CSCD320 Homework 7, Winter 2012, Eastern Washington University. Cheney, Washington.

Name: Eric Fode **EWU ID:00530214**

Solution for Problem 1

1.Idea: Find an k such that it is the smallest value greater then all of the values to the right of j. Swap j and k. Sort the elements to the right of j in ascending order. The last case is an edge case and the array will need to be reversed

2. Pseudocode:

getNextPerm(set)

Input: A set of numbers
Result: All of the permutations in lexicographical order

```
1  temp ← 0;
2  j ← array.length − 1;
3  k ← array.length − 1;
4  while r ≥ s do
5      j ← j − 1;
6      if j < 0 then
7          reverse(array);
8          return array;
       /* Find k                                                    */
9      for array[j] > array[k]; j ← j − 1 do
10     swap(array[k], array[j]);
11     sort(array[j + 1], array[array.length]);
12 return array;
```

3. Analysis of worst case running time: The worst case running time is $O(n)$ depending on the sort that is used. While it looks like it may be $O(n\log(n))$ because of the sort, the numbers in the worse case will always be out of order in a related way because of this you can pick a sort that will work optimally in this situation.

4. Example of worst case: 3412 would require the most work because k is as far away from j as possible making the sort function (the only part of this algorithm that is effect in more then a constant manner by changes in ordering) dominate the running time of this iteration.

5. Aggregate analysis: All operations in this algorithm take the same time during every permutation except for the sort, which over a sequence of iterations will add up to a constant value no matter what permutation you start with because on average it takes constant time. So all of the iterations have the same cost of $O(n)$ thus the amortized cost of each operation is $O(1)$.

6. Accounting analysis: This answer was gathered from research here `http://stackoverflow.com/questions/9560316/amortized-time-cost-using-accounting-method`

1. Measure the running time in terms of swaps since that is the only thing that costs anything (the other work per iteration is in the worst case the same as the number of swaps).

2. The swap of j and k has a amortized cost of 2 because of the sorting that is required (again the cost of the sort varies uniformly with every possible input for a given array size so it in effect can be reduced to a constant cost).

3. Now to show that we don't go into debt, it is enough to show that the swap of j and k leaves credit at position j, then every other swap will be guaranteed to have credit available at its position. Doing this establishes that between iterations, if an item is larger than the one immediately following it, then it

was put in it's current position by a swap that left an as-yet unconsumed credit (from leaving a credit a position j).

4. So the amortized cost is $O(3) = O(1)$ two for j and k swap and one for all of the constant time operations.

7. Potential method analysis:
research was done here:
http://www.scielo.br/scielo.php?pid=S0104-65002001000200009&script=sci_arttext
From 1 to n let $c_i = 1 if \pi_i > \pi_i + 1$ else $c_i = 0$ so for all iterations except the edge case let $c_i = 1$ and define the potential function $\phi(\pi) = \sum_{i=1}^{n-1} c_i$ it is clear that for each iteration this function will increase so it is a valid potential function, so $a = t + \delta\phi = O(1)$.