Assignment 1 [Assignment ID:cpp_basics]

Part A

8.16

8.16 A binary tree is used to represent a set of integers. The partial source code for this tree type is shown below. A node in the tree is represented using the `Node` type. A function `find_element` is provided. This function finds a node that holds a specified value `v` by searching in a given node `n` and all of its descendants. If a node containing the desired value is found, a pointer to the node is returned; otherwise, a null pointer is returned. (To search the entire tree, the value of `n` would simply be chosen as the root node of the tree.)

```
1  // Tree node type.
2  struct Node {
3     Node* parent; // pointer to parent
4     Node* left; // pointer to left child
5     Node* right; // pointer to right child
6     int value; // value
7  };
8
9  // Find tree node that contains specified element.
10 Node* find_element(Node* n, int v) {
11    if (v == n->value) {
12       // We have found the desired element.
13       return n;
14    } else if (v < n->value) {
15       // The element if present must be in the left subtree.
16       return n->left ? find_element(n->left, v) : nullptr;
17    } else {
18       // The element if present must be in the right subtree.
19       return n->right ? find_element(n->right, v) : nullptr;
20    }
21 }
```

(a) If the tree is balanced and contains n nodes, determine the asymptotic time complexity of `find_element` (as a function of n). (A tree is balanced if the depth of each leaf node is $O(\log n)$.)

(b) Determine the time complexity of the function if the tree contains n nodes but is not balanced.

[Hint: In both parts of this exercise, simply consider the number of nodes in the tree that must be visited in the worst case.]

(a)

$$f(n) = \begin{cases} b + f(n/2) & \text{if } n \geq 2 \\ b & \text{if } n = 1 \end{cases}$$

$f(n) \in O(\log n)$

(b)

Since the tree is not balanced which not guarantee each leaf node is $O(\log n)$, therefore $f(n) \in O(n)$

8.17

8.17 Consider the function `sum_lower_triangle` whose source code is given below. This function sums the elements in the lower triangular part of an $n \times n$ matrix and returns the result.

(a) Determine the asymptotic time complexity of this function (where the problem size is n).

(b) Determine the asymptotic space complexity of this function (where the problem size is n).

```
template <int n, class T>
T sum_lower_triangle(const T (&a)[n][n]) {
    T sum(0);
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j <= i; ++j) {
            sum += a[i][j];
        }
    }
    return sum;
}
```

(a)

$$f(n) = 1 + 3n + (3 + 2) \times \frac{(1+n)n}{2}$$

$$f(n) = 1 + 3n + \frac{(5n+5n^2)}{2} \in O(n^2)$$

(b)

Let C_T be the space occupied by type T.

$$f(n) = n^2 \times C_T + 3C_T \in O(n^2)$$

8.18

8.18 Consider the two functions `reverse_array_1` and `reverse_array_2` whose source code is given below. Each of these functions reverses the elements in a one-dimensional array.

```
#include <utility>

template <class T>
void reverse_array_1(T* a, int n)
{
    for (int i = 0; i < n / 2; ++i) {
        std::swap(a[i], a[n - 1 - i]);
    }
}
```

```
#include <vector>

template <class T>
void reverse_array_2(T* a, int n)
{
    // Copy the array elements into a vector.
    std::vector<T> v(&a[0], &a[n]);
    // Copy the elements from the vector to the array in reverse order.
    for (int i = 0; i < n; ++i) {
        a[i] = v[n - 1 - i];
    }
}
```

(a) Find the asymptotic time and asymptotic space complexities of `reverse_array_1`. State any assumptions made in your answer.

(b) Find the asymptotic time and asymptotic space complexities of `reverse_array_2`. State any assumptions made in your answer.

(c) Which function would be preferable to use, based on asymptotic complexity analysis? Explain your answer.

(a)

Time complexity: $f(n) = 1 + 3\frac{n}{2} \in O(n)$

Let C_T be the space occupied by type T, and integer type occupies 4 bytes. Assume there is a temporary variable in `std::swap`.

Space complexity: $f(n) = n \times C_T + \frac{n}{2}C_T + 4 + 4 \in O(n)$

(b)

Time complexity: $f(n) = n + 1 + 5n \in O(n)$

Let C_T be the space occupied by type T, and integer type occupies 4 bytes.

Space complexity: $f(n) = 2n \times C_T + 4 + 4 \in O(n)$

(c)

Based on the asymptotic complexity of both functions, even though they are bounded in O(n), the `reverse_array_1` seems more preferable with the less space being use ($\frac{n}{2}C_T$).

8.21

8.21 The Hamming weight of an integer is the number of one digits in the binary representation of the number. Consider the following function for computing the Hamming weight of an integer.

```
unsigned int hamming_1(unsigned int x)
{
    unsigned int result = 0;
    while (x != 0) {
        // Is the least significant bit nonzero?
        if ((x & 1) != 0) {
            // One more nonzero bit has been found.
            ++result;
        }
        // Shift the bits in the value right by one position.
        x >>= 1;
    }
    return result;
}
```

(a) Determine the asymptotic time and asymptotic space complexities of the `hamming_1` function, where the input problem size is the number *n* of bits in the integer whose Hamming weight is to be computed.

(b) As it turns out, the Hamming weight of an integer can be computed with a lower asymptotic time complexity than that achieved by `hamming_1`. Write a function `hamming_2` that implements such an algorithm. Identify the advantages and disadvantages of the new algorithm (used by `hamming_2`) relative to the original one (used by `hamming_1`).

(c) Comment on the reasonableness of using asymptotic analysis in situations like the one in this exercise.

(a)

Time complexity: $f(n) = 1 + 5n + 1 \in O(n)$

Assume unsigned integer type occupies 4 bytes.

Space complexity: $f(n) = 4 + 4 \in O(1)$

(b)

// reference from http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetParallel

unsigned int hamming_2(unsigned int x){

```
unsigned int result;

x = x - ((x >> 1) & 0x55555555);
x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
result = ((x + (x >> 4) & 0xF0F0F0F) * 0x1010101) >> 24;

return result;
```

}

Advantage: time complexity is O(1).

Disadvantage: only works for 32-bits integer.

(c)

In this case, since the size of the n bits in the integer are relatively small, it may not be a good idea to use asymptotic analysis in this exercise.

8.23

8.23 Three parts A, B, C of a program have been identified as potential bottlenecks. Measurements have shown that the percentage of time spent in each of these parts of the code is as follows:

Part	Fraction of Time
A	5%
B	50%
C	10%

It is believed that through additional optimization the speedup achievable for each part of the code is as follows:

Part	Speedup Factor
A	10
B	1.05
C	3

There is only enough time before the next product release to optimize one of the three parts of the code. Which one should by optimized? Justify your answer.

Amdahl's Law: $S_O = \frac{1}{(1-f_e)+\frac{f_e}{S_e}}$

A: $S_O = \frac{1}{(1-0.05)+\frac{0.05}{10}} \approx 1.04712$

B: $S_O = \frac{1}{(1-0.5)+\frac{0.5}{1.05}} \approx 1.02439$

C: $S_O = \frac{1}{(1-0.1)+\frac{0.1}{3}} \approx 1.07142$

Based on the result, part C should be optimized.