Maria Swartz
Lab04
Algorithms
Time complexity of premutation sort

Analyzing perm(a):

```
function perm(a)
{
   var output = [];                                          Constant time
   //Base case; recursion stops here
   if(a.length == 1)                                                Constant time
   {
      return [a];                                                   Constant time
   }
   //Recursive part
   for(var i=0; i<a.length; i++)                                    Linear time, n
   {
      var head = a[i];                                              Constant time
      //All the elements in the array, except the head
      var rest = a.slice(0, i).concat(a.slice(i+1));          Constant time
      //Find all the permutations of the rest
      var tail = perm(rest);                                  Recursive call,, T(n-1)
      //Find all permutations of the tail
      for(var j=0; j<tail.length; j++)                        Linear time, n
      {
         output.push([head].concat(tail[j]));
      }
   }
   return output;                                                   Constant time
}
```

Using our analysis above, we get the following:

T(n) = { n * T(n-1) * n + 1 = n^2 * T(n-1) + 1

Next we will us substitution to see if we can find a pattern in the recurrence relation.

T(n) = n^2 * T(n-1) + 1
= n^2 * ([(n-1)^2] * T(n-2) + 1) + 1
= n^2*[(n-1)^2] * T(n-2) + n^2 + 1
= n^2*[(n-1)^2] * [[(n-2)^2] * T(n-3) + 1] + n^2 + 1
= n^2*[(n-1)^2]*[(n-2)^2] * T(n-3) + n^2*[(n-1)^2] + n^2 + 1
= n^2*[(n-1)^2]*[(n-2)^2] * ([(n-3)^2]* T(n-4) + 1) + n^2*[(n-1)^2] + n^2 + 1
= n^2*[(n-1)^2]*[(n-2)^2]*[(n-3)^2] * T(n-4) + n^2*[(n-1)^2]*[(n-2)^2] + n^2*[(n-1)^2] + n^2 + 1

For the following equations, we assume that i is the number of recursive calls we have to make

For the first, blue part of our equation, we can see the pattern: $\frac{n^2!}{(n-i)^{2!}}$

For the second, green part of our equation, we can see the pattern: T(n-i)

For the third, orange part of our equation, we can see the pattern: $\sum_{m=0}^{i-1} \frac{n^2!}{(n-m)^{2!}}$

Using this information, we get that:

$$T(n) = \frac{n^2!}{(n-i)^{2!}} * T(n-i) + \sum_{m=0}^{i-1} \frac{n^2!}{(n-m)^{2!}}$$

With the way the code is written, for an array of length 4, we will make 40 recursive calls. With a length of 3, we will make 9 recursive calls, and with a length 2, we will make 2 recursive calls. This ends with i being: $n! \sum_{k=1}^{n-1} \frac{1}{k!}$

We can simplify the above information to incorporate this:

$$T(n) = \frac{n^2!}{\left(n - \left(n - n! \sum_{k=1}^{n-1} \frac{1}{k!}\right)!\right)^2!} * T\left(\left(n - n! \sum_{k=1}^{n-1} \frac{1}{k!}\right)\right) + \sum_{m=0}^{n! \sum_{k=1}^{n-1} \frac{1}{k!} - 1} \frac{n^2!}{(n-m)^{2!}}$$

The top of the first part of this relation will grow the fastest. So we could simplify our equation to get T(n) = (n^2)!.

For our second function:

```
function permutationSort(a)
{
   var bigArr = perm(a);                          T(n) of the above function, so (n^2)!
   //found how to sort an array numerically at: https://www.w3schools.com/js/tryit.asp?filename=tryjs_array_sort2
   var sortedArr = insertionSort(a);   Insertion sort, which has a complexity of
nlogn
   for(var i = 0; i<bigArr.length; i++)            Liner, n
   {
      //compare if permutation i is the sorted array
      if(JSON.stringify(bigArr[i]) == JSON.stringify(sortedArr))              constant
      {
         a = bigArr[i];                                    constant
         return i+1;                                 constant
      }
   }
   //an error occurred because none of the permutations is sorted
   return -999;
}
```

Overall runtime complexity:

$$T(n) = (n^2!) + n + nlogn + 1$$

We are only concerned with the high order terms, so we get

$$T(n) = (n^2!) \in \Theta(\mathrm{n!})$$

Using all of this information, we get that the runtime complexity of our algorithm is Θ(n!)
In the best case, the list is sorted. That would give us a complexity of

$$T(n) = (n^2!) + nlogn + 1$$

This would still simplify Θ(n!)

In the worst case, the list is reverse sorted. That would give us a complexity of

$$T(n) = (n^2!) + n + nlogn + 1$$

This would still simplify Θ(n!)

In both of these cases, our code would still generate all the possible permutations of the list. The only thing that changes between the best and worst cases is the amount of time need to check the array of all permutations for the first permutation that is sorted. Since this amount is dwarfed by the amount of time needed to generate all of the permutations, it can be disregarded.

If we generated permutations randomly without memory instead of systematically trying them, our implementation could range between the best and even worse than the worst case. In the best case, we would randomly generate only one of each of the permutations in the list and then sort through them. The sorting may be faster, but the permutation and recursion part (the most expensive part of the algorithm) would still be the same. In the worst case, we would generate the same permutation multiple times. Since this occurs in the most time intensive part of the algorithm, we could have a case where we have calculated all but one of the possible permutations but are still trying to randomly get the last one. During this, we would be wasting time trying to generate this last permutation so that our array would be complete.