CSCE 221 Cover Page
Homework 2
Due March 23 at midnight to eCampus

First Name: Cristian Last Name: Avalos UIN: 627003137

Username: avalos672918 E-mail address: avalos672918@tamu.edu

Please list all sources in the table below including web pages which you used to solve or implement the current homework. If you fail to cite sources you can get a lower number of points or even zero, read more: Aggie Honor System Office

| | | | |
|---|---|---|---|
| Type of Sources | | | |
| People | Friends (discussion) | | |
| Web pages (provide URL) | http://www.cplusplus.com/reference/list/list/ | | |
| Printed material | | | |
| Other Sources | | | |

I certify that I have listed all the sources that I used to develop the solutions/codes to the submitted work.

"On my honor as an Aggie, I have neither given nor received any unauthorized help on this academic work."

Cristian Avalos March 23, 2020

1. (20 points) Given two sorted lists, $L1$ and $L2$, write an efficient C++ code to compute $L1 \cap L2$ using only the basic STL list operations.

   (a) Provide evidence of testing: submit your code.

```cpp
#include <iostream>
#include <list>
using namespace std;

list<int> intersection(list<int> l1, list<int> l2) {
    list<int> l3;
    list<int>::iterator l1pos, l2pos;
    l1pos = l1.begin();
    l2pos = l2.begin();
    while (l1pos != l1.end() && l2pos != l2.end()) {
        if (*l1pos < *l2pos) {
            l1pos++;
        }
        else if (*l1pos > *l2pos) {
            l2pos++;
        }
        else {
            l3.push_back(*l1pos);
            l1pos++;
            l2pos++;
        }
    }
    return l3;

};

int main() {
    list<int> l1, l2, intersect;
    for (int i = 0; i < 11; i++) {
        l1.push_back(i);
        if (i % 2 == 0) {
            l2.push_back(i);
        }
        if (i % 3 == 0) { //for more numbers and repitition
            l2.push_back(i);
        }
    }
    cout << "list1: ";
    for (auto v : l1) {
        cout << v << " ";
    }
    cout << endl;
    cout << "list2: ";
    for (auto v : l2) {
        cout << v << " ";
    }
    cout << endl;
    intersect = intersection(l1, l2);
    cout << "intersection: ";
    for (auto v : intersect) {
        cout << v << " ";
    }
    return 0;
}
```

Figure 1: My code for problem 1

   (b) What is the running time of your algorithm?
   The run time of my algorithm is $O(n)$.

2. (20 points) Write a C++ recursive function that counts the number of nodes in a singly linked list.

   (a) Test your function using different singly linked lists. Include your code.

```cpp
#include <iostream>
using namespace std;

struct sllnode {
    int data;
    sllnode *next;
};

void addVal(sllnode** first, int new_data) {
    sllnode *temp = new sllnode;
    temp->data = new_data;
    temp->next = *first;
    *first = temp;
};

int getCount(sllnode* head) {
    if (head == nullptr) {
        return 0;
    }
    return 1 + getCount(head->next);
};

int main()
{
    sllnode *list;
    for (int i = 0; i < 10; i++) {
        addVal(&list, i);
    }
    int count = getCount(list);
    cout << "Number of nodes in the singly linked list is " << count << endl;
    return 0;
}
```

Figure 2: My code for problem 2

   (b) Write a recurrence relation that represents your algorithm.
   $sll(n) = sll(n+1) + O(1)$

(c) Solve the recurrence relation using the iterating or recursive tree method to obtain the running time of the algorithm in Big-O notation.

$sll(n) = sll(n+1) + O(1)$

$sll(1) = 0, head == nullprt$

$sll(n+1) = sll(n+1) + O(1)$

$sll(n+2) = sll(n+2) + 2O(1)$

step $k = sll(n+k) + kO(1)$

$k_{max} = n + k = 0$

$k = n$

$sll(n+n) + nO(1)$

$= O(n)$

3. (20 points) Write a C++ recursive function that finds the maximum value in an array (or vector) of integers without using any loops.

   (a) Test your function using different input arrays. Include the code.

```cpp
#include <iostream>
#include <stdexcept>
using namespace std;

struct Empty : public std::runtime_error {
  explicit Empty(char const* msg=nullptr): runtime_error(msg) {}
};

int findMax(int array[], int values) {
    if (values == 0) {
        throw Empty("Empty");
    }
    if (values == 1) {
        return array[0];
    }
    return max(array[values - 1], findMax(array, values - 1));
}

int main() {
    int array[] = {0, 1, 2, 3, 4, 103295, -5, -6, 7, 8, 134, 3, 2903, 2493};
    int array2[] = {};
    int count = 0;
    for (auto v : array) {
        count++;
    }
    try {
        int max = findMax(array, count);
        cout << "The maximum value in the array is " << max << endl;
    }
    catch (exception &test) {
        cout << test.what() << endl;
    }
    return 0;
}
```

Figure 3: My code for problem 3

   (b) Write a recurrence relation that represents your algorithm.
   $fm(n) = fm(n-1) + O(1)$

(c) Solve the recurrence relation and obtain the running time of the algorithm in Big-O notation.

$fm(n) = fm(n-1) + O(1)$

$fm(1) = 0$

$fm(n-1) = fm(n-21) + O(1)$

$fm(n-2) = fm(n-3) + 2O(1)$

step $k = fm(n-k) + kO(1)$

$k_{max} = n - k = 0$

$k = n$

$fm(n-n) + nO(1)$

$= O(n)$

4. (20 points) What is the best, worst and average running time of quick sort algorithm? The best and average case of quick sort algorithm is $O(nlog(n))$ the worst case is $O(n^2)$.

(a) Provide recurrence relations and their solutions.
Worst Case:
$t(n) = t(n-1) + n$
$t(1) = 0$
$t(n-1) = t(n-2) + n$
$t(n-2) = t(n-3) + 2n$
step $k = t(n-k) + kn$
$k_{max} = n - k + kn = 0$
$t(n-n) + n^2$
$= O(n^2)$

Best Case (and average): $t(n) = 2t(n/2) + n$
$t(1) = 0, sorted$
$t(n/2) = 2t(n/4) + n/2$
$t(n) = 2(2t(n/2^2) + n/2) + n$
$= 4t(n/2^2) + 2n$
$t(n/4) = 2t(n/8) + n/4$
$t(n) = 8t(n/8) + 3n$
$= 2^3 t(n/2^3) + 3n$
$= 2^k t(n/2^k) + kn$
$k_{max} = k = log_2(n)$
$2^{log_n(n)} t(n/2_+^{log_2(n)} nlog_2(n)$
$= O(nlogn)$

(b) Provide arrangement of the input and the selection of the pivot point for each case.
Worst Case: input: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
pivot: 1

Best Case (and average): input: 1, 9, 3, 5, 2, 6, 10, 4, 8, 7
pivot: 2

5. (20 points) Write a C++ function that counts the total number of nodes with two children in a binary tree (do not count nodes with one or none child). You can use a STL container if you need to use an additional data structure to solve this problem. Use the big-O notation to classify your algorithm. Include your code.

```cpp
#include <iostream>
#include <queue>
using namespace std;

struct node {
    int data;
    node *left, *right;
    node(int data) {
        this->right = nullptr;
        this->left = nullptr;
        this->data = data;
    }
};

int getNodeCount(node* root) {
    if (!root) { //if tree is empty
        return 0;
    }
    int count = 0;
    if ((root->left != nullptr) && (root->right != nullptr)) {
        count++;
        return count + getNodeCount(root->left) + getNodeCount(root->right);
    }
    if (root->left != nullptr) {
        getNodeCount(root->left);
    }
    if (root->right != nullptr) {
        getNodeCount(root->right);
    }
    return count;
};

node* newNode(int data) {
    node *node2 = new node(data);
    return (node2);
};

int main() {
    node *root = newNode(16);
    root->left = newNode(20);
    root->right = newNode(3);
    root->left->left = newNode(1);
    root->right->left = newNode(18);
    root->right->right = newNode(92);
    root->right->left->left = newNode(17);
    root->right->left->right = newNode(19);
    root->right->left->left->left = newNode(17);
    root->right->left->left->right = newNode(18);
    root->right->left->left->right->right = newNode(18);
    int count = getNodeCount(root);
    cout << "The number of nodes with two children in the tree is " << count << endl;
    return 0;
}
```

Figure 4: My code for problem 5

The Big-O of this function is $O(n)$.