Name: Jude Onyia

Student ID: V00947095

Course: ECE 596C

Due Date: May 22, 2020

Assignment 1: Non – Programming Exercise

8.8 a)

If the tree is balanced and we assume worst case, the asymptotic time complexity of the function is the height of the balanced tree, which is **O(log n)**.

8.8 b)

If the tree is not balanced, assuming worst case of the search for a node with the value and worst case of the imbalance of the tree, the asymptotic time complexity is **O(n)**.

8.9 a)

The source code performs a sequential accumulative sum of the lower triangle of the matrix. From inspecting the source code, it is evident that the elements included in the accumulation consist of half of the matrix excluding the primary diagonal elements (i.e. a(0,0), a(1,1), etc.), plus the primary diagonal elements. Since the code loops over these elements, the asymptotic time complexity is $O(\frac{n^2-n}{2}+n)$, this can be reduced to $\mathbf{O(n^2)}$.

8.9 b)

Since the allocation of memory for the variables created in this function are not dependent on n, assuming the maximum value of type int is greater than n, then the asymptotic space complexity of the function is **O(1)**.

8.10 a)

The asymptotic time complexity of reverse_array_1 is $O(\frac{n}{2})$, this can be reduced to **O(n)**. Assuming the maximum value of type int is greater than n, the asymptotic space complexity is **O(1)**.

8.10 b)

The asymptotic time complexity of reverse_array_2 is **O(n)**. The space complexity is **O(n)** because a vector of size n is created. The assumption here is also that the maximum value of type int is great than n.

8.10 c)

Based on asymptotic complexity analysis, both have the same time complexity, however, reverse_array_1 has a space complexity of O(1) while reverse_array_2 has O(n). Therefore, reverse_array_1 is preferable.

8.12)

We would need to calculate the overall speedup of the program when each of the three parts are optimized.

A) If part A is optimized, the overall speedup of the program is:

$$S_o = \frac{1}{(1 - f_e) + \frac{f_e}{S_e}} = \frac{1}{(1 - 0.05) + \frac{0.05}{10}} = 1.0471$$

B) If part B is optimized, the overall speedup of the program is:

$$S_o = \frac{1}{(1 - f_e) + \frac{f_e}{S_e}} = \frac{1}{(1 - 0.5) + \frac{0.5}{1.05}} = 1.0244$$

C) If part C is optimized, the overall speedup of the program is:

$$S_o = \frac{1}{(1 - f_e) + \frac{f_e}{S_e}} = \frac{1}{(1 - 0.1) + \frac{0.1}{3}} = 1.0714$$

Based on the above calculations, the choice that would yield the most speedup is optimizing part C, therefore, part C should be optimized.

8.13 a)

If we assume the worst case of all bits having the value 1 (or even just the most significant bit having the value 1), the while loops will iterate until the most significant bit of value 1 has been checked. Hence, it will iterate for the bit-length of the integer. The number of bits of the integer is $\log_2(n)$, rounded up. Therefore, the asymptotic time complexity is **O(log n)**. The asymptotic space complexity is **O(1)** because if the number of bits used for n is changed, the only memory affected is that of n.

8.13 b)

The code below is an implementation of the algorithm derived from [1].

```
unsigned int hamming_2(unsigned int n){
   unsigned int total_bit_num = sizeof(int) * CHAR_BITS; // Number of bits in n
   unsigned int partition_1 = (~(unsigned int)0) / 3; // Binary 01010101
   unsigned int partition_2 = (~(unsigned int)0) / 5; // Binary 00110011
   unsigned int partition_4 = (~(unsigned int)0) / 17; // Binary 00001111

   n -= (n >> 1) & partition_1;  //Count the ones of each 2 bits and
                                 //replace those 2 bits with result
```

```
    n = (n & partition_2) + ((n >> 2) & partition_2); //Count the ones of each 4 bits
                                                      //and replace those 4 bits with result
    n = (n + (n >> 4)) & partition_4; // Count the ones of each 8 bits and
                                      //replace those 8 bits with result

    if(total_bit_num > 8)  n += n >>  8; //move result of each 16 bits into lowest 8 bits
    if(total_bit_num > 16) n += n >> 16; //move result of each 32 bits into lowest 8 bits
    if(total_bit_num > 32) n += n >> 32; //move result of each 64 bits into lowest 8 bits
    return n & 0x7F; // bit AND with decimal number 127 will keep the 8 bits
}
```

The advantage of the algorithm is that it's asymptotic time complexity is **O(1)**, less than hamming_1's complexity of O(log n). The disadvantage is that it requires more space in memory than hamming_1.

8.13 c)

The reasoning behind using asymptotic complexity is to have a sense of the effect of problem size on the performance of the program as the problem size increases to relatively huge amount. The asymptotic analysis is necessary to calculate the rate of program's performance and memory requirement as the problem size increases.

Reference

[1] Joel Yliluoma, WP2 - Nifty Revised, without multipliations, Bit-counting algorithms, 2013. https://bisqwit.iki.fi/source/misc/bitcounting/