Data Structures COE 312
ExamII Preparation Exercises

1. Patrick designed an algorithm called *search2D* that can be used to find a target element x in a two dimensional array A of size $N = n^2$. The algorithm *search2D* iterates through the n rows of the array A, terminating when x is found or the entire array is searched. Find hereafter a pseudo-code description of the algorithm *search2D*:

```
Algorithm search2D(x, A):
Input: An element x and an n²-element array, A
Output: The sum (i+j) such that x=A[i][j] or -1 if no
element in A is equal to x.
i ← 0
j ← −1
while i < n do
     j ← arrayFind(i, x, A)
     if j != -1
          return (i+j)
     end if
     i ← i+1
end while
return -1
```

   Note that *search2D* invokes the method called *arrayFind* on each row of the array A in order to determine the index of the column that contains the target element x. The pseudo code for the *arrayFind* method is given as follows:

```
Algorithm arrayFind(i, x, A):
Input: i, the index of the current row, x and A
Output: index j such that x=A[i][j] or -1 if no element in
row A[i] is equal to x.
j ← 0
while j < n do
     if A[i][j] == x
          return j
```

```
     else
          j ← j+1
end while
return -1
```

a. What is the worst-case running time of *search2D* in terms of n?

The worst case running time is $O(n^2)$. In fact, for each iteration of the loop in search2D the loop enclosed in arrayFind will execute completely. Since both loops will execute n times, it follows that in the worst case the algorithm runs in $O(n^2)$

b. What is the worst-case running time of *search2D* in terms of N, where N is the total size of A?

Because $N = n^2$, we can conclude that worst case running time is O(N).

2. Characterize using the big-Theta notation the worst-case running time of the following algorithm. Justify your answer.

Solution

```
Let A be given array of n integers.
for i=1 to n-1 do
     if (A[i]= 0) then
          for j=0 to i do
               Let A[i] = A[i] + A[j];
          end for
     end if
end for
```

Worst case scenario: The statement (A[i]= 0) in if block always returns true so that the inner for loop is always executed.
Value of 'i' the statement A[i] = A[i] + A[j] is executed
1 2 times
2 3 times
3 4 times
. .
. .
. .
. .

n-1 n times
Total number of times the statement is executed: f(n)= 2+3+….+n = n(n+1)/2 – 1 = O(n^2)

3. Write a recursive function to compute the binary equivalent of a given positive integer *n*. This function takes as input a base-10 value *n* and produces an output of type String that gives a binary representation of *n*. The recursive algorithm can be described as follows: recursively compute the binary equivalent of *n/2*. Append 0 to the output if *n* is even, and append 1 to the output if *n* is odd. You continue the division process (i.e., n/2) until you get a quotient that is either 0 or 1 (base cases). Use the following header for the function:

Solution

```
String binaryEquivalent(int n) {
   1.   if (n == 0)
   2.           return "0";
   3.   if (n ==1)
   4.           return "1";
   5.   if (n % 2 == 0)
   6.           return binaryEquivalent(n/2) + "0";
   7.   else
   8.           return binaryEquivalent(n/2) + "1";
}
```

4. What does the algorithm given below do? Give a "Big-Oh" characterization of its running time.

```
Algorithm foo(a, n)
Input: two integers a and n
Output: ?
k ← n
b ← 1
c ← a
while k > 0 do
     if k mod 2 = 0 then
          k ← k/2
          c ← c*c
     else
          k ← k - 1
```

```
            b ← b * c
return b
```

Solution

This algorithm computes a^n.

Its running time is O(*log*n) for the following reasons: The initialization and the **if** statement and its contents take constant time, so we need to figure out how many times the **while** loop gets called. Since *k* goes down (either gets halved or decremented by one) at each step, and it is equal to *n* initially, at worst the loop gets executed *n* times. But we can (and should) do better in our analysis.
Note that if *k* is even, it gets halved, and if it is odd, it gets decremented, and halved in the next iteration. So at least every second iteration of the **while** loop halves *k*. One can halve a number *n* at most *log*n times before it becomes <= 1. If we decrement the number in between halving it, we still get to halve no more than *log*n. Since we can only decrement *k* in between two halving iterations, we get to do a decrementing iteration at most *log*n + 2 times. So we can have at most 2**log*n + 2 iterations. This is obviously O(*log*n).

5. Design and implement a recursive application that allows an end user to test to see whether an input string is a palindrome. The user should be able to test as many strings as desired.

   Recall that a palindrome is a string of characters that reads the same forward or backward. For example, the following strings are palindromes:
   radar
   kayak

   Your application must consist of two classes named PalindromeDriver and PalindromSupport that should be stored in two separate Java files. As its name suggests, the class called PalindromeDrive.java is to be used for the purpose of testing the strings entered by the user to see if they are palindromes.

The other class (i.e. PalindromSupport.java) should contain a recursive support method called testPalindrome that takes two integer parameters representing the indexes of the characters on either end of the string being tested and that returns a boolean value indicating whether or not the tested string is a palindrome. Note that the testPalindrome method accepts only two parameters, namely the two integers storing the indexes of the two characters on either end of the string that is being tested.

Solution

```
//********************************************************************
//  PalindromeDriver.java
//********************************************************************
import java.util.Scanner;

public class PalindromeDriver
{
  //-----------------------------------------------------------------
  //  Tests strings to see if they are palindromes.
  //-----------------------------------------------------------------
   public static void main (String args[])
   {
    String str, another = "y";

        Scanner scan = new Scanner(System.in);

    while (another.equalsIgnoreCase("y"))
    {
      System.out.println ("Enter a potential palindrome:");
      str = scan.nextLine();

      PalindromeSupport test = new PalindromeSupport(str);

      if (test.isPalindrome())
        System.out.println ("That string IS a palindrome.");
      else
        System.out.println ("That string is NOT a palindrome.");
```

```
      System.out.println();
      System.out.print ("Test another palindrome (y/n)? ");
      another = scan.nextLine();
    }
  }
}

//********************************************************************
//  PalindromeSupport.java
//********************************************************************

public class PalindromeSupport
{
  private String testString;

  //-----------------------------------------------------------------
  //  Stores the string to be evaluated.
  //-----------------------------------------------------------------
  public PalindromeSupport(String test)
  {
    testString = convertString(test);
  }

  //-----------------------------------------------------------------
  //  Converts a string to correct format; removes all characters
  //  except for number and digits
  //-----------------------------------------------------------------
  private String convertString (String str)
  {
    String str2 = "";

    str = str.toLowerCase();

    for (int i=0; i < str.length(); i++)
      if (Character.isLetterOrDigit(str.charAt(i)))
        str2 += str.charAt(i);

    return str2;
  }

  //-----------------------------------------------------------------
  //  Determines if the string is a palindrome.
```

```
  //-----------------------------------------------------------------
  public boolean isPalindrome()
  {
    return testPalindrome(0, testString.length()-1);
  }

  //-----------------------------------------------------------------
  //  Recursively determines if the string is a palindrome by
  //  testing smaller substrings.
  //-----------------------------------------------------------------
  private boolean testPalindrome (int startIndex, int endIndex)
  {
    boolean result;

    if (endIndex == startIndex || endIndex < startIndex)  // base
case
      result = true;
    else
      if (testString.charAt(startIndex) ==
testString.charAt(endIndex))
        result = testPalindrome(startIndex+1, endIndex-1);
      else
        result = false;

    return result;
  }
}
```

6. Now, we will turn to a more complex problem that lends itself nicely to a recursive solution. You are required to design a class that lists all permutations of an arbitrary string supplied by the end user. It is important to note that a permutation is simply a rearrangement of the letters that make up a string. For instance, the string “eat” has six permutations associated with it, namely “eat”, “eta”, “aet”, “ate”, “tea”, and “tae”.

   The application that you are asked to develop must be made up of two classes whose partial definitions are presented below. The first class is called PermutationGenerator and is in charge of computing the answer by generating a collection of permuted strings based on the input string. The other class is

called PermutationGeneratorTester, which, as its name implies, is used to test the permutation generator class.

I have already created two program skeletons for you hereafter corresponding to the classes described above. Your job is simply to complete the two implementations as per the guidelines conveyed through the comments that are used throughout the code.

File PermutationGeneratorTester.java

```
import java.util.ArrayList;
import java.util.Scanner;
/**
     This program tests the permutation generator class
*/
public class PermutationGeneratorTester {
     public static void main(String[] args) {
          // Read a string from the end user
          Scanner scan = new Scanner(System.in);
          System.out.println("Please enter a String:");
          String str = scan.nextLine();
          //  Create  an  instance  of  the  Permutation
Generator
          PermutationGenerator     generator     =     new
          PermutationGenerator(str);
          ArrayList<String>          permutations          =
          generator.getPermutations();
          // Print all possible permutations out
          for(int i=0; i<permutations.size(); i++)
               System.out.println(permutations.get(i));
     }

}
```

File PermutationGenerator.java

```
import java.util.ArrayList;
/**
     This program generates permutations of a word
*/
public class PermutationGenerator {
     // instance variable goes here
     private String word;
     /**
```

```
        Constructs a permutation generator
        @param aWord is the word to permute
    */
    public PermutationGenerator(String aWord) {
        // other code to initialize the instance variable
        // would go here
        word = aword;
    }

    /**
        Gets all permutations of a given word
    */
    Public ArrayList<String> getPermutations() {

        ArrayList<String> result = new
ArrayList<String>();
        // The empty string has a single permutation:
        // itself
        if(word.length() == 0) {
                result.add(word);
                return resul;
        }

        /*
        Loop through all character positions using a for
        loop whose index i ranges from 0 to one less than
        the length of the string. Form a simpler word by
        removing the ith character then generate all
        permutations of the simpler word. Finally, add
        the removed character to the front of each
        permutation of the simpler word. */
        for(int i=0; i<word.length();i++) {
            //Form a simpler word by removing ith char
            String shorterWord = word.substring(0, i) +
                word.substring(i+1);

            //Generate all permutations of simpler word
            PermutationGenerator
            shorterPermutationGenerator = new
            PermutationGenerator(shorterWord);
            ArrayList<String> shorterWordPermutations =
        shorterPermutationGenerator.getPermutations();

            //Add the removed character to the front of
            //each permutation of the simpler word
            for(int j=0;
            j<shorterWordPermutations.size(); j++)
```

```
                    result.add(word.charAt(i)+
                    shorterWordPermutations.get(j));
            }
            // Return all permutations;
            return result;
        }
}
```

Here is an example that illustrates the main idea behind generating permutations recursively. Consider the string “eat”. Let us simplify the problem. First, we will generate all permutations that start with the letter ‘e’, then those that start with ‘a’, and finally those that start with ‘t’. How do we generate the permutations that start with ‘e’? To accomplish this, we need first to know the permutations of the substring “at”. This is the same problem all over again but with a simpler input, namely the shorter string “at”. Thus, we can use recursion. Generate the permutations of the substring “at”. They are: “at” and “ta”. Then, for each permutation of that substring, prepend the letter ‘e’ to get the permutations of “eat” that start with ‘e’, namely: “eat” and “eta”. Now, let us turn our attention to the permutations of “eat” that start with ‘a’. We need to produce the permutations of the remaining letters, “et”. They are: “et” and “te”. Then, we add the letter ‘a’ a the front of the strings and obtain: “aet” and “ate”. We generate the permutations that start with ‘t’ in the same way. That is the main gist of the recursive solution.