CSCE 221 Cover Page

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 https://aggiehonor.tamu.edu/

Name	chase albright
UIN	529008060
Email address	chasealbright@tamu.edu

Cite your sources using the table below. Interactions with TAs and resources presented in lecture do not have to be cited.

People	1. None
Webpages	1. None
Printed Materials	1. None
Other Sources	1. None

Homework 2

Due March 25 at 11:59 PM

Typeset your solutions to the homework problems preferably in LaTeXor LyX. See the class webpage for information about their installation and tutorials.

1. (15 points) Provided two sorted lists, `l1` and `l2`, write a function in C++ to *efficiently* compute `l1` ∩ `l2` using only the basic STL list operations. The lists may be empty or contain a different number of elements e.g |`l1`| ≠ |`l2`|. You may assume `l1` and `l2` will not contain duplicate elements.

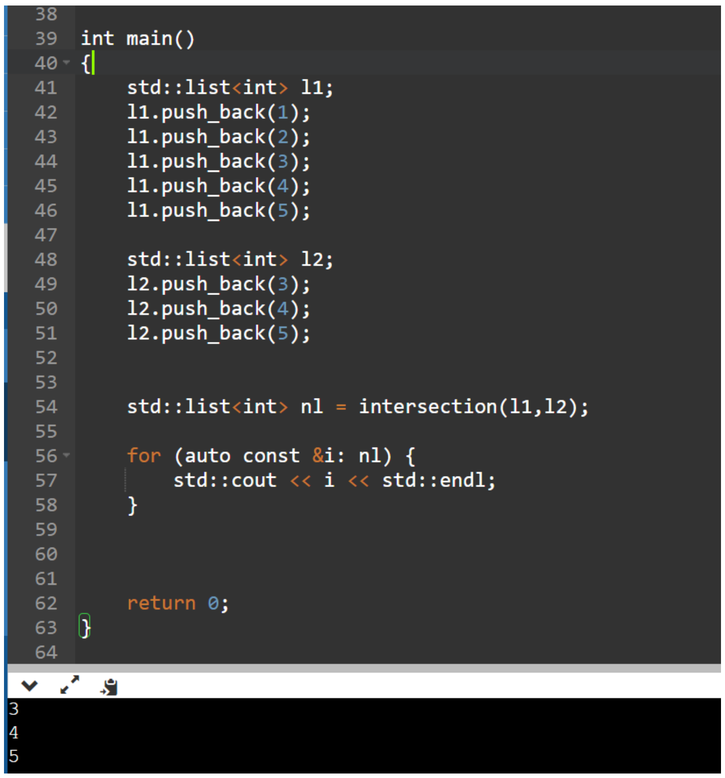
Examples (all set members are list node):

- $\{1, 2, 3, 4\} \cap \{2, 3\} = \{2, 3\}$
- $\emptyset \cap \{2, 3\} = \emptyset$
- $\{2, 9, 14\} \cap \{1, 7, 15\} = \emptyset$

(a) Complete the function below. Do not use any routines from the algorithm header file.

```
#include <list>

std::list<int> intersection(const std::list<int> & l1,
    const std::list<int> & l2) {
    std::list<int> newList;

    //iterators to first and last elements of both lists
    auto first1 = l1.begin();
    auto last1 = l1.end();

    auto first2 = l2.begin();
    auto last2 = l2.end();

    while (first1!=last1 && first2!=last2){
        if (*first1<*first2){
            ++first1;
        } else if (*first2<*first1){
            ++first2;
        } else {
            newList.push_back(*first1);
            ++first1; ++first2;
        }
    }

    return newList;
}
}
```

(b) Verify that your implementation works properly by writing two test cases. Provide screenshot(s) with the results of your testing.



```
int main()
{
    std::list<int> l1;
    l1.push_back(1);
    l1.push_back(2);
    l1.push_back(3);
    l1.push_back(4);
    l1.push_back(5);

    std::list<int> l2;
    l2.push_back(3);
    l2.push_back(4);
    l2.push_back(5);


    std::list<int> nl = intersection(l1,l2);

    for (auto const &i: nl) {
        std::cout << i << std::endl;
    }



    return 0;
}
```

```
3
4
5
```

(c) What is the running time of your algorithm? Provide a big-O bound. Justify.

$$f(n) \in O(n) \tag{1}$$

Since we are using a single while loop and both lists are sorted, we traverse each list and compare elements to see where the first common element occurs, this is an efficient algorithm to find the common elements of a list that takes O(n) time.

2. (15 points) Write a C++ recursive function that counts the number of nodes in a singly linked list. Do not modify the list.

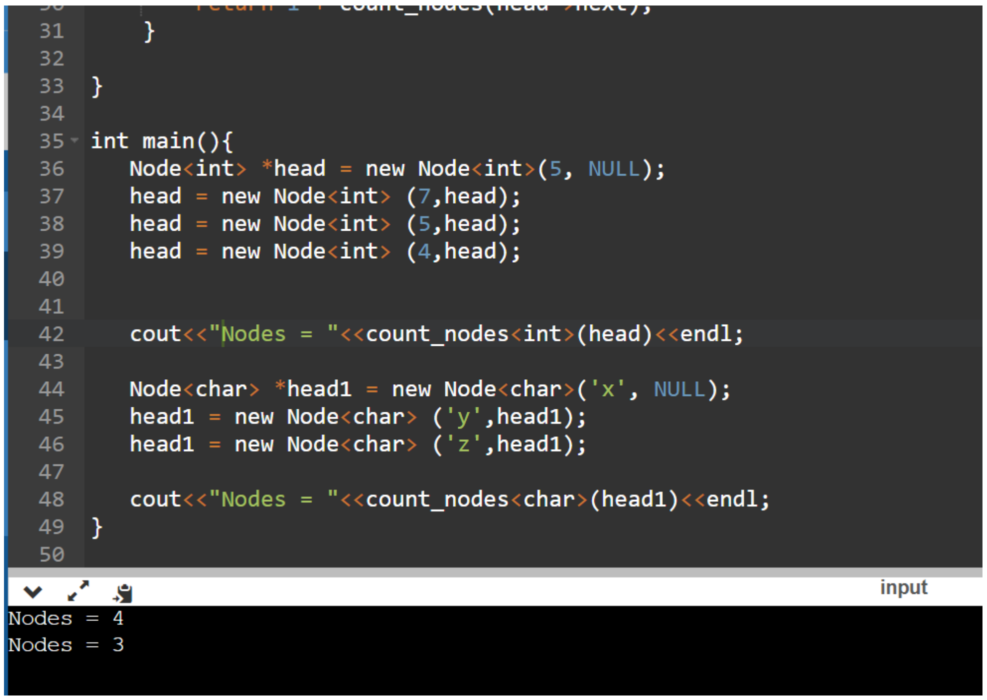
Examples:

- `count_nodes(`$(2) \to (4) \to (3) \to$ `nullptr)` $= 3$
- `count_nodes(nullptr)` $= 0$

(a) Complete the function below:

```
template<typename T>
struct Node {
  Node * next;
  T obj;

  Node(T obj, Node * next = nullptr)
    : obj(obj), next(next)
  { }
};

template<typename T>
int count_nodes(Node<T> * head) {
    if (head == nullptr) {
        return 0;
    } else {
        return 1 + count_nodes(head->next);
    }

}
```

(b) Verify that your implementation works properly by writing two test cases for the function you completed in part (a). Provide screenshot(s) with the results of your testing.



```
        return 1 + count_nodes(head->next);
    }

}

int main(){
   Node<int> *head = new Node<int>(5, NULL);
   head = new Node<int> (7,head);
   head = new Node<int> (5,head);
   head = new Node<int> (4,head);


   cout<<"Nodes = "<<count_nodes<int>(head)<<endl;

   Node<char> *head1 = new Node<char>('x', NULL);
   head1 = new Node<char> ('y',head1);
   head1 = new Node<char> ('z',head1);

   cout<<"Nodes = "<<count_nodes<char>(head1)<<endl;
}
```

```
Nodes = 4
Nodes = 3
```

(c) Write a recurrence relation that represents your algorithm.

$$T(n) = \begin{cases} T(n-1)+1, & \text{if } n > 1 \\ 1, & \text{if } n = 1 \end{cases} \tag{2}$$

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

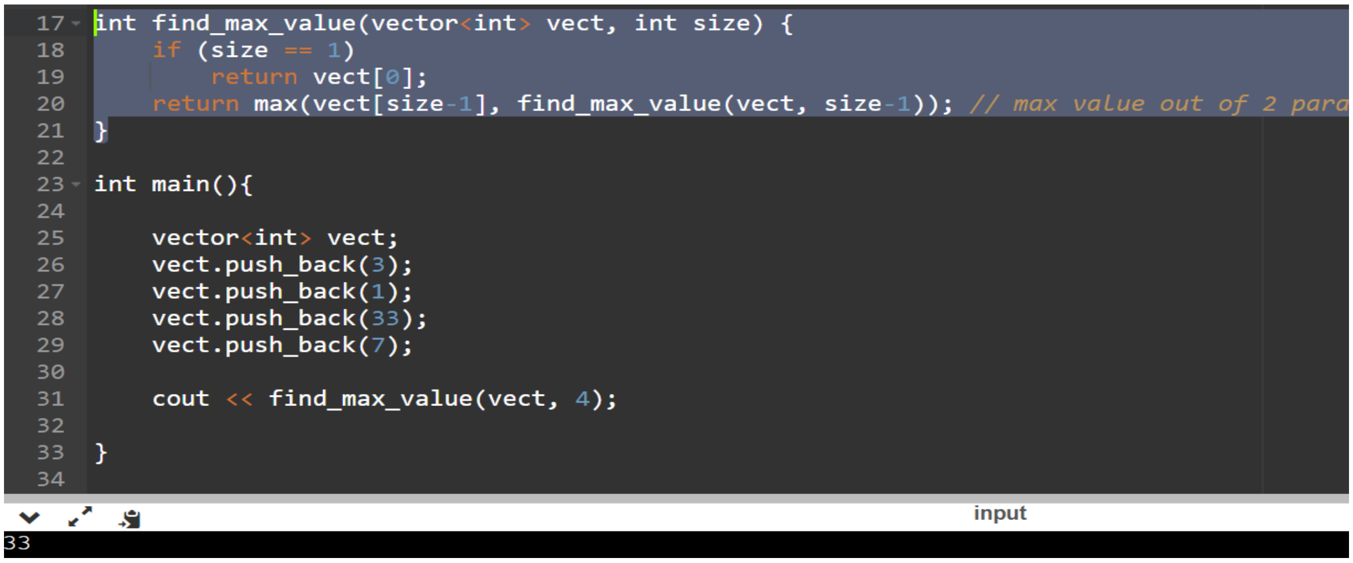
$$\begin{aligned} T(n) &= T(n-1)+1 && (3)\\ &= T(n-2)+1+1 && (4)\\ &= T(n-3)+1+1+1 && (5)\\ &= 1+1+1...(\text{n amount of times}) && (6)\\ \text{big-o} &= O(n) && (7) \end{aligned}$$

3. (15 points) Write a C++ recursive function that finds the maximum value in an array (or vector) of integers *without* using any loops. You may assume the array will always contain at least one integer. Do not modify the array.

(a) Complete the function below:

```
#include <vector>

int find_max_value(vector<int> vect, int size) {
        if (size == 1)
        return vect[0];
    return max(vect[size -1], find_max_value(vect, size -1)
        ); // max value out of 2 params
}
```

(b) Verify that your implementation works properly by writing two test cases. Provide screenshot(s) with the results of the tests.



```
int find_max_value(vector<int> vect, int size) {
    if (size == 1)
        return vect[0];
    return max(vect[size-1], find_max_value(vect, size-1)); // max value out of 2 para
}

int main(){

    vector<int> vect;
    vect.push_back(3);
    vect.push_back(1);
    vect.push_back(33);
    vect.push_back(7);

    cout << find_max_value(vect, 4);

}
```

```
33
```

(c) Write a recurrence relation that represents your algorithm.

$$T(n) = \begin{cases} T(n-1)+1, & \text{if } n > 1 \\ 0, & \text{if } n = 1 \end{cases} \tag{8}$$

(d) Solve the recurrence relation and obtain the running time of the algorithm in Big-O notation. Show your process.

$$\begin{aligned} T(n) &= T(n-1)+1 && (9)\\ &= T(n-2)+1+1 && (10)\\ &= T(n-3)+1+1+1 && (11)\\ &= 1+1+1\text{...(n amount of times)} && (12)\\ \text{big-o} &= O(n) && (13) \end{aligned}$$

4. (15 points) What is the best, worst and average running time of quick sort algorithm?

(a) Provide recurrence relations. For the average case, you may assume that quick sort partitions the input into two halves proportional to c and $1-c$ on each iteration.

Best:

$$T(n) = \begin{cases} T(n/2)+O(n), & \text{if } n > 1 \\ 0, & \text{if } n = 1 \end{cases} \tag{14}$$

Average:

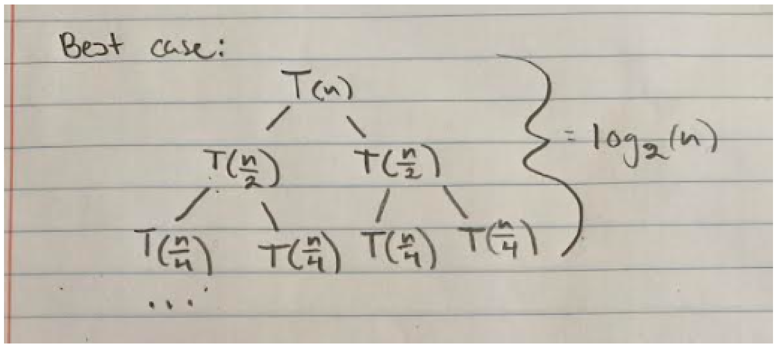
$$T(n) = \begin{cases} T(n-c)+T(n(1-C))+O(n), & \text{if } n > 1 \\ 0, & \text{if } n = 1 \end{cases} \tag{15}$$

Worst:

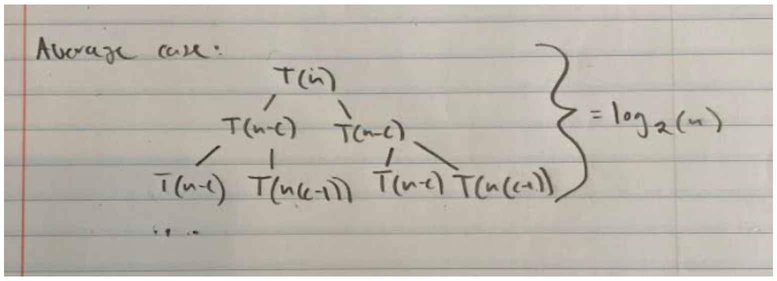
$$T(n) = \begin{cases} T(n-1)+O(n), & \text{if } n > 1 \\ 0, & \text{if } n = 1 \end{cases} \tag{16}$$

(b) Solve each recurrence relation you provided in part

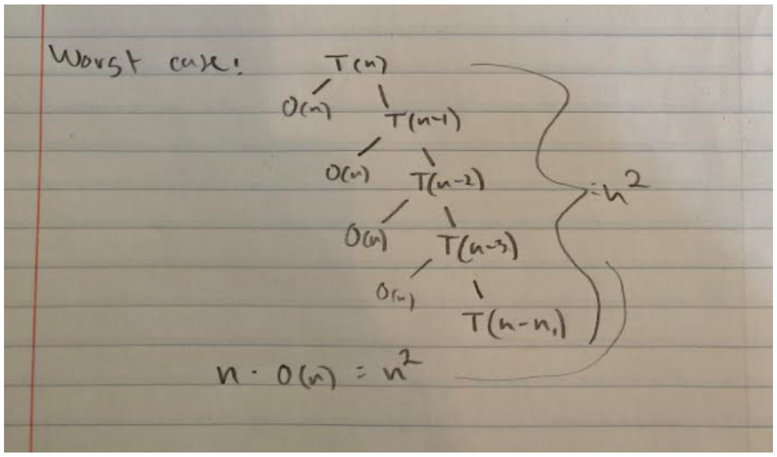
For the best case, each part of the tree halves after each iteration, this leads to O(nlog(n))



Similar to the best case, the halving occurs as well, it is a little slower but still over all nlog(n)



For the worst case, the list is completely unsorted and the algorithm has to loop through twice for each element leading to n^2*time*



(c) Provide an arrangement of the input array which results in each case. Assume the first item is always chosen as the pivot for each iteration.

Best	$\{4, 1, 2, 9, 5\}$
Average	$\{2, 4, 1, 9, 5\}$
Worst	$\{1, 2, 3, 4, 5\}$

best - The best case occurs whenever the middle element is first and chosen as the pivot point

average - The average case happens between when a value that is the pivot point is close to the middle value

worst - the worst case occurs whenever the list is completely sorted or reverse sorted and the lowest or highest value is chosen as the pivot

5. (15 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.

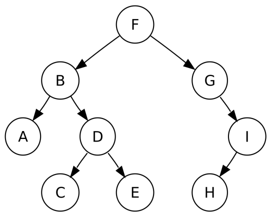


Figure 1: Calling `count_filled_nodes` on the root node F returns 3

(a) Complete the function below. The function will be called with the root node (e.g. `count_filled_nodes(root)`). The tree may be empty. Do not modify the tree.

```
#include <vector>

template<typename T>
struct Node {
  Node<T> *left, *right;
  T obj;

  Node(T obj, Node<T> * left = nullptr, Node<T> * right =
       nullptr)
    : obj(obj), left(left), right(right)
  { }
};

template<typename T>
int count_filled_nodes(const Node<T> * node) {
        if (node->right != nullptr && node->left !=
            nullptr){
            return ( 1 + count_filled_nodes(node->left) +
                 count_filled_nodes(node->right));
        } else if (node->right != nullptr && node->left
            == nullptr){
            return count_filled_nodes(node->right);
        } else if (node->left != nullptr && node->right
            == nullptr){
            return count_filled_nodes(node->left);
        } else {
            return 0;
        }
}
```

(b) Use big-O notation to classify your algorithm. Show how you arrived at your answer.

$$f(n) \in O(n) \tag{17}$$

This algorithm recursively calls itself at each node. Since checking is O(1) and there are n nodes in the tree, the going through the algorithm leads to O(n).

6. (15 points) For the following statements about red-black trees, provide a justification for each true statement and a counterexample for each false one.

(a) A subtree of a red-black tree is itself a red-black tree.
false - a subtree with a red root is not a red-black tree, if it were true this would contradict the definition of a red black tree.

(b) The sibling of an external node is either external or red.
True - If an external node had an internal black sibling, then these two node siblings would not have the same depth, this would contradict the fact that in red-black trees, all leaves must have the same black depth.

(c) There is a unique 2-4 tree associated with a given red-black tree.
true - for Every node that is in a red black tree, if it has two red children it can be uniquely represented in a 4 node tree. Each node with one red child will be a 3 node, and no red children will be a 2 node.

(d) There is a unique red-black tree associated with a given 2-4 tree.
false - this is false because a red black tree has two different representations of a 3 node tree (2-4 tree).

7. (10 points) Modify this skip list after performing the following series of operations: `erase(38)`, `insert(48,x)`, `insert(24, y)`, `erase(42)`. Provided the recorded coin flips for `x` and `y`. Provide a record of your work for partial credit.

$-\infty$	—	—	—	—	—	$+\infty$
$-\infty$	—	17	—	—	—	$+\infty$
$-\infty$	—	17	—	—	42	$+\infty$
$-\infty$	—	17	—	—	42	$+\infty$
$-\infty$	12	17	—	38	42	$+\infty$
$-\infty$	12	17	20	38	42	$+\infty$

erase 38 column

$-\infty$	—	—	—	—	$+\infty$
$-\infty$	—	17	—	—	$+\infty$
$-\infty$	—	17	—	42	$+\infty$
$-\infty$	—	17	—	42	$+\infty$
$-\infty$	12	17	—	42	$+\infty$
$-\infty$	12	17	20	42	$+\infty$

random height from coin flip =2

$-\infty$	—	—	—	—	—	$+\infty$
$-\infty$	—	17	—	—	—	$+\infty$
$-\infty$	—	17	—	42	—	$+\infty$
$-\infty$	—	17	—	42	—	$+\infty$
$-\infty$	12	17	—	42	48	$+\infty$
$-\infty$	12	17	20	42	48	$+\infty$

insert 24
random height coin flip = 1

$-\infty$	—	—	—	—	—	—	$+\infty$
$-\infty$	—	17	—	—	—	—	$+\infty$
$-\infty$	—	17	—	—	42	—	$+\infty$
$-\infty$	—	17	—	—	42	—	$+\infty$
$-\infty$	12	17	—	—	42	48	$+\infty$
$-\infty$	12	17	20	24	42	48	$+\infty$

erase 42

$-\infty$	—	—	—	—	—	$+\infty$
$-\infty$	—	17	—	—	—	$+\infty$
$-\infty$	—	17	—	—	—	$+\infty$
$-\infty$	—	17	—	—	—	$+\infty$
$-\infty$	12	17	—	—	48	$+\infty$
$-\infty$	12	17	20	24	48	$+\infty$