WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

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().

g(n) = g(n-1)
g(n) = O(n)
	f(n) = 1 + 2(n+1) + 2n + 1
	= 4n + 4
	f(n) = O(n)

g() is a recursive function that runs (n) amount of times. O(n)

f() is a loop function that also runs (n) times. The time complexity is O(n).

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

O(1)

```
int h (int n) {
        return n;
}
```

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);
  }
}
```

g() uses a for loop with each iteration multiplying the index by 2. The time it takes to check if (n <=1) doesn't run (n) times so it is irrelevant to time complexity. Halving the time it takes to run the loop by doubling the iterator is O(log n) run time.

f() uses a recursive with each iteration halving the index by 2. Halving the index reduces the time to run the loop by 2 resulting in O(log n) time complexity.

Since the functions are nested, the time complexities are multiplied. O(log n) * O(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, …, kn

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

and thus k would be 10. $1*1, 2*1, 3*1, 4*1, 5*1, 6*1, 7*1, 8*1, 9*1, 10*1$

Other examples:

Integer Value	K value
10	9
123456789	3
3141592	5

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
     int n;      // user entered value
     int k = 0; // number of processes though loop
     bool found[10]; // conditional array for finding each digit

     // Initialize with false values
     for (int i = 0; i < 10; i++)
         found[i] = false;

     // get n from user
     cout << "Enter n:\n";
     cin >> n;

     // repeat until each digit is found
     while (!found[0] || !found[1] || !found[2] || !found[3] || !found[4] ||
!found[5] || !found[6] || !found[7] || !found[8] ||
            !found[9])
     {
         k++;

          // set z to n multiplied by k
          int z = n*k;

          // check if
          while (z != 0)
          {
              // convert R to string for parsing
              string str = to_string(z);

              // i is used to parse z and find individual digits
              int i = str.length() - 1;

              // set j to each digit individually starting from the right
              int j = stoi(str.substr(i,str.length()));

              // if digit is found, update found array
              found[j] = true;

              z = z / 10; // remove digits by dividing by 10
          }
     }

     cout << "k = " << k << endl;
          return 0;
}
```

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

Yes, the worst-case time complexity for this algorithm is O(log n) due to the while loop dividing z by 10 each iteration.

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.

   The algorithmic efficiency of determining if provided number is even or odd consists of a single arithmetic operation. The conditional is (n % 2).
   O(1)

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

   Depends on if sorted or not.
   Sorted = O(N), loops n times though array until conditional is met.
   Unsorted = O(log(n), binary search splits the array until conditional is met.

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

   Depends on if sorted or not.
   Sorted = O(1), the smallest number is the first number in the array.
   Unsorted = O(n) loops n times through the array until the conditional is met.

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

   O(n^2), finding each a value in a single table is O(n), doing it twice is O(n^2)
   O(n) if there are frequency tables used. The search is linear.

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

   O(n), the values are sorted so the algorithm runs n times until the conditional is met.

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

   Depends if the BST is sorted.
   Sorted is O(log(n)), the tree traverses each branch until the node with the number is found, splitting the chances in half each time.
   Unsorted is O(n), the worst case would be an array, searching until the conditional is met.

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!

1. Get lengths of both s1 and s2.
2. If s1 length does not equal s2 length, then the strings are not anagrams. Time complexity=O(1)
3. Create two frequency arrays of size 256 to store ascii values in the strings.
4. Parse each frequency array and set each element in the tables to be 0.
5. Parse each string to place each character in the ascii frequency array.
6. Parse the frequency arrays for both strings to compare if each element is equal to each other. That is, each frequency where there is a 1 for string 1 matches the location of each 1 for string 2. The algorithms are anagrams, O(n) is the time complexity.

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.