CptS 223 – Advanced Data Structures in C++

Written Homework Assignment 2: Big-O and Algorithms

I. Problem Set:

1

a. (10 pts) State the runtime complexity of both f() and g().

```
int f (int n)
{
        int sum = 0;                                  O(1)
       for(int i = 0; i < n; i++)                     O(n)
        {
                sum += 1;                             O(1)
        }
        return sum;                                   O(1)

}
```

∴ The time complexity = O(1 + n + 1 + 1) = O(n)

The runtime complexity of f(n) is O(n).

```
int g (int n)
  {
        if(n <= 0)                                            O(1)
        {
                return 0;                                     O(1)
         }
         return 1 + g(n - 1);                                 O(n-1)
  }
```

∴ The time complexity = O(1 + 1 + (n -1)) = O(n)

The runtime complexity of g(n) is O(n).

b. (10 pts) Write another function called "int h(int n)" that does the same thing, but is significantly faster.

```
int h(int n)
{

    return n;

}
```

Complexity = O(1)

2. **(15 pts)** State g(n)'s runtime complexity:

```
int f (int n)
{
        if(n <= 1)                          O(1)
       {
                return 1;                   O(1)
        }
        return 1 + f(n/2);                  O(logn )
}
```

Complexity f(n) = O(1 + 1 + n/2) = O(n)

```
int g(int n)

{
        for(int i = 1; i < n; i *= 2)       O(log n)
       {
                f(n);                       O(log n)
        }
}
```

Outer loop executes log n times, and inner function has the complexity O(n).
Hence complexity of g(n) = O(log n * log n) = O($\log^2$ n)

3. **(20 pts)** Write an algorithm to solve the following problem (10 pts)

Given a nonnegative integer n, what is the smallest value, k, such that

$$1n, 2n, 3n, \ldots, kn$$

contains all 10 decimal numbers (0 through 9) at least once? For example, given an input of "1", our sequence would be:

$$1*1, 2*1, 3*1, 4*1, 5*1, 6*1, 7*1, 8*1, 9*1, 10*1$$

and thus k would be 10.

We would use a frequency array of length ten to track which digits have occurred.
To get the digits of every number we would use %10 and divide the number by 10.
Assumptions made : n is the given number, k is the number to be found

The pseudo code is :

1. Declare an array of integers of size 10 with every element as zero. Every Index represents a digit and the value of the element is the frequency of the digit. Our goal is to run the while loop until every digit has a frequency greater than zero.
2. Set k as 0, set a while loop which runs until every element in array is greater than zero.

3. Inside the while loop increment k by 1 and multiply it with n. Call that number num
4. Run another while loop for which the terminating condition is that num == 0
5. Inside the while loop do num % 10. Increment the index of num % 10 by 1. Divide num by 10
6. Once all the elements of array are greater than 1, we will get the value of k which we want.

Implemented in functions :

```
#include <iostream>
bool arr_elements(int* arr)
{
for (int i = 0; i < 10; i++)
{
          if (arr[i] == 0)
         {
                   return false;
         }

}

return true;

}

int main()
{
    int arr[10] = { 0 };
    int n = 123456789;
    int k = 0,num  = 0;
    while (!arr_elements(arr))
    {
          k++;
          num = k * n;
          while (num != 0)
          {
                   arr[num % 10]++;
                   num = num / 10;

          }

    }

    cout << "The value of k is " << k;



}
```

The worst case time complexity would be O(kn)

4.

a. (3 pts) Determining whether a provided number is odd or even.

The time complexity would be O(1). Consider this function which determines whether a number is odd or even

```
bool number_is_odd(int k)
{
        if(k % 2 == 1)                      O(1)
        {
                return true;                O(1)
        }
        return false;                       O(1)
}
```

Complexity = O(1 + 1 + 1) = O(1)

b. (3 pts) Determining whether or not a number exists in a list.

Assumptions - List is unsorted and is stored in an array
The time complexity will be O(n)
Pseudo code :
//arr is the array name n is the array size and num is the number whose presence we have to confirm in the array

```
for(int i = 0; i < n; i++)                                  O(n)
{
        if(arr[i] == num)                                   O(1)
        {
                //NUM FOUND IN LIST
                //exit loop
        }
}
```

The worst case complexity would be when the number does not exist in the list and we have to traverse the entire list.

Complexity = O(n + 1) = O(n)

c. (3 pts) Finding the smallest number in a list.

Assumptions - List is unsorted and is stored in an array
The time complexity will be O(n)
Pseudo code :
//arr is the array name n is the array size and min is the minimum number in the list

```
int min = arr[0]                                    O(1)

for(int i = 1; i < n; i++)                          O(n)
{

        if(min > arr[i])                            O(1)
        {
                min = arr[i];

        }
}
```

We have to traverse the entire array to determine the minimum
Complexity = O(1 + n + 1) = O(n)

d. (4 pts) Determining whether or not two **unsorted** lists of the same length contain all of the same values (assume no duplicate values).

There are multiple ways to solve this. One approach would be to first sort both the lists and the compare corresponding elements. Let us look at the time complexity for that.
Assumptions - Lists are unsorted and stored in arrays
Suppose we use bubble sort to sort the two arrays. The time complexity for that is O(n^2) for each array.
After we have the sorted arrays,
We would traverse the array to check whether corresponding elements are the same.

Suppose the two sorted arrays are arr1 and arr2 and their size is n

```
for(int i = 0; i < n; i++)                                  O(n)
{

    if(arr1[i] != arr2[i])                                  O(1)
    {
        // exit the loop arrays do not contain the same elements.
    }

}
```

The time complexity of the entire algorithm would be
Complexity = O(n^2 + n^2 + n + 1) = O(n^2)

e. (4 pts) Determining whether or not two **sorted** lists contain all of the same values (assume no duplicate values).

Assumptions - Lists are sorted and stored in arrays
The first step would be to check whether both lists have the same number of elements. If they have the same number of elements, the next step would be to check if every corresponding element is equal.
Suppose the two sorted arrays are arr1 and arr2

```
int size1 = sizeof arr1 / sizeof arr1[0];                O(1)
int size2 = sizeof arr2 / sizeof arr2[0];                O(1)
if(size1 == size2)
{
        for(int i = 0; i < n; i++)                       O(n)
        {

                if(arr1[i] != arr2[i])                   O(1)
                {
                        //exit the loop arrays do not contain the same elements.
                }

        }
}
```

Complexity = O(1 + 1 + n + 1) = O(n)
Therefore the time complexity is O(n)

f. (3 pts) Determining whether a number is in a balanced BST.

Suppose num is the number we're looking for, root is the pointer to the root node of the balanced BST. The function returns NULL if the number isn't in the bst or returns a node pointer if the number does exist in the BST
The pseudo code would be -

```
BSTNODE *  find(BSTNODE * ptr, int data)
{

        if (ptr == NULL || ptr->data == num)                   O(1)
        {
                return root;
        }
        if (ptr->data < num )                                   O(1)
        {
                return search(ptr->right, num);                 O(log n)
        }

        return search(ptr->left, key);                          O(log n)

    }
```

Complexity = O(1+ 1+ log n + log n) = O(log n)

5. **(25 pts)** Write a pseudocode or C++ algorithm to determine if a string `s1` is an *anagram* of another string `s2`. If possible, the time complexity of the algorithm should be in the worst case O(n). For example, 'abc' - 'cba', 'cat' - 'act'. s1 and s2 could be arbitrarily long. It only contains lowercase letters a-z. Hint: the use of histogram/*frequency* tables would be helpful!

The logic behind the program is that we create two frequency arrays for both our strings and map which letters occur how many times. We then compare the two frequency arrays to see whether the two strings are anagrams of each other.
The C++ program is -

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
   string s1 = "cataa";                                     O(1)
   string s2 = "atcaa";                                     O(1)
   if (s1.length() == s2.length())                          O(1)
   {
         int arr1[26] = { 0 };                              O(1)
         int arr2[26] = { 0 };                              O(1)
         for (int i = 0; i < s1.length(); i++)              O(n)
         {
                 arr1[s1[i] - 'a']++;
                 arr2[s2[i] - 'a']++;
         }
         bool check = true;
         for (int i = 0; i < 26; i++)                       O(26) -> O(1)
         {
                 if (arr1[i] != arr2[i])                    O(1)
                 {
                         check = false;                     O(1)
                 }
         }

         if (check == true)                                 O(1)
```

```
          {
                    cout << "\n Strings are an anagram of each other";   O(1)
          }
          else                                                          O(1)
          {
                    cout << "\n Strings are not an anagram of each other"; O(1)
          }
   }
   else                                                                 O(1)
   {
          cout << "\n Strings are not an anagram of each other"; O(1)
   }




}
```

Hence the time complexity of the program is O(n)