GENERATE PERMUTATIONS

This problem is concerned with computing all permutations of an array. For example, if the array is (2,3,5, 7} one output could be (2,3,5, 7), (2,3, 7,5), (2,5,3, 7), (2,5, 7,3), <2,7,3,5>, <2,7,5,3>, <3,2,5,7>, <3,2,7,5>, <3,5,2,7>, <3,5,7,2>, <3,7,2,5>, <3,7,5,2>, <5, 2,3, 7>, (5, 2, 7,3}, <5,3,2,7>, <5,3,7,2>, <5,7,2,3>, <5,7,2,3>, <7,2,3,5>, <7,2,5,3>, <7,3, 2,5), <7,3,5, 2), <7,5, 2,3), <7,5,3, 2). (Any other ordering is acceptable too.)
Write a program which takes as input an array of distinct integers and generates all permutations of that array. No permutation of the array may appear more than once.
*Hint:*How many possible values are there for the first element?
Solution: Let the input array be *A.* Suppose its length is *n.* A truly brute-force approach would be to enumerate all arrays of length *n* whose entries are from *A,*and check each such array for being a permutation. This enumeration can be performed recursively, e.g., enumerate all arrays of length *n* -1 whose entries are from *A,* and then for each array, consider the *n* arrays of length *n* which is formed by adding a single entry to the end of that array. Since the number of possible arrays is *n",* the time and space complexity are staggering.
A better approach is to recognize that once a value has been chosen for an entry, we do not want to repeat it. Specifically, every permutation of *A* begins with one of A[0], *A[l],...* ,*A[n* -1], The idea is to generate all permutations that begin with A[0], then all permutations that begin with *A[*1], and so on. Computing all permutations beginning with A[0] entails computing all permutations of A[1 : *n—1],*which suggests the use of recursion. To compute all permutations beginning with A[l] we swap A[0] with A[l] and compute all permutations of the updated *A[*1: *n*-1]. We then restore the original state before embarking on computing all permutations beginning with *A[* 2], and so on.
For example, for the array (7, 3,5), we would first generate all permutations starting with 7. This entails generating all permutations of (3,5), which we do by finding all permutations of (3,5) beginning with 3. Since (5) is an array of length 1, it has a single permutation. This implies (3,5) has a single permutation beginning with 3.
Next we look for permutations of (3,5) beginning with 5. To do this, we swap 3 and 5, and find, as before, there is a single permutation of (3,5) beginning with 5, namely, (5,3). Hence, there are two permutations of *A* beginning with 7, namely (7,3,5) and (7, 5,3). We swap 7 with 3 to find all permutations beginning with 3, namely (3, 7, 5) and (3,5,7). The last two permutations we add are (5,3,7) and (5,7,3). In all there are six permutations.

```
public static List<List<Integer>> permutations(List<Integer> A) {
List<List<Integer>> result = new ArrayList <>();
directedPermutations(0 , A, result);
return result;
}
private static void directedPermutations(int i, List<Integer> A,
List<List<Integer>> result) {
```

```
if (i == A.sizeO - 1) {
result.add(new ArrayList <>(A));
return ;
}
// Try every possibility for A[i].
for (int j = i; j < A.sizeO; ++j) {
Collections.swap(A , i, j);
// Generate all permutations for A.subList(i + 1, A.sizeO).
directedPermutations(i + 1, A, result);
Collections.swap(A , i, j);
}
}
```

The time complexity is determined by the number of recursive calls, since within each function the time spent is *0(1),* **not including the time in the subcalls. The number of function calls,** *C(n)* **satisfies the recurrence C(n) = 1 +** *nC(n* **- 1) for** *n >* **1, with C(0) = 1. Expanding this, we see** *C(n)* **= 1 +** *n+ n(n* **- 1) +** *n(n* **- 1)(n - 2) + …+ n! = n!(1/n! + 1***/(n* **- 1)! + 1** */(n* **- 2)! +…+ 1/1!). The sum (1 + 1/1! + 1/2! + …+1/n!) tends to Euler's number** *e,* **so** *C(n)* **tends to** *(e - 1)n!,* **i.e.,** *0(n!).* **The time complexity T(n) is** *0(n* X **n!), since we do***0(n)* **computation per call outside of the recursive calls.**

```
public class AllPermutationsArray {


    static int count = 0;


    private static int factorial(int n) {

        if (n == 1 || n == 0) {

            return 1;

        } else {

            return n * factorial(n - 1);

        }



    }


    public static int[][] permutations(int[] A) {

        int[][] result = new int[factorial(A.length)][A.length];

        directedPermutations(0, A, result);

        return result;
    }

    private static void directedPermutations(int i, int[] A,
                                             int[][] result) {
        if (i == A.length - 1) {
            int[] perm = Arrays.copyOf(A, A.length);
            result[count++] = perm;
            return;
        }
// Try every possibility for A[i].
        int temp;
        for (int j = i; j < A.length; ++j) {
            // Collections.swap(A, i, j);
            temp = A[i];
            A[i] = A[j];
            A[j] = temp;

// Generate all permutations for A.subList(i + 1, A.sizeO).
            directedPermutations(i + 1, A, result);
            // Collections.swap(A, i, j);
            temp = A[i];
            A[i] = A[j];
            A[j] = temp;
        }
    }

//    public static void main(String[] args) {
//
//       int[][] list = permutations(new int[]{2, 3, 5, 7});
//        for (int[] is : list) {
```

```
//              System.out.println(Arrays.toString(is));
//          }
//
}
```

Variant: Solve Problem 16.3 on Page 287 when the input array may have duplicates.

You should not repeat any permutations. For example, if A = (2, 2,3, 0} then the out¬

put should be <2, 2,0,3), <2, 2, 3, 0>, <2,0, 2,3>, <2, 0,3, 2>, <2,3, 2, 0>, <2,3, 0, 2), <0, 2, 2,3>,

<0, 2,3, 2), <0,3, 2, 2), <3, 2, 2,0>, <3, 2, 0, 2), <3, 0, 2, 2>.

You can use the most common implementation of permutations (swap an element with the first and permute the rest). First build the string, sort it, then generate all possible permutations. Don't allow duplicates.

An implementation could be:

```
static String swap(String s, int i, int j) {
    char [] c = s.toCharArray();
    char tmp = c[i];
    c[i] = c[j];
    c[j] = tmp;
    return String.copyValueOf(c);
}

static void permute(String s, int start) {
    int end = s.length();

    if(start == end) {
        System.out.println(s);
        return;
    }

    permute(s, start + 1);

    for(int i = start + 1; i < end; i++) {
        if(s.charAt(start) == s.charAt(i)) continue;
        s = swap(s, start, i);
        permute(s, start + 1);
    }
}

public static void main(String [] args) {
    String s = "aabb";
    permute(s, 0);
}
```

Produces output:

```
aabb
abab
abba
baab
baba
bbaa
```

credits -http://k2code.blogspot.in/2011/09/permutation-of-string-in-java-efficient.html

```
    public class Permutation {

public static void printDuplicates(String str,String prefix)
{
    if(str.length()==0)
    {
        System.out.println(prefix);
    }
    else
    {
        for (int i = 0; i<str.length();i++)
        {
            if(i>0)
            {
                if(str.charAt(i)==str.charAt(i-1))
                {
                    continue;
                }
            }

                printDuplicates(str.substring(0, i)+str.substring(i+1, str.length()),prefix+str.charAt(i));

        }
    }
}
    public String sort(string str){
    // Please Implement the sorting function, I was lazy enough to do so
    }
public static void main(String [] args)
{
    String test="asdadsa";
    test = sort(test);
    printDuplicates(test,"");
}
}
```