Hetav Pandya (Student ID: 1005729124) and Chirag Sethi (Student ID: 1006219263)

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	...	...	Total
Mark										

Table 1: Marking Table

Exercise 1

(a) Swapping takes constant time, hence the first 'if' block is in O(1). Then our algorithm diverges into three function calls each with a size of $\frac{2n}{3}$. This gives us the time complexity equation of:

$$T(n) = O(1) + 3 * T(\frac{2n}{3})$$

Using Master's Theorem we get $T(n) = n^{log_{1.5}3}$

(b) To prove it by induction we first test the case where n = 1. In this case, i = 0 and j = 0. Hence, we do not enter the 'if' condition block. Along the same lines, we do not enter the while loop because j - i = 0, which is less than 2. Hence, an array with a single element remains unchanged, as one should expect with a sorting algorithm. Hence, the algorithm is correct for n = 1.

Now we assume that the algorithm is correct for n elements. If we were to assume this, then it follows that the sorting of the 2n/3 sized subarrays within the algorithm is also valid. Notice that in the case of n = n, i = 0 and j = n-1. Hence, t (the variable in the while loop) will be equal to n/3, where the question wants us to assume that n is a multiple of 3.

With this info, we try to prove that the algorithm is valid for n + 1 elements. Here, i = 0 and j = n in the first iteration of the algorithm. The variable 't', will be equal to $\lfloor \frac{n+1}{3} \rfloor$. Assuming that n is a multiple of 3 this basically converts to t = n/3. Clearly, this means that the next recursive calls of sort will be placed on array's with the length 2n/3. This is the same length which was called when sorting array of length n instead of n + 1. Given our assumption that the sorting algorithm works for size n arrays, it should also work on size n + 1 arrays because they call sort on the same sized arrays. Therefore, it is

also valid for n + 1 sized arrays.

In conclusion, the algorithm works for 1, n and n + 1 elements. Hence, proved by induction.

Exercise 2

We will first iterate through the array and insert the values in a hash table. We know that the values will range between $-n^2$ and n^2. Hence, we will loop from $-n^2$ to n^2 and add elements to a new array if they are found in the hashtable. The pseudo code is as follows:

```
dict hash
for elem in array:
     // The key is for accounting for duplicates
     if elem exists in hash:
          hash[elem] += 1
     else:
          hash.insert(elem)
          hash[elem] = 1
// Iterate through the array and add elements to the new array
output = []
for i in range(0, n):
     if -(i * i) exists in hash:
          while(hash[-(i*i)]--):
               output.add(-(i * i))
for i in range(0, n):
     if (i * i) exists in hash:
          while(hash[(i*i)]--):
               output.add(i * i)

return output
```

The first 'for' loop takes O(n) time to iterate through the array and inserting in a hashtable is an O(1) operation. Next we loop twice from 0 to n and call the search function on a hashtable. Search is also a constant time operation for hash tables. Hence, this gives a total time complexity of O(n + n + n), which is in O(n).

Also we use memory for two operations, one is for constructing the hashtable and the other is for constructing the output array. Both use auxilary space of O(n),

hence our memory consumption is also in O(n)!

Exercise 3

(a) We will first perform an inorder traversal of the two AVL trees. This will give us two sorted arrays. We will then merge the two arrays using the following algorithm. Note that this is the same algorithm we use in merge sort to join two sorted arrays.

```
output = []
i, j = 0, 0
while(i < len(arr1) and j < len(arr2)):
      if (arr1[i] > arr[j]):
            output.add(arr[j])
            j++
      else:
            output.add(arr[i])
            i++
if (i != len(arr1)):
      while (i < len(arr1)):
            output.add(arr1[i])
            i++
else:
      while (j < len(arr2)):
            output.add(arr2[j])
            j++
```

Once we have a sorted array we use binary splitting to make an AVL tree. For example, assume we have the sorted array of the form: 1, 2, 3, 4, 5, 6.

We now divide the array at the middle (at 4). The middle element becomes the root and the two smaller arrays are then passed recursively to the function as the left and right subtree of 4.

So the middle of the first subarray (1, 2, 3) is 2. This becomes the left child of 4. Similarly, 5 is the middle element of the right subarray and hence, becomes the right child of 4. This happens recursively.

Complexity Analysis

The inorder traversals take O(n1) and O(n2) time respectively. The merge operation takes O(n1 + n2). The binary tree generation procedure follows the following complexity equation and takes O(n1 + n2) according to master's theorem.

$$T(n) = 2T(n/2) + O(1)$$

(b) A few realizations that helped me formulate the solution are as follows:

- I cannot insert or delete more than constant number of elements because doing so can lead the complexity in $O(nlogn)$.
- Because on tree is strictly less than the other we can create a valid BST by just using the max element of the smaller tree as a root. The new small tree will then be the left sub-tree and the large tree will be the right subtree. However, this won't necessarily be an AVL. Here small and large refer to key values.
- I cannot traverse all the elements in the tree because doing so will lead the complexity in $O(n)$.

Keeping these in mind, here's my solution. We first traverse through both the trees T_1 and T_2. This takes $O(h_1)$ and $O(h_2)$ time respectively. Let's assume that the height of T_1 is less than the height of T_2. Based on my realization in point 2, we first find the maximum element in the smaller tree T_1. This will just require us to go right each time we traverse and hence, will take $O(h_1)$ time. Let's call this node the 'pivot'.

Unfortunately, we cannot directly make it the root and insert T_2 as the right subtree as it won't preserve the AVL property in general (Realization point 2). What we could do instead is traverse left through the tree T_2 and find an element which is at a height h_1 (even $h_1 + 1$ should work). Let's call this node the alpha node for future reference. This traversal would require $O(h_1)$ time. The reason why we do this is because, we will now add this node as the right node to the pivot.

Now because the right subtree is from T_2 it will be bigger than the pivot, and its height would be h_1 or $(h_1 + 1)$. Now we add tree T_1 as the left subtree to the pivot. This operation takes constant time. So let's summarize what we have now: We have the pivot as the root. Its right subtree is a portion of T_2 with height equal to that of T_1 (+/- 1).

Now the next step, is to insert this in place of the alpha node. The tree at the alpha node is a valid AVL tree by the nature of design. But the entire tree might be one off from the AVL property because the new height of the alpha node may increment by one. To nullify this, we basically perform a suitable rotation to make up for this differnce. This could be one of Left, Right, Left-Right or Right-Left rotations. Because we only have to do a constant number of rotations, the complexity of this step is O(1).

Time complexity

Based on our algorithm, the most time consuming step is finding the height of the trees which takes $O(h_1) + O(h_2)$ time. Because they are AVL trees, this translates to: $O(log(n_1))+O(log(n_2))$ time. In our case, h_1 was smaller than h_2, hence n_2 was the dominant term. Therefore, the complexity will be bounded by $O(logn_2)$. In general, the logarithm of the maximum element bounds the algorithm.

Exercise 4

The hash function described in the text is a s follows:

$$h(key, i) = (key \ mod \ 7 + i((3 * key) \ mod \ 4)) \ mod \ 7$$

Now let's start inserting the keys, first its 24:

$$h(24, 0) = 3$$

Then its key 19:

$$h(19, 0) = 5$$

Followed by key 51:

$$h(51, 0) = 2$$

Then next key is 30:

$$h(30, 0) = 2 \ [Occupied]$$
$$h(30, 1) = 4$$

Then we have 10:

$$h(10, 0) = 3 \ [Occupied]$$
$$h(10, 1) = 5 \ [Occupied]$$

$$h(10, 2) = 0$$

Then we have 32:

$$h(32, 0) = 4\ [Occupied]$$

$$h(32, i) = 4\ [Being\ dropped : (]$$

Then we have 41:

$$h(41, 0) = 6$$

Then 8:

$$h(8, 0) = 1$$

Then finally we have 63:

$$h(63, 0) = 0\ [Occupied]$$

All slots are filled so any value of 'i' won't help.

The final hash table is of the form: 10, 8, 51, 24, 30, 19, 41. The numbers that are dropped are 32 and 63. 32 was dropped because the none of the values of i yielded an empty spot even though the table had an empty spot when inserting 32. However, in case of 63, the entire hash table was full so no value of i could yield an empty block.

Exercise 5

(a) Let's consider not putting the element in that specific slot as a success (p) and define a failure (q) as putting an element in that slot. Based on this we have:

$$p = \frac{m-1}{m}\ and\ q = \frac{1}{m}$$

Now to ensure that our specific slot is empty after filling all n elements we need to have n successes.

$$Prob = p^n$$

$$\implies Prob = (1 - \frac{1}{m})^n$$

Using binomial theorem this translates to:

$$Prob = 1 - \frac{n}{m} + \frac{n(n-1)}{2m^2} + ...$$

$$\implies Prob > 1 - \frac{n}{m}$$

Hence, $c_0 = 1$ and c_1 = -1

(b) In this case we are looking for only one failure and n-1 success. This gives:

$$Prob = \binom{n}{1} * p^{n-1} * q$$

$$\implies Prob = n * (1 - \frac{1}{m})^{n-1} * \frac{1}{m}$$

Using binomial theorem:

$$\implies Prob > (1 - \frac{n-1}{m}) * \frac{n}{m}$$

$$\implies Prob > \frac{n}{m} - \frac{n(n-1)}{m^2}$$

This gives us, $f_1(n) = n$ and $f_2(n) = n(1-n)$

(c) Here the key word is 'atmost', hence the slots can contain 0 or 1 elements, in other words no chaining. To do this let's find what is the probability of single slot hlding atmost one element:

$$Prob = Part(a) \ + \ Part(b)$$

$$Prob = \left(1 - \frac{n(n-1)}{m^2}\right)$$

For all m slots it would be:

$$Prob = \left(1 - \frac{n(n-1)}{m^2}\right)^m$$

Using binomial expnsion:

$$Prob > 1 - \frac{n(n-1)}{m} \geq 1 - \frac{1}{n^\epsilon}$$

For very large m ¿¿ n, we have:

$$1 - \frac{n^2}{m} \geq 1 - \frac{1}{n^\epsilon}$$

$$\implies n^{2+\epsilon} \geq m$$

Hence, if $m = n^{2+\epsilon}$, we will have the probability bounded by $1 - \frac{1}{n^\epsilon}$, for $\epsilon > 0$.