Hetav Pandya (1005729124) and Chirag Sethi (1006219263)

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

Table 1: Marking Table

Exercise 1

(a) $f(n) = 5n^2$ and $g(n) = n^2$

$f(n) < cg(n)$, where $c > 5$. Additionally $f(n) < cg(n)$, when $c < 5$. Hence, $f(n) = \Theta(g(n))$

(b) $f(n) = 3n^6 + 4n^2 + 5$ and $g(n) = n^5$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{3n^6 + 4n^2 + 5}{n^5} = \lim_{n\to\infty} 3n + \frac{4}{n} + \frac{5}{n^5}$$

Clearly $f(n)$ is asymptotically greater than $g(n)$. This means $f(n) = \Omega(g(n))$.

(c) $f(n) = \log(n) + \frac{1}{n}$ and $g(n) = \log(n)$

Because of the additional $1/n$ term $f(n)$ will be greater than $g(n)$ for $n > 0$. But as n approaches ∞ we get:

$$\lim_{n\to\infty} f(n) = \log(n) < cg(n)$$

This is true for very big n and $c > 1$. Hence $f(n) = \Theta(g(n))$.

(d) $f(n) = \frac{\log(n)}{n}$ and $g(n) = \frac{1}{n}$

We can clearly see that $f(n)$ is $\log(n) \cdot g(n)$. However, let's say that there's a value of c for which $g(n)$ is greater than $f(n)$.

$$\frac{\log(n)}{n} < \frac{c}{n} \implies \log(n) < c$$

Clearly, this is not possible when n is unbounded. Hence $f(n) = \Omega(g(n))$.

(e) $f(n) = 2^n$ and $g(n) = 2^{2n}$

Clearly $g(n)$ is greater than $f(n)$ for $c = 1$. But is it true for all c? Let's assume:

$$f(n) > cg(n) \implies \log(f(n)) > \log(cg(n))$$
$$\implies n > \log(c) + 2n$$
$$\implies c = e^{-n}$$

There is no constant value of c for which our assumption is true and hence, $f(n) = O(g(n))$.

(f) $f(n) = 2^{k\log(n)}$ and $g(n) = n^k$

Assuming base 2, we know $a^{\log(n)} = n$, using this:

$$f(n) = n^k$$

Clearly this means, $f(n) = \Theta(g(n))$.

(g) $f(n) = 2^{2022}n^2$ and $g(n) = \frac{n^2}{2^{2022}}$ (We will only look at very large values of n)

Both $f(n)$ and $g(n)$ are of the order n^2 and hence $f(n) = \Theta(g(n))$

(h) $f(n) = 2^{\sqrt{\log(n)}}$ and $g(n) = (\log n)^{100}$

Take log both sides and then we get, $\sqrt{\log(n)}$ and $\log(\log(n))$ (We ignore the constant). We know that taking a log of a value reduces its value much more than taking a square root. Eg. $log(100) < \sqrt{100}$. Hence, $f(n) = \Omega(g(n))$.

(i) $f(n) = (\log n)^{\log(n)}$ and $g(n) = n^{\log(\log(n))}$

Taking log both sides, we get: $\log(n) \cdot \log(\log(n))$ and $\log(\log(n)) \cdot \log(n)$. Hence, $f(n) = \Theta(g(n))$.

(j) $f(n) = 3^{2^n}$ and $g(n) = 2^{2^{n+1}}$

Taking log both sides $f(n)$ and $cg(n)$:

$$1.58 \cdot 2^n < 2 \cdot 2^n + \log(c)$$

This means that $g(n)$ will be asymptotically greater than $f(n)$, when n is unbounded regardless of the constant c. Hence, $f(n) = O(g(n))$.

Exercise 2

(a) $T(n) = 2T(n/4) + n\log(n)$

Using master's theorem: a = 2 and b = 4. This is the third case where $f(n) = \Omega(n^{0.5+\epsilon})$. Next we check $af(n/b) < cf(n)$:

$$\implies (n/2)\log(n/4) < cn\log(n)$$

$$\implies 2c > 1 - \frac{\log(4)}{\log(n)}$$

The above equation is true for large n and $c > 0.5$. This means $T(n) = \Theta(n\log(n))$.

(b) $T(n) = 3T(n/2) + n^{1.5}$

Using master's theorem: a = 3 and b = 2. This is the first case because $\log_2(3) > 1.5$. This means $T(n) = \Theta(n^{\log_2(3)})$.

(c) $T(n) = 27T(n/3) + n^3$

Using master's theorem: a = 27 and b = 3. This is the second case because $n^{\log_3 27} = n^3$. This means $T(n) = \Theta(n^3\log(n))$.

(d) $T(n) = 4T(n/2) + (n\log n)^2$

Using master's theorem: a = 4 and b = 2. This does not fall in any of the basic three cases, because, n^2 is not polynomially greater than $n^2\log^2(n)$.
This is actually a special case and we can solve it using the theorem:

Theorem 1.1: In a recurrence equation given by $T(n) = aT(n/b) + f(n)$, if $f(n) = \Theta(n^{\log_b a}\log^k(n))$, where $k > 0$, then recurrence of $T(n) = \Theta(n^{\log ba}\log^{k+1} n)$.

Based on this $T(n) = \Theta(n^2\log^3 n)$.

Exercise 3

(a) Array with only 1's and 2's

The algorithm I devised for this question uses the following pseudo code:

```
start = 0
end = len(array) - 1
while (start < end)
      if (array[start] == 2 and array[end] == 1)
            flip(start, end)
            start++
            end--
      else if (array[start] == 2 and array[end] == 2)
            end--
      else if (array[start] == 1 and array[end] == 1)
            start++
      else
            start++
```

The cost of each flip operation is said to be constant in the question. Each conditional if/else statement also takes constant time. So all the code inside the while loop takes constant time during one iteration. The while loop however runs over the entire array (from both sides). Therefore, the total time complexity is O(n), where n is the amount of money.

(b) Array with arbitary integer value

In this question, I followed a reverse engineering approach using Master's theorem. $O(nlog^2n)$ complexity is not a basic case of the theorem. However, it is a special case described by Theorem 1.1 [See Exercise 2 (d)]. Hence, the complexity equation should look similar to:

$$T(n) = aT(\frac{n}{a}) + O(n\log(n))$$

The first term hints at a binary division of the array and the second term refers to the merge operation. We will explore them one by one. The pseudo code for part one looks as follows:

```
binary_divide(array):
    if (len(array) == 1)
        return array
    else if (len(array) == 2):
        if (array[0] < array[1])
            return array
        else
            flip(0, 1) %% Will need to pass array
            return array
    else
        first_part = binary_divide(array[:len(array) / 2])
        second_part = binary_divide(array[len(array)/2:])
        final_array = merge(first_part, second_part)
        return final_array
```

flip(0, 1) is a constant time operation because O(j - i) = O(1 - 0). All conditionals if/else statements also take constant time. However, merging requires non-constant time and for now we can refer to it as O(Merge). The algorithm divides the array into two equal parts and recursivly calls itslef on each of them. Based on this our time complexity equation now becomes:

$$T(n) = 2T(\frac{n}{2}) + O(Merge)$$

Note that merging receives two sorted subarrays which it has to join and return a sorted array. The first element of the second array will be smaller than all the other elements in the second array. We will use this as a pivot.

$$< 1, 5, 8 > < 4, 6, 9 >$$

We will start comparing this element with the first array starting from index 0 (first element of first array). When we find a number that is greater than the pivot we call a flip operation. In the example above 4 and 5 will be flipped as follows

$$< 1, 4, 8 > < 5, 6, 9 >$$

Now all the elements on the right of the pivot (4) are greater than the pivot and the ones on the left are less than the pivot. However, on the right we have reversed the sorted order of the newly flipped subarray (8, 5). It is quite easy to restore it. We just need to call one more flip operation after the new position of the pivot to the original position of pivot.

$$< 1 > < 4 > < 5, 8, 6, 9 >$$

Now the left and pivot will always be sorted and in the correct order. However, the right of pivot still might not be sorted. Hence, we recursively call the merge function on the reduced sub arrays.

$$< 1, 4 > + merge(< 5, 8 >, < 6, 9 >)$$

A few notes on the above algorithm:

- The time complexity is solely dependent on the size of first array, because we do not traverse the second array except pivot.
- At max we will need to traverse all the elements in the first array which takes O(n) time. Flip also takes O(n) time in general.
- However, if the pivot is greater than all elements of the first array then we simply join them. This is the best case with complexity in O(1)
- The worst case occurs when pivot is smaller than all elements in the first array. This will give a complexity equation:

$$T(n) = O(n) + T(n-1)$$

- On average we can assume that the pivot falls nearly at the middle of the first array. This means that the next iteration of merge will have n/2 elements where n were the number of elements in the first array.

$$T(n) = O(n) + T(n/2) \implies T(n) = O(n)$$

This makes the entire algorithm function in:

$$T(n) = 2T(\frac{n}{2}) + O(n) \implies T(n) = O(nlog(n))$$

Suprisingly, the average case is better than $O(nlog^2(n))$

Exercise 4

I first thought about divide and conquer algorithms but they seemed to not work because the left and right subtree of a max-heap are not dependent and/or share any specific relationship as in a binary search tree. Hence, I then decided to use another max-heap as described in the hints. Here are a list of realizations that helped me develop the pseudocode:

1. We know that the root of a max-heap is the largest element by definition.
2. At a level k (from top) any element has atleast k - 1 elements greater than itself.
3. Left and right subtrees do not necessarily share a relationship like less than or greater than.

```
new_max_heap = max_heap()
k_largest(heap, k):
    new_max_heap.insert(heap.root)
    depth = 1
    while (depth < k - 1)
        max_elem = new_max_heap.root
        new_max_heap.delete(max_elem)
        left_child = heap.leftchild(max_elem)
        right_child = heap.rightchild(max_elem)

        new_max_heap.insert(leftchild)
        new_max_heap.insert(rightchild)
        depth++
    return new_max_heap.root
```

Note that I have ommited basic error checking like if the heap is NULL or if the right or left child nodes exist, beacuse it is fairly trivial and dependent on the programming language used.

Here, insert and delete operations take log(n) time complexity. But because we are using these operations on the new heap, which will have a maximum of k elements they turn out to be O(log(k)) operations. Getting the left and right child is constant time operation as they can be obtained by using index formula (l = 2i and r = 2i + 1). This makes the code inside the while loop have a complexity of log(k). The while loop will run for k - 1 iterations and hence, the total time complexity is in O(k log(k))!

Exercise 5

(a) Keys are basically arranged as a binary search tree so we will simply use the binary search algorithm. This algo starts from the root, goes left if root is

greater than the key to be found and right otherwise. Eventually it will find it provided the key exists.

(b) We can use the same algorithm we use to insert nodes in a BST but with some modifications. We will first insert the new node based on its key value, using the insert algo from BST. And then we will correct it's postitions using rotation to satisfy the priority conditions.

```
insert(PKTree, node):
    insert_BST(PKTree.root(), node)
    fix_priority(PKTree, node)
    return

insert_BST(root, node):
    if (node.key < root.key)
        if (root.left exists)
            insert_BST(root.left, node)
        else
            root.left = node
            return
    else
        if (root.right exists)
            insert_BST(root.right, node)
        else
            root.right = node
            return

fix_priority(tree, node):
     if (node.priority > node.parent.priority)
        if (node.parent.left == node)
            new_node = right_rotate(tree, node.parent)
            fix_priority(tree, new_node)
        else
            new_node = left_rotate(tree, node.parent)
            fix_priority(tree, new_node)
    else
        return
```

The left and right rotations are explained in section 13.2 of the textbook and we will use similar implementation.

(c) To delete an element we use a similar idea. We first delete the node based on its key values similar to how we delete it in binary search trees. This is sufficient if the node that is deleted is a leaf node or has only left or right subtree. However, if it has both left and right subtree then we need to call a function to fixup priority.

To solve this, the idea behind the fixup is that we will start from the node that replaces the deleted node. Everything above this is already in a priorotized order as we didn't alter it. But the same can't be said for the left and right sub trees of the newly deleted node. So we check is the left and right children of the deleted node have a greater priority and if so we use rotations and this is done recursively until the order is restored or we reach a leaf node.

```
delete(PKTree, node):
    new_node = delete_BST(PKTree.root(), node)
    fix_priority(PKTree, new_node)
    return

fix_priority(tree, node):
    if (node.priority > node.left.priority)
        if (node.priority > node.right.priority)
            return
    else
        if (node.left.priority > node.right.priority)
            new_node = right_rotate(tree, node)
            fix_priority(tree, new_node)
        else
            new_node = left_rotate(tree, node)
            fix_priority(tree, new_node)
```

I have not listed the delete method in BST as it is fairly common algorithm and given in textbook.

(d) Analysis

Insertion in a BST takes log(n) time. Left and right rotations take constant time but in insert we start from the inserted node, which is always a leaf node and go towards the root. In the worst case we would have to rotate all the way till the top and this would mean that log(n) rotations would be required. Hence, the total runtime will also be a constant factor of log(n) and the time

complexity is $\Theta(log(n))$.

Deletion in a BST also takes time proportional to the height and assuming a balanced BST, we get the complexity as log(n). Here the rotations start from the deleted node and goes down towards the leaf node. In the worst case, we end up deleting the root and it might take log(n) rotations along the entire height. Hence, the total runtime will also be a constant factor of log(n) and the time complexity is $\Theta(log(n))$.

Searching is a well defined algorithm and will take log(n) time a fairly balanced BST. Hence, complexity is $\Theta(log(n))$.

Note that the analysis was done assuming that we have a fairly balanced BST. In case of a completely unblanced BST the complexities of insert and delete operations are bound by O(n) and our results change accordingly be all linear time.