CMPS 12B
Introduction to Data Structures
Midterm 1 Review
Solutions to Selected Problems

1. Recall the recursive function `C(n, k)` in the class `BinomialCofficients` discussed in lecture and posted on the webpage. Write a box trace of the function call `C(5, 3)`. Use this trace to find the value of `C(5, 3)`. Notice that in the full recursion tree for `C(5, 3)`, the value `C(3, 2)` is evaluated 2 times, and `C(2, 1)` is evaluated 3 times. Suggest a modification to the function that would allow it to avoid computing the same values multiple times. (Don't write the code, just explain it in words.)

   Solution to second question:
   Suggested modification: when `C(n, k)` is computed for the first time (for a particular *n* and *k*), save the value in a static 2-dimensional array for later re-use. If the value `C(n, k)` is needed at some later time, look it up in the array instead of computing it again.

   Even though the question specifically says to not write the code, it's a nice exercise to do it. The main problem is to figure out how to how to access a 'static' array. For those that wonder what that means, 'static' simply means 'unchanging'. In other words there must be some memory that each recursive invocation of the function has access to. One way to do this would be to have a global array variable, but as I've said this is deemed to be, in general, bad programming practice. The other, preferred way is to simply pass in an array reference variable to the recursive function, as follows.

```
static int C(int n, int k, int[][] BinCoef){
   if( BinCoef[n][k]!=0 ){
      return BinCoef[n][k];
   }else if( k==0 || k==n ){
      return 1;
   }else{
      return C(n-1,k-1)+C(n-1,k);
   }
}
```

   The calling function of this recursive function must allocate the array `BinCoef[][]` and initialize it to all zeros. As an exercise, envelope this function in a class, and write a `main()` function to test it.

2. Write a recursive function called `sum(n)` that computes the sum of the integers from 1 to *n*. Hint: recall the recursive function `fact(n)` in the class `Factorial` discussed in lecture and posted on the webpage. Modify your answer so as to recursively compute the sum of the integers from *n* to *m*, where $n \leq m$. (If $n>m$, return 0.)

   Solution to second question:

```
static int sum(int n, int m){
   if( n<=m ){
      return sum(n, m-1) + m;
   }else{
      return 0;
   }
}
```

3. Write a recursive function called `sumArray()` that determines the sum of the integers in an array `A[0...n-1]`. Do this in 3 ways.
   a. Recur on `A[0...n-2]`, add the result to `A[n-1]`, then return the sum.
   b. Recur on `A[1...n-1]`, add the result to `A[0]`, then return the sum.
   c. Split `A[0...n-1]` into two subarrays of length (approximately) `n/2`, recur on the two subarrays, add the results and return the sum. Hint: think about `MergeSort()`.

   Solution to part c:

```
static int sumArray(int[] A, p, r){
   if( p<r ){
      int q = (p+r)/2;
      int a = sumArray(A, p, q);
      int b = sumArray(A, q+1, r);
      return a+b;
   }else if( p==r ){
      return A[p];
   }else{
      return 0;
   }
}
```

4. Write a modification of the recursive function `BinarySearch()` that prints out the sequence of array elements that are compared to the target.

   Solution:

```
static int BinarySearch(int[] X, int p, int r, int target){
   if( p<=r ){
      int q = (p+r)/2;
      System.out.print(X[q]+" ");
      if( target==X[q] ){
         return q;
      }else if( target<X[q] ){
         return BinarySearch(X, p, q-1, target);
      }else{  // target>X[q]
          return BinarySearch(X, q+1, r, target);
      }
   }else{
      return -1;
   }
}
```

7. Use what you learned in problem 6 above to create a recursive function called `integerToString()` that returns a String representation of an integer *n* expressed in base *b*. For instance the function call `integerToString(100,8)` would return the String "144", which is what was printed in problem 6.

Solution:

```
static String integerToString(int n, int b){
   String s="";;
   if( n>0 ){
      if( n>=b ){ // same as n/b>0
         s = integerToString(n/b, b);
      }
      return s + String.valueOf(n%b);
   }else{
      return s;
   }
}
```

8. Recall the IntegerList ADT discussed in class whose states were the finite integer sequences, and whose operations were `isEmpty()`, `size()`, `get()`, `add()`, `remove()`, and `removeAll()`. Write the methods described below using only these six ADT operations. In other words you are writing methods belonging to a client of `IntegerList`.
   a. Write a `static void` method called `swap(IntegerList L, int i, int j)` that will interchange the items currently at positions `i` and `j` of the List.
   b. Write a `static int` method called `search(IntegerList L, int x)` that will perform a linear search of `L` for the target `x`. `search()` will return the List index where `x` was found, or it will return 0 if no such index exists. (Recall List indices range from 1 to `size()`.)
   c. Write a `static void` method called `reverse(IntegerList L)` that reverses the order of the items in `L`.

Solution to part a:

```
static void swap(IntegerList L, int i, int j){
   int a = L.get(i);
   int b = L.get(j);
   L.remove(i);
   L.add(i, b);
   L.remove(j);
   L.add(j, a);
}
```

Solution to c:

```
static reverse(IntegerList L){
   int i=1, j=L.size();
   while( i<j ){
      swap(L, i, j);
      i++;
      j++;
   }
}
```