Cs2851 – Recursion

Objectives:

- Recognize characteristics of problems for which recursive solutions may be appropriate.
- Compare recursive and iterative methods with respect to time, space, and ease of development.

This lab contains two parts:

a) Convert a recursive binary-search algorithm to an iterative algorithm.
b) Develop a recursive algorithm to generate all permutations of a string.

The entire lab is due in 2 weeks including a demonstration of both programs, prior to week 8 lab period. Submit your complete lab report including your reaction, analysis, results, and source to urbain@msoe.edu.

A) Convert the following recursive version of the binary search algorithm to an iterative algorithm.

- Include sample output and timing analysis for both recursive and iterative versions.

Recursive Version of *binarySearch* Method

```
/**
 *  Returns an int indicating where (whether) an element searched for
 *  between two indexes in an array was found or not.  The worstTime(n) is
 *  O(log n).
 *
 * @param a - an array of references of type Object in which key will be
 *           searched for.
 * @param low an int that is the low index of the area of the array
 *           currently being searched.
 * @param high an int that is the high index of the area of the array
 *            currently being searched.
 * @param key a reference of type Object being searched for within the
 *           array a.
 *
 * @return an int representing either A)the element being searched for
 *        was found and its index is returned. Or, B) the element being
 *        searched for was not found and the -insertionPoint -1 is returned.
 *        The insertionPoint is where the element being searched for could
 *        be added without disordering the array.
 **/
public static int binarySearch(Object[] a, int low, int high, Object key)
{
  if (low <= high)
  {
    int mid = (low + high) / 2;
    Comparable midVal = (Comparable)a [mid];
    int comp = midVal.compareTo (key);
    if (comp < 0)
      return binarySearch (a, mid + 1, high, key);
    if (comp > 0)
      return binarySearch (a, low, mid - 1, key);
    return mid;  // key found
  } // if low <= high
  return -low - 1; // key not found; belongs at a [low]
} // method binarySearch
```

B) Develop a recursive method to print all permutations of a string using a wrapper method.

- Include sample output and timing analysis for strings of varying length.

A ***permutation*** is an arrangement of elements in a linear order. For example, if the elements are the letters 'A', 'B', 'C' and 'D', we can generate the following 24 permutations:

ABCD BACD CABD DABC

ABDC BADC CADB DACB

ACBD BCAD CBAD DBAC

ACDB BCDA CBDA DBCA

ADBC BDAC CDAB DCAB

ADCB BDCA CDBA DCBA

In general, for n elements, there are n choices for the first element in a permutation. After the first element has been chosen, there are (n-1) choices for the second element. Continuing in this fashion, we see that the total number of permutations of n elements is

$$n * (n-1) * (n-2) * \ldots * 2 * 1.$$

That is, there are n! different permutations of n distinct elements.

We will develop a method to print all permutations of a String *s*. From the above example, where *s* = "ABCD", we can print out the permutations of *s* by printing:

the six permutations that start with 'A';

the six permutations that start with 'B';

the six permutations that start with 'C';

the six permutations that start with 'D'.

How can we accomplish the printing of the six permutations that start with 'A'? Look at the above list of permutations and try to figure out how to proceed. (Hint: 6 = 3!)

The key observation is that, for those six permutations, each one starts with 'A' and is followed by a different permutation of "BCD". This suggests a recursive solution. For each of the six permutations of "BCD", we write out all of *s*, and so we get the six permutations of "ABCD" that start with 'A'.

For the next six permutations, we first swap 'A' and 'B', so that s = "BACD". We then repeat the above process -- this time permuting "ACD" and printing out all of *s* for each permutation.

For the next six permutations, we start by swapping 'B' and 'C', so that s = "CABD". We then permute "ABD" and print *s* after each permutation.

For the last six permutations, we start by swapping 'C' and 'D' -- so that s = "DABC" -- and then print s after each permutation of "ABC".

Start with a wrapper method. Then the starting position for each level of permuting can be an argument to the recursive method. Also, Java strings are immutable, so to allow swapping of characters – and use of the index operator, [ ], – we copy *s* to an array of characters. To hide these implementation details from the user, the wrapper method has a single parameter, of type *String*:

```
  /**
   * Finds all permutations of a specified String.
   *
   * @param s - the String to be permuted.
   *
   * @return a String representation of all the permutations.
   **/
  public static String permute (String s)
```

The javadoc for the recursive method, *recPermute*, is

```
  /**
   * Finds all permutations of a subarray from a given position to the end of the
array.
   *
   * @param c - an array of characters
   * @param k - the starting position in c of the subarray to be permuted.
   *
   * @return a String representation of all the permutations.
   **/
```