1. Given two functions below

Table 1: Test1

```
Test1(a) :
1: n = a.length
2: key = bad_func(n^4)
3: print(key)
4: binary_search(a,key)
```

Table 2: Test2

```
Test2(a) :
1: for i = 1 :n
2:     if i < n / 10 :
3:       binary_search(a, key)
4:     else :
5:     linear_search(a, a[1])
```

Table 3: functions

1. What is the growth of the below functions?

   Test1 time complexity

$$\underbrace{O(1)}_{line\,1} + \underbrace{O((n^4)!)}_{line\,2} + \underbrace{O(1)}_{line\,3} + \underbrace{O(log_2(n))}_{line\,4} \quad = \quad O((n^4)!)$$

   Test2 time complexity

$$\underbrace{\sum_{i=1}^{\frac{n}{10}-1} O(log_2(n^3))}_{if\ statement\ operations\ cost} + \underbrace{\sum_{i=\frac{n}{10}}^{n} O(1)}_{else\ statement\ operation\ costs} = (\frac{n}{10}-1)\cdot O(3\cdot log_2(n)) + (n-\frac{n}{10}+1)\cdot O(1)$$

$$= (\frac{n}{10}-1)\cdot O(log_2(n)) + (n-\frac{n}{10}+1)\cdot O(1)$$
$$= O(n\cdot log_2(n)) + O(n)$$
$$= O(n\cdot log_2(n))$$

2. Compare the growth of Test1(n) and Test2(n).

   $f(n) = (n^4)!$ and $g(n) = n\cdot log_2(n)$

$$lim_{n\to\infty}(log_2(\frac{(n^4)!}{n\cdot log_2(n)})) = lim_{n\to\infty}log_2((n^4)!) - log_2(n\cdot log_2(n))$$
$$= lim_{n\to\infty}log_2((n^4)!) - log_2(n) - log_2(log_2(n))$$
$$= lim_{n\to\infty}\Theta(n^4\cdot log_2(n^4)) - log_2(n) - log_2(log_2(n))$$
$$= \infty$$
$$2^{\infty} = \infty$$

   We conclude that $f(n) = \omega(g(n))$ this implies $f(n) = \Omega(g(n))$. We can also conclude that $g(n) = O(f(n))$.

3. Let's say you can finish running Test(10^6) in 1 sec. Could you estimate when you finish running Test1(100)?

   Yes, we have $10^6 \cdot log_2(10^6) \cdot c = 1$ where c is the time for one single line for our machine. We solve for c to obtain

$$\begin{aligned} c &= \frac{1}{10^6 \cdot log_2(10^6)} \\ &= \frac{10^{-6}}{6 \cdot log_2(10)} \end{aligned}$$

   We use this c to obtain the running time for $Test1(n)$ when $n = 100$.

$$\begin{aligned} Test(100) \cdot c &= ((10^2)^4)! \cdot c \quad (Test(n) = O((n^4)!) \\ &= (10^8)! \cdot \frac{10^{-6}}{6 \cdot log_2(10)}. \end{aligned}$$

2. A sorted array and a random number are given to you. Develop an algorithm to find the total number of repetitions of the given number.

1. How would you find the total number of repetitions for the given number? Explain each solution/algorithm in a few lines.

   The brute-force solution is to have a variable that keeps tracks of the amount of times you have you seen the number and set it to 0. Iterate through the array and every time you encounter the number increment the value of the counter by 1. This is $O(n)$.

   The previous solution does not take advantage of the fact that the array is sorted. Specifically we use the divide-and-conquer paradigm to first find the lower bound that is the smallest index i such that $a[i] = num$. we then find the upper bound, that is the largest index j such that $a[j] = num$. The count of the number can then be expressed as $j - i + 1$ aka the interval between the lowest index and upper index. The recurrence will be of the form $T(n) = T(\frac{n}{2}) + O(1)$ which when solved will give $T(n) = O(log_2(n))$.

2. Write the pseduocode for the best algorithm you came up with.

Algorithm 1 count repetitions

prodedure(A,key) :

1: $lo := 0, hi := len(A) - 1$
2: $lower\_bound := -1$
3: $while\, lo \leq hi$:
4: $\quad mid := \lfloor \frac{lo+hi}{2} \rfloor$
5: $\quad if\, A[mid] < key$:
6: $\quad\quad\quad lo := mid + 1$
7: $\quad else\, if\, A[mid] > key$:
8: $\quad\quad\quad hi := mid - 1$
9: $\quad else$:
10: $\quad\quad\quad lower\_bound := mid$
11: $\quad\quad\quad hi := mid - 1$
12: $if(lower\_bound == -1)\, return\, 0$
13: $lo := 0, hi := len(A) - 1$
14: $upper\_bound := -1$
15: $while\, lo \leq hi$:
16: $\quad mid := \lfloor \frac{lo+hi}{2} \rfloor$
17: $\quad if\, A[mid] < key$:
18: $\quad\quad\quad lo := mid + 1$
19: $\quad else\, if\, A[mid] > key$:
20: $\quad\quad\quad hi := mid - 1$
21: $\quad else$:
22: $\quad\quad\quad upper\_bound := mid$
23: $\quad\quad\quad lo := mid + 1$
24: return upper_bound-lower_bound+1

3. Implement your answer using any programming language you want to.

```cpp
#include <iostream>
#include <vector>

using namespace std;

// T(n) = O(log_2(n))
int elegant_solution(const vector<int>& A, int key){
    int lower_bound = -1; // O(1)
    int lo = 0, hi = A.size() - 1;//O(1)
    // O(log_2(n)) in total for the while loop
    while(lo <= hi){
        int mid = (lo + hi) / 2;
        if(A[mid] < key) lo = mid + 1;
        else if(A[mid] > key) hi = mid - 1;
        else{
            lower_bound = mid;
            hi = mid - 1;
        }
    }
    // O(1) for the comparison
```

```
21      if(lower_bound == -1) return 0;
22
23      int upper_bound = -1; // O(1)
24      lo = 0, hi = A.size() - 1; // O(1)
25      // O(log_2(n)) in total for the while loop
26      while(lo <= hi){
27          int mid = (lo + hi) / 2;
28          if(A[mid] < key) lo = mid + 1;
29          else if(A[mid] > key) hi = mid - 1;
30          else {
31              upper_bound = mid;
32              lo = mid + 1;
33          }
34      }
35      return upper_bound - lower_bound + 1; // O(1)
36  }
37
38  int main(int argc, char* argv[]){
39      vector<int> a1 = {0, 1, 1,2, 2, 2, 3, 3, 6};
40      cout << elegant_solution(a1, 2) << endl;
41      vector<int> a2 = {0, 0, 2, 2, 3, 9, 10, 12, 15};
42      cout << elegant_solution(a2, 10) << endl;
43      vector<int> a3 = {0, 1, 3, 8, 12};
44      cout << elegant_solution(a3, 5) << endl;
45  }
```

4. What is the time complexity of your answer?

   The most number of operations are done in the while loops so that will be analyzed. The while loops are variations of the binary search algorithm, concretely the only statement that has changed is the case when $A[mid] == key$, everything else remains the same. In the case when we are looking for the lowerbound if we find an element that matches the key we pretend we have not found it, store the index and continue to search the left half of the array. Since this is similar to doing a binary search where the key could not be found we claim that order of the while loop is $O(log_2(n))$. A similar analysis is used to show that the order of the while loop to search for the upper bound is $O(log_2(n))$. The total work done is

$$\underbrace{O(1)}_{\text{operations not in while loops}} + \underbrace{O(log_2(n))}_{\text{finding lower bound}} + \underbrace{O(log_2(n))}_{\text{finding upper bound}} = O(log_2(n)).$$

3. A random array of size n is given to you. You know that the elements are nonnegative less than n. Develop an algorithm to find the mode and the numbers repeated more than once.

1. How would you find the mode and the numbers occuring more than once?

   I would use the portion of counting sort that creates a histogram. The reason being that we know that all the elements are non-negative and the max element is less than n. Once we have the histogram we can find the

max element in the histogram to find the modes. the modes values for the histogram will be the found element and thus we can insert it into the mode array. We can simultaneously insert elements e in the repeated array if $hist[e] > 1$. This does multiple passes through arrays of size n and thus the algorithm would run in $\Theta(n)$ time. The only downside to this algorithm is that we would have to allocate an array of size n which could be infeasible for large datasets.

2. Write the pseduocode for the best algorithm you cam up with

Algorithm 2 mode and repeated numbers

```
prodedure(A) :
hist = array(len(A) + 1)
for e in A:
        hist[e]++
modes = array()
numbers_repeated = array()
max_count = max(hist)
for e in A:
    if hist[e] == max_count:
        modes.insert(e)
    if hist[e] > 1:
        numbers_repeated.insert(e)
return {modes, numbers_repeated,hist}
```

3. Implement you answer using any programming language you want to

```
#include <iostream>
#include <vector>
#include <random>
#include <chrono>

using namespace std;

struct my_pair{
    vector<int> modes;
    vector<int> hist;
    vector<int> repeated;
};

bool get_line(const string& prompt, string& userinput){
    cout << prompt;
    getline(cin, userinput);
    return !userinput.empty();
}

// total runtime is O(n)
vector<int> histogram_generator(const vector<int>& A){
    vector<int> result(A.size() + 1);
    // O(n)
```

```
24      for(int e : A){
25          ++result[e];
26      }
27      return result;
28  }
29
30  // the total runtime is O(n)
31  my_pair find_mode_and_numbers_repeated_more_than_once(const
        vector<int>& A){
32      auto hist = histogram_generator(A); // O(n)
33      vector<int> modes;
34      vector<int> numbers_repeated_more_than_once;
35      int max_count = 0;
36      // O(n)
37      for(int i = 0;i < hist.size();i++){
38          max_count = max(max_count, hist[i]);
39      }
40      // O(n)
41      for(int i = 0;i < hist.size();i++){
42          if(hist[i] == max_count){
43              modes.emplace_back(i);
44          }
45
46          if(hist[i] > 1){
47              numbers_repeated_more_than_once.push_back(i);
48          }
49      }
50      return {modes, hist, numbers_repeated_more_than_once};
51  }
52
53  void display_arr(const vector<int>& arr){
54      for(int e : arr) cout << e << " ";
55  }
56
57  int main() {
58      string userinput;
59      mt19937 gen(chrono::system_clock::now().time_since_epoch()
            .count());
60      while (get_line("enter a positive integer n: ", userinput)
            ) {
61          int n = stoi(userinput);
62          uniform_int_distribution<int> uniform_int_distribution
                (0, n - 1);
63          vector<int> a;
64          for(int i = 0;i < n;i++){
65              a.push_back(uniform_int_distribution(gen));
66          }
67          cout << "a: [";
68          display_arr(a);
69          cout << "]" << endl;
70          auto the_pair =
                find_mode_and_numbers_repeated_more_than_once(a);
71          cout << "modes:";
72          for (const auto &mode : the_pair.modes) {
73              cout << " " << mode;
74          }
75          cout << endl;
```

```
76          for(const auto &repeated_number : the_pair.repeated){
77              cout << repeated_number << " was repeated " <<
                    the_pair.hist[repeated_number] << " times" <<
                    endl;
78          }
79      }
80  }
```

4. What is the time complexity of your answer?

   We analyze the pseudocode. The first line allocates an array of size $n+1$ and zeros out all the values this is $\Theta(n)$. The second to third line iterates through all the elements in A and count their frequency. This is $\Theta(n)$. Lines 4 and 5 create dynamic arrays, the time complexity we can consider $\Theta(1)$. Line 6 iterates through the hist variable to find the largest element, since hist is size $n+1$, this is $\Theta(n)$. Lines 7 to 11 add elements to the modes and number_repeated array if the conditions are satisfied, since this loop runs through the array A this is $\Theta(n)$. We can thus conclude that the overall running time of the algorithm is order n and hence the algorithm is $\Theta(n)$.