Homework 1

Christian Che

1.

Code in C:

```c
#include <stdio.h>

//left is the starting index of the array
//right is the final index of the array (n - 1, where n is the number of
elements in the array)
int FindMaximum(int array[], int left, int right) {
  //base case (array has 1 element)
  if (left == right) {
    return array[left];
  }

  // cases when the array has 2 elements
  if ( right == left + 1) {
    if (array[left] > array[right]) {
      return array[left];
    }
    else { // if (array[left] < array[right]) {
      return array[right];
    }
  }

  int mid = (left + right) / 2;
  _Bool midIsBiggerThanLeftOfMid = (array[mid] > array[mid - 1]);
  _Bool midIsBiggerThanRightOfMid = (array[mid] > array[mid + 1]);

  if (midIsBiggerThanRightOfMid) {
    if (midIsBiggerThanLeftOfMid) {
      //if mid is bigger than both its left and right elements, then it is
the maximum
      return array[mid];
    }
    else { //the max is on the left side of mid
      return FindMaximum(array, left, mid-1);
    }
  }
  return FindMaximum(array, mid + 1, right);
}

int main() {
  // test array
  int array[] = {1,2,3,4,10,9,8,7,6,5,4,3,2,1};
  int n = sizeof(array) / sizeof(array[0]);
```

```
  printf("The maximum is: %d\n", FindMaximum(array, 0, n - 1));
}
```

Explanation in English:

The function have the following parameter and local variables:

Parameters	Description
`array`	the array A[1..n]
`left`	the left-most index of the array
`right`	the right-most index of the array

Variables	Description
`mid = (left + right) / 2`	index of the middle value
`midIsBiggerThanLeftOfMid`	check if the value at middle index is bigger than the value of the left index
`midIsBiggerThanRightOfMid`	check if the value at middle index is bigger than the value of the right index

This check the base case when the array only has 1 element. It returns the element as it is the maximum.

```
if (left == right) {
  return array[left];
}
```

This check the cases when the array only has 2 element. It returns the bigger value of the 2 elements.

```
if ( right == left + 1) {
  if (array[left] > array[right]) {
    return array[left];
  }
  else {
    return array[right];
  }
}
```

If the middle element is bigger than both its left and right elements, return the middle element

If mid is only bigger than the right element, the maximum must be on the left side of mid, so do a recursive call of the function with mid-1 as the right-most array index

Otherwise, the maximum must be on the right side of mid, so do a recursive call with mid + 1 as the left-most array index

```
if (midIsBiggerThanRightOfMid) {
  if (midIsBiggerThanLeftOfMid) {
    return array[mid];
  }
  else { //the max is on the left side of mid
    return FindMaximum(array, left, mid-1);
  }
}
return FindMaximum(array, mid + 1, right);
```

Time complexity analysis:

The algorithm has $O(log(n))$ complexity. There are no loop inside the function. There are only recursive call. And each time the function is call recursively, the search range for the maximum element is halved.

2a.

The total number of print statement (iterations of the inner loop) will be

$\frac{n}{1} + \frac{n}{2} + \frac{n}{4} + ... + \frac{n}{n}$

$= n * (\frac{1}{1} + \frac{1}{2} + \frac{1}{4} + ... + \frac{1}{n})$ <-- geometric series

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

$= n * log , n = \Theta(nlog , n)$

The overall time complexity of the algorithm is $\Theta(n log , n)$.

2b.

Here's some sample input and their outputs from my C implementation of the algorithm. Output is the number of times the print statement is called

Input	Output
1	1
2	3
4	9
8	24
16	61
32	145

Input	Output
64	337
128	765
256	1713
512	3782
1024	8275
2048	17973

The total number of print statement (iterations of the inner loop) will be

$\frac{n}{1} + \frac{n}{2} + \frac{n}{3} + ... + \frac{n}{n}$

$n * (\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n})$ <-- harmonic series, which can be written as $O(log , n)$ in big-O notation

$= n * O(log , n) = O(nlog , n)$

The overall time complexity of the algorithm is $O(n log , n)$.

3.

The function would have to following parameters and local variables:

Parameters	Description
`array`	the array A[1..n]
`n`	length of the array
`k`	the minimum number of occurences required

Variables	Description
`counter = 0`	return value
`frequencyHashMap`	<int, int> hashmap to count the frequency of each unique number

The function would have a **for** loop going through the array. If the number is not in the hashmap yet, the hashmap would create a new entry and increment it to 1. If the number is already in the hashmap, increment it by 1.

```
for(int i = 0, i < n, i++) {
    frequencyHashMap[array[i]] += 1;
}
```

The function would have another **for** loop going through the hashmap and count the number of elements that appear at least k times.

```
    for (const int& elements : frequencyHashMap) {
        if (elements.second >= k) {
            counter += 1;
        }
    }
```

And return the `counter` as the final step of the function.

```
    return counter;
```

Time complexity analysis:

This algorithm has the time complexity of O(n). There are two independent **for** loop in the function:

- The first **for** loop iterates over the array $\implies$ time complexity of $O(n)$
- The second **for** loop iterates over the hashmap. The worst case scenario is if the hashmap has all the elements from the array (all the elements in the array are unique) $\implies$ time complexity of $O(n)$

$O(n) + O(n) = O(2n) = O(n)$

4a. $f(n) \in \Theta(g(n))$ can be proven by proving $f(n) \in O(g(n))$ and $g(n) \in O(f(n))$

1. $f(n) \in O(g(n))$
   $\implies f(n) ; \leq ; c_{2} * g(n) \quad \forall n\geq , n_{0}$
   $3n^3 + n^2 ; \leq ; c_{2} * (n^3 - n^2) \quad \forall ; n\geq , n_{0}$
   $c_{2}=4$ and $n_{0}=1$ would satisfy the inequality
   $3n^3 + n^2 ; \leq ; 4 * (n^3 - n^2) \quad \forall ; n\geq , 1$

2. $g(n) \in O(f(n))$
   $\implies g(n) ; \leq ; c_{1} * f(n) \quad \forall n\geq , n_{0}$
   $(n^3 - n^2) ; \leq ; c_{1} * 3n^3 + n^2 \quad \forall ; n\geq , n_{0}$
   $c_{1}=2$ and $n_{0}=1$ would satisfy the inequality
   $(n^3 - n^2) ; \leq ; 2 * (3n^3 + n^2) \quad \forall ; n\geq , 1$

4b. If we simplified $f(n)$, we would get

$n^2 log^2 n + 4n^2 log n + 4n^2 + n log^2 n + 4n log n + 4n$

Since n^2log^2n is the most dominant term,

$f(n) \in O(g(n))$

$\implies f(n) ; \leq ; c*g(n) \quad \forall n\geq , n_{0}$

$(n^2+n)(log,n+2)^2; \leq ; c*n^2 log^2n \quad \forall ; n\geq , n_{0}$

$c=\frac{27}{2}$ and $n_{0}=2$ would satisfy the inequality

$(n^2+n)(log,n+2)^2 ; \leq ; \frac{27}{2} * n^2 log^2n \quad \forall ; n\geq ,2$

---

5. Ordered from the least to the highest time complexity

- $log(log(n))$
- $log^2(n)$
- $(log(n))^{log(n)}$
- $2^{log(n)} = n$
- $6nlog(n)$
- $n^{log(log(n))}$
- n^2
- 2^n
- e^n