CptS 223 – Advanced Data Structures in C++

Written Homework Assignment 2: Big-O and Algorithms

I. Problem Set:

1. **(20 pts)** Given the following two functions which perform the same task:

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

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

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

The runtime complexity of g() is O(n).
The runtime complexity of f() 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){
        if(n<=0){
                return 0;
        }
        else{
                return n;
        }
}
```

// The time complexity of h() is O(1), because there are no loops, only simple comparison
//statements. Therefore it is much faster than g() and f().

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

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

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

The runtime complexity of g(n) is O((log (n))^2). This is because there are the for loop has a complexity of O(log (N)), and since the loop calls a function f(n) that also has a time complexity of O(log (N)), the result is O((log (n))^2).

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, …, 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. Other examples:

Integer Value	K value
10	9
123456789	3
3141592	5

```
#include <iostream>
 int allDigits(int n) {
        int a = 1;
        if (n != 0) {
                int arr[10] = { 0 };
                int flag = 0;
                while (flag == 0) {
```

```
                int size = 0;
                int temp = n;
                while (temp > 0) {
                        temp /= 10;
                        size++;
                }

                int arr_temp[1] = { 0 };
                int count = 0;
                int m = n;
                while (size>0) {
                        arr_temp[0] = m%10;
                        int b = arr_temp[0];
                        arr[b]++;
                        m = m / 10;
                        size--;
                        count++;
                }

                flag = 1;
                for (int i = 0; i < 10; i++) {
                        if (arr[i] == 0)
                                flag = 0;
                }

                if (flag == 0) {
                        n = n / a;
                        a++;
                        n = n * a;
                }
        }

}
else {
        return -1;
}
return a;
}
```

(10 pts). Can you directly formalize the worst case time complexity of this algorithm? If not, why?

The worst-case time complexity of this algorithm is O(log n), because the loops both use the number of digits (which is fine by taking log base 10 of n, or repeatedly dividing n by 10) and so therefore the worst case complexity is O(log n).

4. **(20 pts)** Provide the algorithmic efficiency for the following tasks. Justify your answer, often with a small piece of pseudocode to help with your analysis.

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

      This algorithm has an efficiency of O(1). This is because, as can be seen from the pseudocode below, all of the operations are simple comparisons or return statements (no loops) so the total complexity is O(4) which is equivalent to O(1).

```
//note return true means even, return false means odd

bool is_even(int a){

//int a; -> O(1)

//if(a%2==0), return true ->O(1)+O(1)

//else return false -> O(1)

}
```

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

The time complexity of this algorithm would be O(n). This is because if the size of the list to look through changes, the algorithm might have to spend longer looking for n. Therefore, since it is dependent on the size, the complexity is O(n).

```
bool is_in_list(int num, int list[], int size){

    //for(int i = 0; i<size; i++){ -> O(n)

        //if(list[i]==a), return true ->O(1)+O(1)

    }

    //return false -> O(1)

}
```

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

The time complexity of this algorithm would be O(n). This is because if the size of the list changes, the algorithm has to make more comparisons, so it is dependent on the size of the list, with a complexity of O(n).

```
    int find_min(int num, int list[], int size){

    //int min defined as the first list element, complexity O(1)

    //for loop to compare the rest of the elements with min, complexity O(n)

        //if the element is less than min, min = element, complexity O(1)

    //return min, complexity O(1)

}
```

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

The time complexity of this algorithm would be O(n^2). This is because since there are two unsorted loops, there would need to be a nested for loop, which would have a complexity of O(n)*O(n) = O(n^2).

```
bool list_compare (int size, int list1[], int list2[]){
        //for(int i = 0; i<size; i++){                    ->complexity O(n)
                //set a flag integer as 0
                //for(int j = 0; i<size; j++){        ->also O(n)
                        //compare list1[i] with list2[j] ->complexity O(1)
                        //update flag value              -> complexity O(1)
                }
                //if flag value is 0 return false; -> complexity O(1)
        }
        //return true -> complexity O(1)
}
```

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

The time complexity of this algorithm would be O(n). This is because since the list is sorted, there only needs to be one comparison done on each pair of elements in the lists, as seen in the pseudocode/sample function below. Therefore the size of the lists have a linear change on the time complexity, so it is O(n).

```
bool same_val( int size, int list1[], int list2[]){
        for(int i = 0; i<size; i++){          //complexity O(n)
                if(list1[i]!=list2[i]          //complexity O(1)
                        return false;          //complexity O(1)
        }
        return true;                           //complexity O(1)
}
```

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

The time complexity of this algorithm would be O(log n). This is because the binary search for a balanced tree means that there are two options at every node (other than the leaves), so with each comparison the size of the tree for the search is halved. Therefore the time complexity is O(log n).

```
//time complexity is O(log n)
bool is_in_tree(Node* treeNode, int num){
//while(treeNode->data!=num)
//compare num to treenode
        // if num is greater than treeNode->data then treeNode = Node-> right
        //if it's less treeNode = Node->left
        //if num= treeNode->data, return true

    //if treeNode has no children, return false

    //end while

    }
```

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 time complexity of this algorithm is O(n), based on the size of the strings
//return true = is anagram, return false = is not anagram

```
bool is_anagram(char* s1, char* s2, int arr1[26], int arr2[26]) {
       int s1_size = strlen(str1);
       int s2_size = strlen(str2);
       int temp = 0;

       for (int i = 0; i < str1_size; i++) {
              temp = s1[i];
              arr1[temp - 97]++;
       }
       for (int j = 0; j < str2_size; j++) {
              temp = s2[j];
              arr2[temp - 97]++;
       }
       for (int k = 0; k < 26; k++) {
              if (arr1[k] != arr2[k]) {
                     return false;
              }
       }
       return true;
}
```

II. Submitting Written Homework Assignments:

1. On your local file system, create a new directory called HW2. Move your HW2.pdf file into the directory. In your local Git repo, create a new branch called HW2. Add your HW2 directory to the branch, commit, and push to your private GitHub repo created in PA1.
2. Do not push new commits to the branch after you submit your link to Canvas otherwise it might be considered as late submission.
3. Submission: You must submit a URL link of the branch of your private GitHub repository to Canvas.