recursion is to limit the height of the recursion tree. The standard recursive definition for Fibonacci numbers only reduces the problem size by one and two. This results in a tree that has height *n*. To control the height, you must define the recursion in terms of much smaller problem sizes.

Consider the values of F(n) in the following table.

n	F(n)
0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21
9	34
10	55
11	89
12	144
13	233
14	377

If the value of F(2n) can be related to the value of F(n), the problem size will be reduced by half and the growth of the tree will be tamed. In the following table, write down the numerical relation between F(n) and F(2n).

n	F(n)	F(2n)	Relation between F(n) and F(2n)
1	1	1	
2	1	3	
3	2	8	
4	3	21	
5	5	55	
6	8	144	
7	13	377	

Perhaps there is a pattern here. Clearly, though F(2n) does not depend on just F(n). Since Fibonacci is double recursion, perhaps the values depend on values that neighbor F(n). In the following table, write down the numerical relation.

n	F(n-1)	F(n)	F(n+1)	F(2n)	Relation between F(n-1), F(n), F(n+1) and F(2n)
1	0	1	1	1	
2	1	1	2	3	
3	1	2	3	8	
4	2	3	5	21	
5	3	5	8	55	
6	5	8	13	144	
7	8	13	21	377	

What simple formula does this relation follow?

While the pattern of values we see in the tables is a good sign that there is a general relationship, by itself it is not concrete evidence. It is possible that the relation fails for larger values of n. A proof by mathematical induction of the discovered formula is required and can be done.

The F(n+1) term in the formula can be eliminated to produce

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

While this definition works for even values, what about odd values? Starting with the formula

$$F(2n+2) = F(2n+1) + F(2n)$$

one can derive

$$F(2n+1) = 2F(n)F(n) + 2F(n)F(n-1) + F(n-1)\,F(n-1)$$

This results in the recursive definition (**Height Limited Doubly Recursive**)

F(n)=	0	n=0
	1	n=1
	F^2 (n/2) + 2F(n/2)F(n/2–1)	n is even and > 1
	$2F^2$ (n/2) + 2F(n/2)F(n/2-1) + F^2 (n/2-1)	n is odd and > 1

Show the pattern of calls for F(17) and record the values produced by each call.

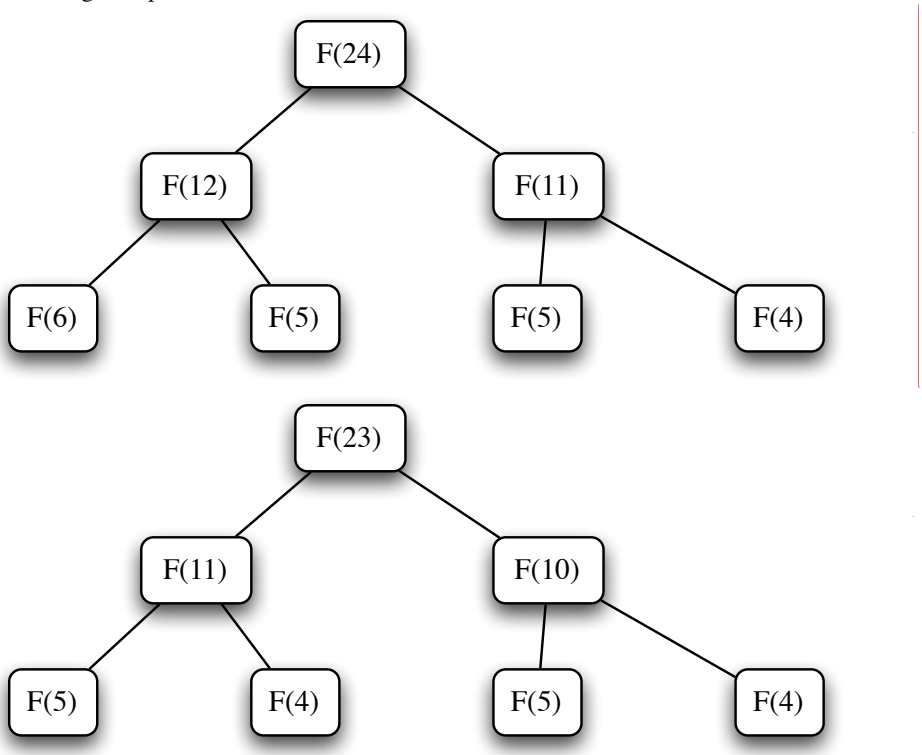
What is the height of the tree for F(n)?

How many recursive calls are made? Express your answer in big-O notation.

Tail Recursive Fibonacci

Note that the previous recursive formulation still has repeated subproblems. Therefore, memoization or an iterative formulation can improve the performance even more. We see, however, that not all possible subproblems need to be computed. To do an iterative solution from the bottom, the needed subproblems have to be identified.

Consider the following two partial trees.

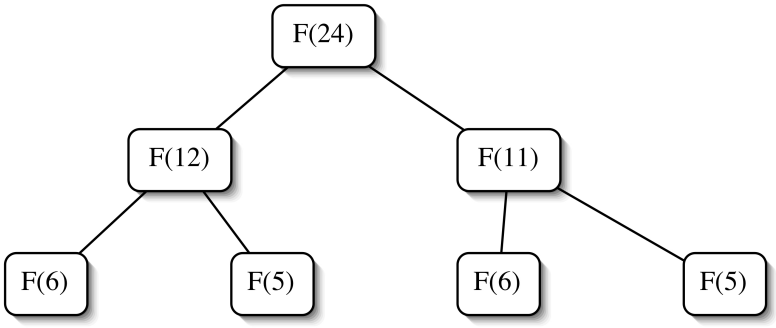


Looking at the third line, the first tree depends on three different problems, while the second only depends on two problems. The second tree is more desirable in that it has fewer problems it repeats. The first tree needs to be fixed so that only two values are needed at each level of the tree.

The problem arises when the larger of the two values in the second line is even. What is desired is a definition of F(2n–1) in terms of F(n) and F(n–1). Looking at the definitions for F(2n) and F(2n+1) and recalling that any Fibonacci number is the sum of the previous two, it is easy to derive the relations.

$$F(2n-1) = F(n)F(n) + F(n-1)F(n-1)$$
$$F(2n) = F(n)\,F(n) + 2F(n)F(n-1)$$
$$F(2n+1) = 2F(n)\,F(n) + 2F(n)F(n-1) + F(n-1)\,F(n-1)$$

Using the relation for F(2n–1) for F(11) in the first tree gives



Now each level in the tree will only have two values. The two values needed on the next level are determined by the n value on the level above. If it is even, use the formulas:

F(2n) = F(n)F(n) + 2F(n)F(n-1)
F(2n–1) = F(n)F(n) + F(n-1)F(n-1)

If it is odd, use the formulas

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

The only remaining problem is to determine which pairs of Fibonacci numbers will be computed for a given n.

Write down the pairs needed for n=50.

Suppose n is 163. Each of the pairs is recorded in the following table. The values will depend on the bits in 163. To the right is the bit pattern for 163. Circle the bit that is associated with each pair. Indicate which pair of formulas was used to get that row from the row below.

F(81)	F(80)	1 0 1 0 0 0 1 1	
F(40)	F(39)	1 0 1 0 0 0 1 1	
F(20)	F(19)	1 0 1 0 0 0 1 1	
F(10)	F(9)	1 0 1 0 0 0 1 1	
F(5)	F(4)	1 0 1 0 0 0 1 1	
F(2)	F(1)	1 0 1 0 0 0 1 1	
F(1)	F(0)	1 0 1 0 0 0 1 1	

What pair is always at the bottom?

Which bit determines the row for F(2) and F(1)?

If the determining bit is even, which pair of formulas is used?

It is now time to design the tail recursive method for Fibonacci numbers. Again, a tail recursive helper method will be used. This time, however, two partial solutions are required (one for each of the pairs of values). The bits in n will determine the pattern of values. Two extra methods will be needed. The first will get the second most significant bit (the bit to the right of the most significant bit) of a number n. The second will remove the second most significant bit from a number n.

Consider the number 5. It has the bit pattern 101_2. The second bit from the left is 0. Removing the 0 gives 11_2, which is 3.

What are the bit patterns for 96, 95, 16, 15, and 9?

Give an algorithm to find the second most significant bit in a number n.

Verify that it works on the bit patterns for 96, 95, 16, 15, and 9.

Give an algorithm to return the value found by removing the second most significant bit in a number n.

Verify that it works on the previous bit patterns.

Using the methods `secondMSB()` and `reduceBySecondMSB(),` design a tail recursive algorithm for Fibonacci numbers.

 **Identify the problem**:

 **Identify the smaller problems**:

 **Identify how the answers are composed**:

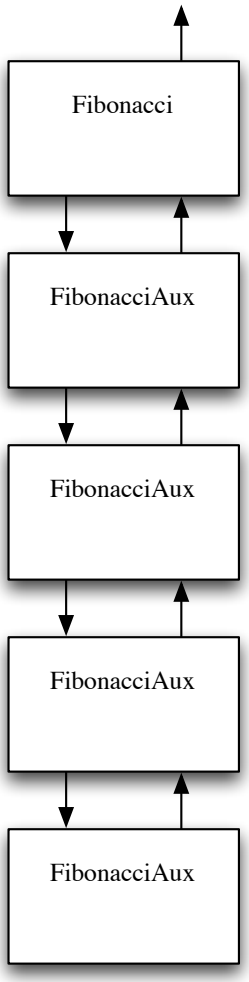
 **Identify the base cases**:

 **Compose the recursive definition**:

Does your definition work for n=0?

Does your definition work for n=1?

Show the operation of your definition on the number 14 in the following diagram. Inside the boxes, show the values of the arguments passed into the method. On the left-hand side, show the operations done before the recursive call by the method. On the right-hand side, show operations done after the recursive call and indicate what value is returned.



Directed Lab Work

Count

The first goal of this lab is to implement a couple of simple recursive methods that do not compute or return anything. Their sole purpose is to print integer values and give you some practice with recursion.

Step 1. Look at the skeleton in *`Count.java`*. Compile and run the `main` method in `Count`.

Checkpoint: The program will ask you for an integer value. Enter any value. A couple messages will be displayed, but no counting will happen.

Step 2. Refer to the count up recursive design from the pre-lab exercises. Complete the recursive method `countUp()`.

Checkpoint: Run Count. For the integer value enter 5. You should see 1 2 3 4 5.

Step 3. Refer to the count down recursive design from the pre-lab exercises. Complete the recursive method `countDown()`.

Final checkpoint: Run Count. For the integer value enter 5. You should see 5 4 3 2 1.

String Replace

The next goal is to complete a recursive method that will replace all occurrences of a given character with another character.

Step 1. Compile and run the `main` method in `TestReplace`.

Checkpoint: The program will run and get a null pointer exception.

Step 2. Refer to the string replace recursive design from the pre-lab exercises and complete the method `replace()` in *`RecursiveStringReplace.java`*..

Final Checkpoint: Compile and run TestReplace. All tests should pass.

Tail Recursive Factorial

The next goal is to complete a tail recursive helper method that will compute the factorial function.

Step 1. Compile and run the `main` method in `TestFactorial`.

Checkpoint: The program will run and fail most tests.

Step 2. Refer to the recursive design from the pre-lab exercises and complete the methods `tailRecursive()` and `helper()` in *`RecursiveFactorial.java`*.

Final Checkpoint: Compile and run TestFactorial. All tests should pass.

Timing Basic Fibonacci

The next goal is to see how long it takes to compute Fibonacci numbers using the basic recursive formulation.

Step 1. Compile `TimeFibonacci`.

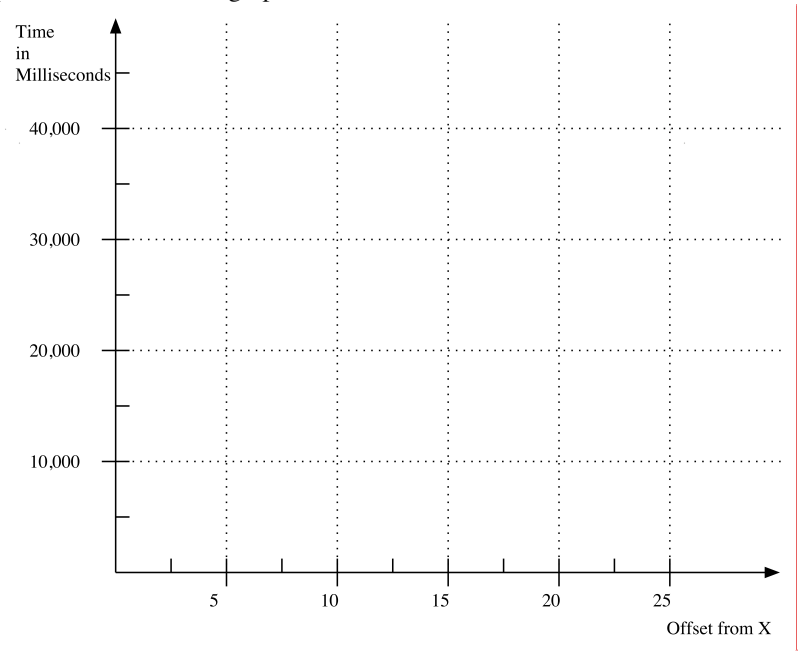
Step 2. Run the code for succesively larger even values of n and record the first value of n for which the time is greater than 100 milliseconds. (24 is a good place to start your search.)

FIRST EVEN VALUE OF N FOR WHICH THE TIME OF BASIC FIBONACCI IS GREATER THAN 100 MILLISECONDS	X =

Step 3. Fill in the values for *n* in the following table. Run the program and fill in the times. Stop timing when the time is longer than 100,000 milliseconds (about 2 minutes).

N	TIME IN MILLISECONDS TO COMPUTE F(N) USING THE BASIC FIBONACCI RECURSION
X =	
X+ 2 =	
X+ 4 =	
X+ 6 =	
X+ 8 =	
X+ 10 =	
X+ 12 =	
X+ 14 =	
X+ 16 =	
X+ 18 =	
X+ 20 =	
X+ 22 =	
X+ 24 =	
X+ 26 =	
X+ 28 =	
X+ 30 =	

Step 4. Plot points from the preceding table on the following graph. Don't worry about plotting the points that are off the graph.



Step 5. Draw a smooth curve that approximates the points.

A Better Version of Fibonacci

The next goal is to implement the better versions of fibonacci that were discovered in the pre-lab exercises. They will be timed.

Step 6. Refer to the pre-lab exercise and complete the implementation of the method `better()` in `RecursiveFibonacci`. (Use the height limited doubly recursive formula.)

Step 7. Compile `TestFibonacci`.

Checkpoint: Run TestFibonacci. All tests for the better Fibonacci formulation should pass.

A Tail Recursive Version of Fibonacci

Step 8. Refer to the recursive formulation from the tail recursion section on Fibonacci in the pre-lab exercises and complete the implementation of the method `secondMSB()` in `RecursiveFibonacci`. Don't forget to change the return statement.

Step 9. Refer to the recursive formulation from the tail recursion section on Fibonacci in the pre-lab exercises and complete the implementation of the method `reduceBy2ndMSB()` in `RecursiveFibonacci`.

Step 10. Test the two methods you just created.

Step 11. Refer to the recursive formulation from the tail recursion section on Fibonacci in the pre-lab exercises and create a LM recursive helper method in `RecursiveFibonacci` that uses `secondMSB()` and `reduceBy2ndMSB()`.

Step 12. Complete the method `tailRecursive()` in `RecursiveFibonacci` that calls the tail recursive helper method you created.

Step 13. Compile `TestFibonacci`.

Checkpoint: Run TestFibonacci. All tests for the both Fibonacci formulations should pass.

More Timing of Fibonacci

Step 14. Comment out the call to `timeBasic()` in `TimeFibonacci`.

Step 15. Uncomment the code to time the better and tail recursive versions of Fibonacci in `TimeFibonacci`.

Step 16. Run `TimeFibonacci`. Enter 100 for n and 100000 for the number of trials. Fill in the value in the table.

TIME IN MILLISECONDS TO COMPUTE F(100) USING THE BETTER RECURSIVE FORMULA	T =

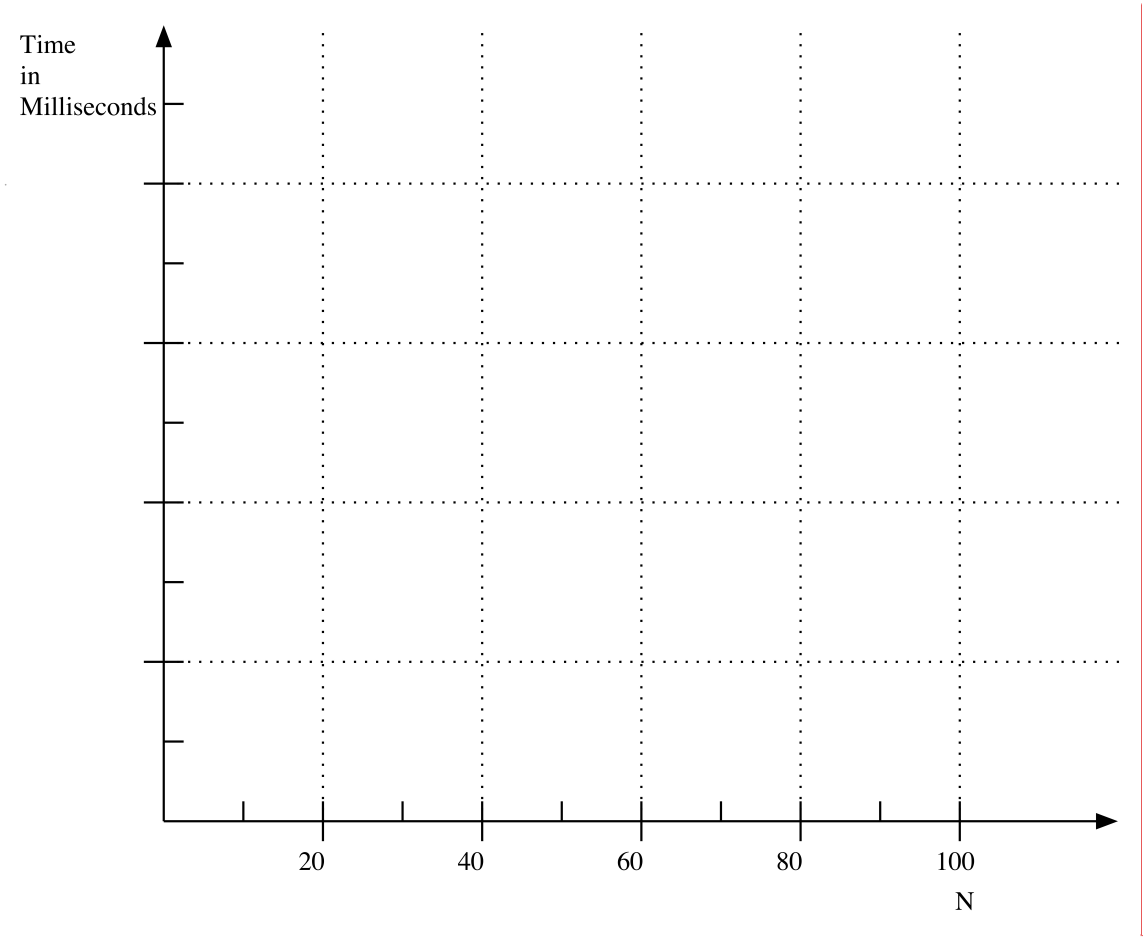
Step 17. Complete the following computation.

TRIALS = 10000 / T =

Step 18. Fill in the following table. Use the value of TRIALS from the previous step for the number of trials.

	TIME IN MILLISECONDS FOR BETTER FIBONACCI	TIME IN MILLISECONDS FOR TAIL RECURSIVE FIBONACCI
n=10		
n=20		
n=30		
n=40		
n=50		
n=60		
n=70		
n=80		
n=90		
n=100		

Step 19. Plot points (in different colors) for the times for two different versions of Fibonacci in the table. Put appropriate value labels on the *y*-axis of the graph.



Note that even though the better formulation allows computations for larger values of N in terms of time, the computed value also grows exponentially and is still problematic. The methods all use the long data type and will quickly overflow.

Post-Lab Follow-Ups

1. Develop a recursive algorithm for computing the product of a sequence of odd values. (Eg. ProdOdd(11) = 1*3*5*7*9*11.) Use that method to develop a recursive algorithm for factorial that splits the problem into a product of even values and a product of odd values.

2. Develop a recursive algorithm that given a and n, computes the sum
   $S = 1 + a + a^2 + ... + a^n$

3. Develop a recursive algorithm similar to string replace which works in place on an array of characters.

4. Develop a recursive algorithm for computing the second most significant bit of a number n.

5. Develop a recursive algorithm for computing the result of removing the second most significant bit from a number n.

6. Look at the ratio of the times for computing Fibonacci numbers F(n) and F(n+2) using the basic recursive formula. Given that you know F(X), predict the amount of time it would take to compute F(X+50). Predict the amount of time it would take to compute F(X+100).

7. Write a loop to compute successive values of F(n) using the tail recursive version. For what value of n does the computation overflow?