202: Computer Science II

Northern Virginia Community College

Recursion

Cody Narber, M.S.
September 23, 2017

Recursion Definition

Recusive definition is a definition in which something is defined in terms of smaller versions of itself. Consists of:

- Base Case (lowest level with not recursions)
- General/Recusive Case - solution is expressed in terms of a smaller version of itself

Example Factorial:

$$fac(x) = \begin{cases} 1 & x \leq 0 \\ x * fac(x-1) & x > 0 \end{cases} \tag{1}$$

Recursion Definition

Verifying solution:

1. Base Case Question
   - Is the base case correct for the values
2. Smaller-Caller Question
   - Does the recusive statement progress us closer towards the base case
3. General Case Question
   - Assuming the previous one resulted correctly, is the general correct?

$$fac(x) = \begin{cases} 1 & x \leq 0 \\ x * fac(x-1) & x > 0 \end{cases} \tag{2}$$

Spacing Recursion

```
public static void main(String[] args){
   int res = factorialSpc(3,"");
   System.out.println(res);
}
public static int factorialSpc(int n, String space){
  space+="  ";
  System.out.println(space+"Fac: "+n);
  if(n==1) {
   System.out.println(space+"return: 1");
   return 1;
  }
  int res = factorialSpc(n-1,space);
  System.out.println(space+"return: "+ (n*res));
  return n * res;
}
```

```
   Fac: 3
     Fac: 2
       Fac: 1
       return: 1
     return: 2
   return: 6
```

Method-Call Stack Recursion 1

```
1  //...
2  public static void main(String
       [] args){
3  >  int res = factorial(4);
4     System.out.println(res);
5  }
6  public static int factorial(
       int n){
7     if(n==1) return 1;
8     int res = factorial(n-1);
9     return n * res;
10 }
11 //...
```

Method-Call Stack

main(...)	
args	[]
cline	3

Method-Call Stack Recursion 2

```
 1 //...
 2 public static void main(String
       [] args){
 3    int res = factorial(4);
 4    System.out.println(res);
 5 }
 6 public static int factorial(
       int n){
 7    if(n==1) return 1;
 8 >  int res = factorial(n-1);
 9    return n * res;
10 }
11 //...
```

Method-Call Stack

factorial(...)	
n	4
cline	8

main(...)	
args	[]
cline	3

Method-Call Stack Recursion 3

```
1  //...
2  public static void main(String
       [] args){
3    int res = factorial(4);
4    System.out.println(res);
5  }
6  public static int factorial(
       int n){
7    if(n==1) return 1;
8  >  int res = factorial(n-1);
9    return n * res;
10 }
11 //...
```

Method-Call Stack

factorial(...)	
n	3
cline	8

factorial(...)	
n	4
cline	8

main(...)	
args	[]
cline	3

Method-Call Stack Recursion 4

```
 1 //...
 2 public static void main(String
        [] args){
 3    int res = factorial(4);
 4    System.out.println(res);
 5 }
 6 public static int factorial(
        int n){
 7    if(n==1) return 1;
 8 >  int res = factorial(n-1);
 9    return n * res;
10 }
11 //...
```

Method-Call Stack

factorial(...)	
n	2
cline	8

factorial(...)	
n	3
cline	8

factorial(...)	
n	4
cline	8

main(...)	
args	[]
cline	3

Method-Call Stack Recursion 5

```
 1 //...
 2 public static void main(String
       [] args){
 3    int res = factorial(4);
 4    System.out.println(res);
 5 }
 6 public static int factorial(
       int n){
 7 >  if(n==1) return 1;
 8    int res = factorial(n-1);
 9    return n * res;
10 }
11 //...
```

Method-Call Stack

factorial(...)	
n	1
cline	7

factorial(...)	
n	2
cline	8

factorial(...)	
n	3
cline	8

factorial(...)	
n	4
cline	8

main(...)	
args	[]
cline	3

Method-Call Stack Recursion 6

```
 1  //...
 2  public static void main(String
          [] args){
 3     int res = factorial(4);
 4     System.out.println(res);
 5  }
 6  public static int factorial(
          int n){
 7    if(n==1) return 1;
 8     int res = factorial(n-1);
 9  >  return n * res;
10  }
11  //...
```

Method-Call Stack

factorial(...)	
n	2
res	1
cline	8

factorial(...)	
n	3
cline	8

factorial(...)	
n	4
cline	8

main(...)	
args	[]
cline	3

Method-Call Stack Recursion 6

```
1  //...
2  public static void main(String
        [] args){
3     int res = factorial(4);
4     System.out.println(res);
5  }
6  public static int factorial(
        int n){
7    if(n==1) return 1;
8     int res = factorial(n-1);
9  >  return n * res;
10 }
11 //...
```

Method-Call Stack

factorial(...)	
n	3
res	2
cline	8

factorial(...)	
n	4
cline	8

main(...)	
args	[]
cline	3

Method-Call Stack Recursion 7

```
 1 //...
 2 public static void main(String
       [] args){
 3    int res = factorial(4);
 4    System.out.println(res);
 5 }
 6 public static int factorial(
       int n){
 7   if(n==1) return 1;
 8    int res = factorial(n-1);
 9 >  return n * res;
10 }
11 //...
```

Method-Call Stack

factorial(...)	
n	4
res	6
cline	8

main(...)	
args	[]
cline	3

Method-Call Stack Recursion 8

```
 1 //...
 2 public static void main(String [] args){
 3    int res = factorial(4);
 4 >  System.out.println(res);
 5 }
 6 public static int factorial(int n){
 7   if(n==1) return 1;
 8   int res = factorial(n-1);
 9   return n * res;
10 }
11 //...
```

Method-Call Stack

main(...)	
args	[]
res	24
cline	3

Direct vs Indirect Recursion

- **Direct:** Is when the recursive call is on the same method (a chain of size 1: A $\rightarrow$ A)

$$fac(x) = \begin{cases} 1 & x \leq 0 \\ x * fac(x-1) & x > 0 \end{cases} \quad (3)$$

- **Indirect:** Is when a method is called and eventually on the method-call stack is another reference back to the same method (a chain of size n: A $\rightarrow$ B $\rightarrow$ ... $\rightarrow$ A))

$$fac1(x) = \begin{cases} 1 & x \leq 1 \\ fac2(x) & x\%2 == 0 \\ x * fac2(x-1) & x\%2 == 1 \end{cases} \quad (4)$$

$$fac2(x) = \begin{cases} x * fac1(x-1) & x\%2 == 0 \\ fac1(x) & x\%2 == 1 \end{cases} \quad (5)$$

Adding a new node single linked-list - 1

Given a single linked list, we want to write an algorithm that can add it:

```java
public class ItemNode<T>{
     private T data;
     private ItemNode<T> next;
     public ItemNode(T data){
          this.data = data;
          next=null;
     }
     public ItemNode getNext(){return next;}
     public T getData(){return data;}
     public void setNext(ItemNode<T> n){next = n; }
}
```

```java
public class LinkList<T>{
     private ItemNode<T> first;
     public LinkList(){
          first=null;
     }
     /...
}
```

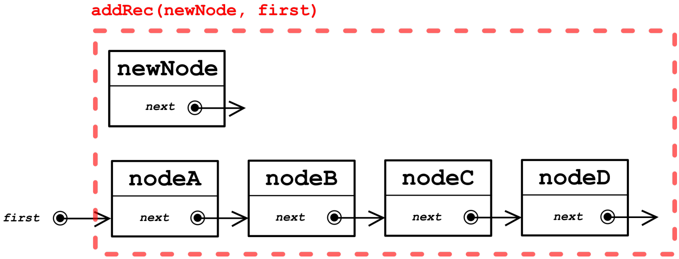
Adding a new node single linked-list - 2

Given a single linked list, write an algorithm to add a node:

```
public class LinkList<T>{
     private ItemNode<T> first;
     //...
     public void add(T data){
      if(first==null) first = new ItemNode(data);
         else addRec(new ItemNode(data), first);
     }
     private void addRec(ItemNode<T> nNode, ItemNode<T> cNode){
         if(cNode.getNext()==null) cNode.setNext(nNode);
         else addRec(nNode, cNode.getNext());
     }
     //...
}
```

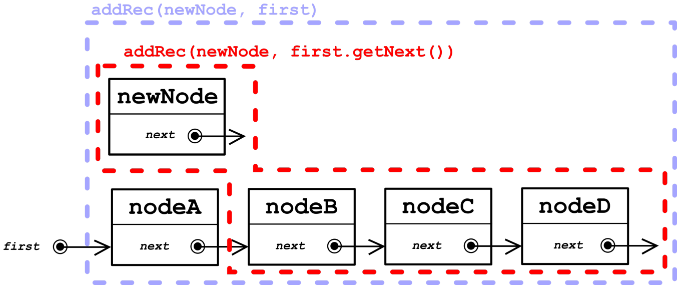
Adding a new node single linked-list - 3

```
//...
public void add(T data){
 if(first==null) first = new ItemNode(data);
    else addRec(new ItemNode(data), first);
}
private void addRec(ItemNode<T> nNode, ItemNode<T> cNode){
    if(cNode.getNext()==null) cNode.setNext(nNode);
    else addRec(nNode, cNode.getNext());
}
```



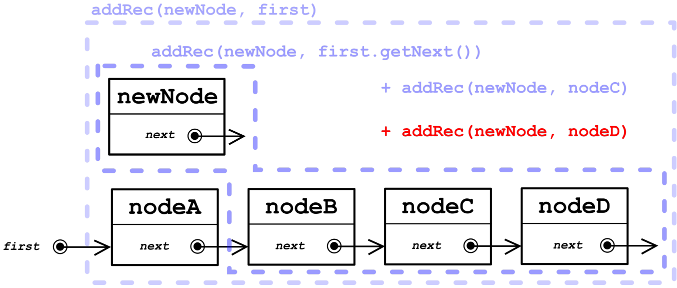
Adding a new node single linked-list - 4

```
//...
public void add(T data){
 if(first==null) first = new ItemNode(data);
    else addRec(new ItemNode(data), first);
}
private void addRec(ItemNode<T> nNode, ItemNode<T> cNode){
    if(cNode.getNext()==null) cNode.setNext(nNode);
    else addRec(nNode, cNode.getNext());
}
```



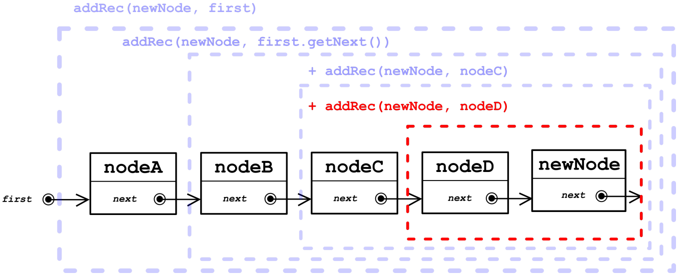
Adding a new node single linked-list - 5

```
//...
public void add(T data){
 if(first==null) first = new ItemNode(data);
    else addRec(new ItemNode(data), first);
}
private void addRec(ItemNode<T> nNode, ItemNode<T> cNode){
    if(cNode.getNext()==null) cNode.setNext(nNode);
    else addRec(nNode, cNode.getNext());
}
```



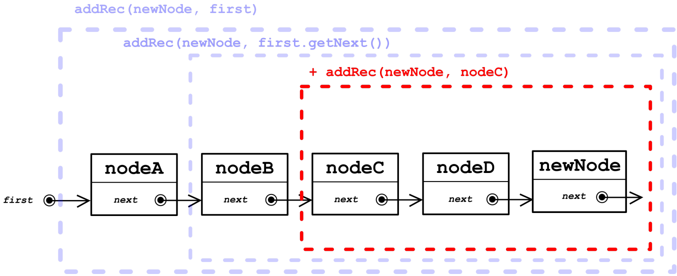
Adding a new node single linked-list - 6

```
    //...
    public void add(T data){
     if(first==null) first = new ItemNode(data);
        else addRec(new ItemNode(data), first);
    }
    private void addRec(ItemNode<T> nNode, ItemNode<T> cNode){
        if(cNode.getNext()==null) cNode.setNext(nNode);
        else addRec(nNode, cNode.getNext());
    }
```



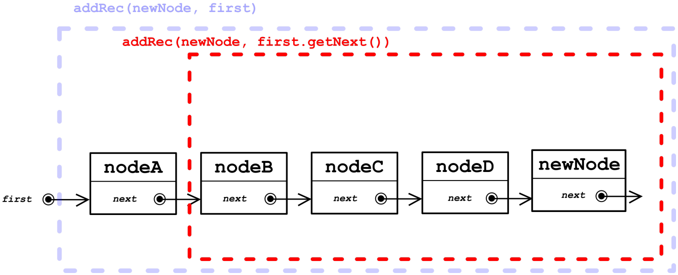
Adding a new node single linked-list - 7

```
//...
public void add(T data){
 if(first==null) first = new ItemNode(data);
    else addRec(new ItemNode(data), first);
}
private void addRec(ItemNode<T> nNode, ItemNode<T> cNode){
    if(cNode.getNext()==null) cNode.setNext(nNode);
    else addRec(nNode, cNode.getNext());
}
```



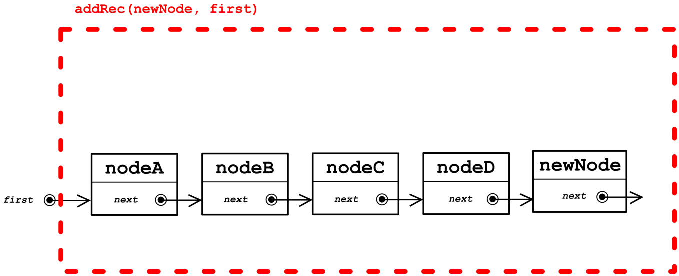
Adding a new node single linked-list - 8

```
//...
public void add(T data){
 if(first==null) first = new ItemNode(data);
    else addRec(new ItemNode(data), first);
}
private void addRec(ItemNode<T> nNode, ItemNode<T> cNode){
    if(cNode.getNext()==null) cNode.setNext(nNode);
    else addRec(nNode, cNode.getNext());
}
```



Adding a new node single linked-list - 9

```
//...
public void add(T data){
 if(first==null) first = new ItemNode(data);
    else addRec(new ItemNode(data), first);
}
private void addRec(ItemNode<T> nNode, ItemNode<T> cNode){
    if(cNode.getNext()==null) cNode.setNext(nNode);
    else addRec(nNode, cNode.getNext());
}
```



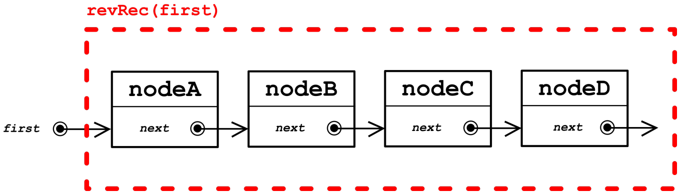
Reversing a single linked-list - 1

Given a single linked list, Write algorithms to print out the list in reverse or reverse the list.

```
//...
public void printRev(){printRev(first);}
public void printRev(ItemNode<T> cNode){
    if(cNode==null) return;
    printRev(cNode.getNext());
    System.out.println(cNode.getData();
}
public void reverse(){reverse(first);}
public ItemNode<T> reverse(ItemNode<T> cNode){
    if(cNode==null) return null;
    if(cNode.getNext()==null) return cNode;
    ItemNode<T> last = reverse(cNode.getNext());
    last.setNext(cNode);
}
```

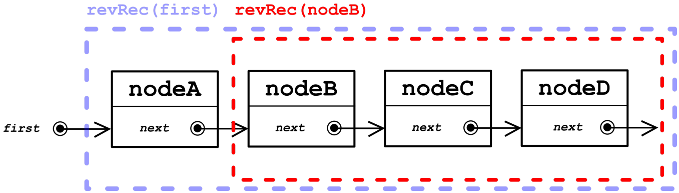
Reversing a single linked-list - 2

```
//...
public void reverse(){reverse(first);}
public ItemNode<T> reverse(ItemNode<T> cNode){
     if(cNode==null) return null;
     if(cNode.getNext()==null) return cNode;
     ItemNode<T> last = reverse(cNode.getNext());
     last.setNext(cNode);
}
```



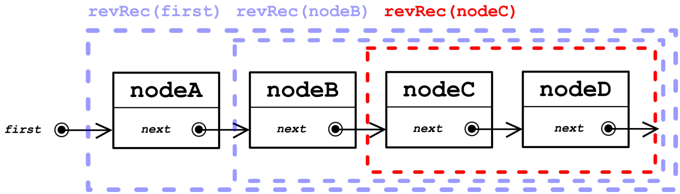
Reversing a single linked-list - 3

```
    //...
    public void reverse(){reverse(first);}
    public ItemNode<T> reverse(ItemNode<T> cNode){
        if(cNode==null) return null;
        if(cNode.getNext()==null) return cNode;
        ItemNode<T> last = reverse(cNode.getNext());
        last.setNext(cNode);
    }
```



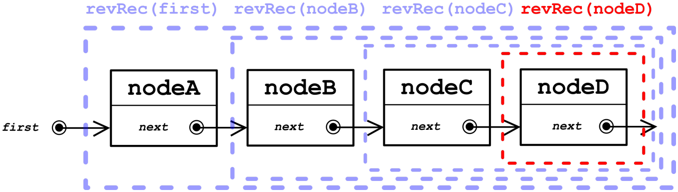
Reversing a single linked-list - 4

```
//...
public void reverse(){reverse(first);}
public ItemNode<T> reverse(ItemNode<T> cNode){
    if(cNode==null) return null;
    if(cNode.getNext()==null) return cNode;
    ItemNode<T> last = reverse(cNode.getNext());
    last.setNext(cNode);
}
```



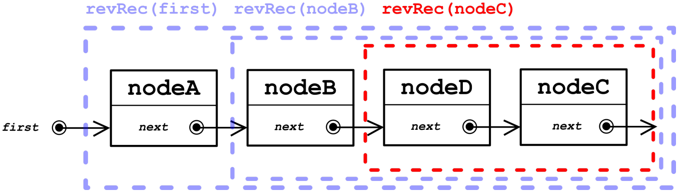
Reversing a single linked-list - 5

```
//...
public void reverse(){reverse(first);}
public ItemNode<T> reverse(ItemNode<T> cNode){
     if(cNode==null) return null;
     if(cNode.getNext()==null) return cNode;
     ItemNode<T> last = reverse(cNode.getNext());
     last.setNext(cNode);
}
```



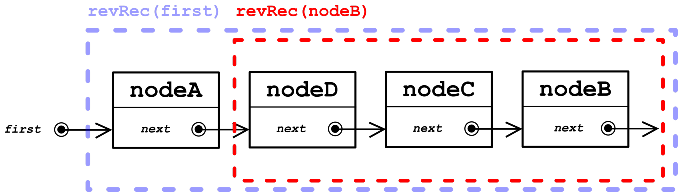
Reversing a single linked-list - 6

```
    //...
    public void reverse(){reverse(first);}
    public ItemNode<T> reverse(ItemNode<T> cNode){
        if(cNode==null) return null;
        if(cNode.getNext()==null) return cNode;
        ItemNode<T> last = reverse(cNode.getNext());
        last.setNext(cNode);
    }
```



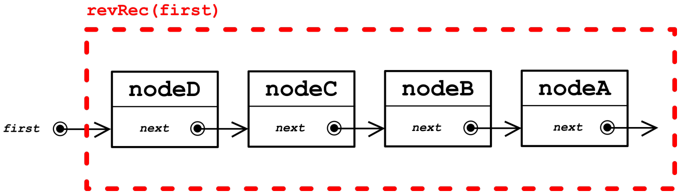
Reversing a single linked-list - 7

```
//...
public void reverse(){reverse(first);}
public ItemNode<T> reverse(ItemNode<T> cNode){
     if(cNode==null) return null;
     if(cNode.getNext()==null) return cNode;
     ItemNode<T> last = reverse(cNode.getNext());
     last.setNext(cNode);
}
```



Reversing a single linked-list - 8

```
//...
public void reverse(){reverse(first);}
public ItemNode<T> reverse(ItemNode<T> cNode){
     if(cNode==null) return null;
     if(cNode.getNext()==null) return cNode;
     ItemNode<T> last = reverse(cNode.getNext());
     last.setNext(cNode);
}
```



Towers of Hanoi

Given three pegs, with one peg starting with a set of any number of rings. The rings increase in size as they get lower. The goal is to move all rings to the opposite end, given that no larger ring can be placed ontop of a smaller ring. **Online code available to step through recursion process**

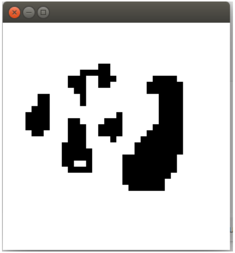
`moveRings(N, StartPeg, AuxPeg, EndPeg)`

- Base Case: N == 1. Move Ring to EndPeg
- General Case:
  - `moveRings(N-1, StartPeg, EndPeg, AuxPeg)`
  - Move Last Ring to EndPeg
  - `moveRings(N-1, AuxPeg, StartPeg, EndPeg)`

4-Connected Blobs

Given a matrix of on/off values count the number of connected blobs. Iterate all pixels and check for blobs.

GUI of the process, check out the code there

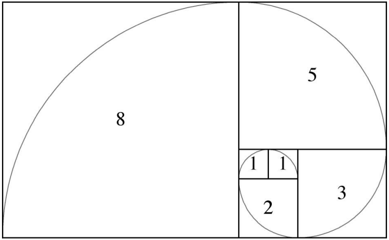


```
checkBlob(blobID, x,y)
```

- Base Case: (x,y) is visited. return;
- General Case:
  - Mark (x,y) as visited, if colored then label and:
    - `checkBlob(blobID, x-1,y)`
    - `checkBlob(blobID, x+1,y)`
    - `checkBlob(blobID, x,y-1)`
    - `checkBlob(blobID, x,y+1)`

Call-Stack Out of Space

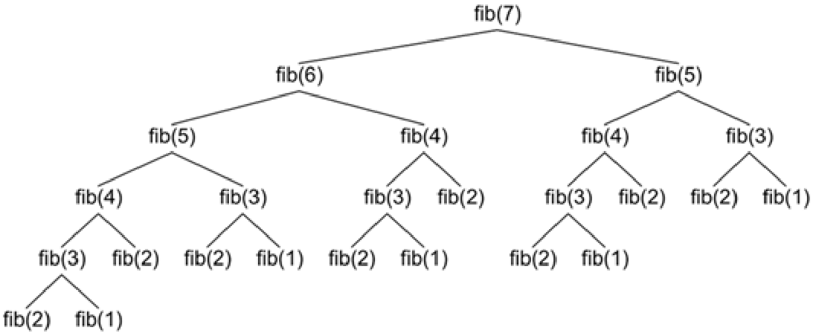
Given each method call in java gets a new element pushed onto a stack; more and more data will be allocated. And a limit might be reached resulting in a **out of space in run-time stack**. Think about a solution for finding the fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...



$$fib(n) = \begin{cases} 1 & 0 < n \leq 2 \\ fib(n-1) + fib(n-2) & n > 2 \end{cases} \tag{6}$$

Call-Stack Out of Space

$$fib(n) = \begin{cases} 1 & 0 < n \leq 2 \\ fib(n-1) + fib(n-2) & n > 2 \end{cases} \tag{7}$$



Looking at the number of steps compared to the number of subnodes: 4 →4. 5 → 8. 6 → 14. 7 → 24.

Call-Stack Out of Space

There is a lot of ineffiencies as many steps are being calculated many many times. we can rethink our algorithm to produce a tuple. Some way to recall the previous value.

$$fib(n) = \begin{cases} \{1,1\} & n = 2 \\ \{a+b, a\} s.t. \{a,b\} = fib(n-1) & n > 2 \end{cases} \tag{8}$$

```java
public static int[] fibonacci(int n){
      int [] next = new int[2];
      if(n==2) {
            next[0] = next[1] = 1;
            return next;
      }
      int[] prev = fibonacci(n-1);
      next[0] = prev[0]+prev[1];
      next[1] =prev[0];
      return next;
}
```

Verify Recursion

Verifying solution:

1. Base Case Question
   - Is the base case correct for the values
2. Smaller-Caller Question
   - Does the recusive statement progress us closer towards the base case
3. General Case Question
   - Assuming the previous one resulted correctly, is the general correct?

$$fib(n) = \begin{cases} \{1,1\} & n = 2 \\ \{a+b, a\} s.t. \{a, b\} = fib(n-1) & n > 2 \end{cases} \tag{9}$$

Memoization

Memoization: Is when data is computed as needed, but is also saved for later use.

```java
public class MemFib{
    private int[] savedFibs;
    public MemFib(){
        savedFibs=new int[999]; //defaults to 0 in all spots
    }
    public int fibonacci(int n){
        if(n<2) {
            savedFibs[n]=1;
            return 1;
         }
         int lRes = savedFibs[n-1];
         if(lRes==0) lRes = fibonacci(n-1);
         int tRes = savedFibs[n-2];
         if(tRes==0) tRes = fibonacci(n-2);
         savedFibs[n]=lRes + tRes;
         return savedFibs[n];
    }
}
```

Memoization - Factorial

```java
public class ProbabilityFacs{
    private int[] savedFacs;
    public ProbabilityFacs(){
        savedFacs=new int[999]; //defaults to 0 in all spots
    }
    public int fac(int n){
        if(n==0) {
            savedFacs[n]=1;
            return 1;
         }
         if(savedFacs[n]!=0) return savedFacs[n];
         savedFacs[n]=n*fac(n-1);
         return savedFacs[n];
    }
    public int perm(int n, int k){
        return fac(n)/fac(n-k);
    }
    public int comb(int n, int k){
        return fac(n)/(fac(n-k)*fac(k));
    }
}
```