

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	Andrew Shang
UIN	830002330
Email address	ashang1524@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 \LaTeX or 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 $l1 \cap l2$ using only the basic STL list operations. The lists may be empty or contain a different number of elements e.g $|l1| \neq |l2|$. 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.

```
1 #include <list>
2
3 std::list<int> intersection(const std::list<int> & l1 ,
4     const std::list<int> & l2) {
5
6     list<int> both;
7     auto astart = l1.begin();
8     auto bstart = l2.begin();
9
10    while (astart != l1.end() && bstart != l2.end()) {
11        if (*astart < *bstart) {
12            astart++;
13        } else if (*bstart < *astart) {
14            bstart++;
15        } else {
16            if (both.back() != *astart || *astart == 0) {
17                both.push_back(*astart);
18            }
19            astart++;
20            bstart++;
21    }
```

```

22
23     return both;
24 }

```

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



```

l1: 0 1 2 3 4 5 6 7 8 9 9
l2: 0 5 6 7 8 9 10 11 12 13 14
intersection: 0 5 6 7 8 9

```

Figure 1: Results of testing the `intersection` function

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

$$f(n) \in O(\min(l1, l2)) \quad (1)$$

For any given intersection, such numbers must be included in both initial sets. Thus, if any given set were greater in length than the other, additional numbers would not be included. Thus is it only necessary to loop through the smaller set checking for any intersections.

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) → (4) → (3) → nullptr) = 3`
- `count_nodes(nullptr) = 0`

- (a) Complete the function below:

```

1  template<typename T>
2  struct Node {
3      Node * next;
4      T obj;
5
6      Node(T obj, Node * next = nullptr)
7          : obj(obj), next(next)
8          { }
9  };
10
11 template<typename T>

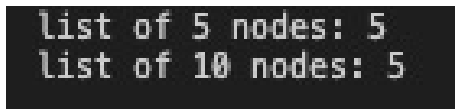
```

```

12 int count_nodes(T* node) {
13     if (node != nullptr) {
14         return 1 + count_nodes(node->next);
15     }
16 }

```

- (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.



```

list of 5 nodes: 5
list of 10 nodes: 5

```

Figure 2: Results of testing the `count_nodes` function

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

$$T(n) = \begin{cases} 0, & \text{if } n = 1 \\ T(n-1) + C, & \text{if } n = n \end{cases} \quad (2)$$

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

$$T(n) = T(n-1) + C \quad (3)$$

$$= T(n-2) + 2C \quad (4)$$

$$= T(n-3) + 3C \quad (5)$$

$$\dots \quad (6)$$

$$T(n-n) + nC \quad (7)$$

$$O(n) \quad (8)$$

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:

```

1 #include <vector>
2

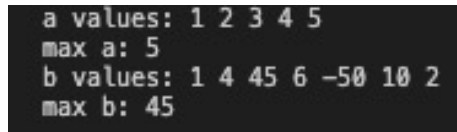
```

```

3 int find_max_value(int array[], int n) {
4     if (n == 1) { return array[0]; }
5     return max(array[n - 1], find_max_value(array, n - 1))
6     ;
7 }

```

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



```

a values: 1 2 3 4 5
max a: 5
b values: 1 4 45 6 -50 10 2
max b: 45

```

Figure 3: Results of testing the `find_max_value` function

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

$$T(n) = \begin{cases} = 0, & \text{if } n = 1 \\ = T(n - 1) + C, & \text{if } n = n \end{cases} \quad (9)$$

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

$$T(n) = T(n - 1) + C \quad (10)$$

$$= T(n - 2) + 2C \quad (11)$$

$$= T(n - 3) + 3C \quad (12)$$

$$\dots \quad (13)$$

$$T(n - n) + nC \quad (14)$$

$$O(n) \quad (15)$$

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:

$$T(n) = \begin{cases} = 0, & \text{if } n = 0 \\ = 2T(n/2), & \text{if } n = n \end{cases} \quad (16)$$

Average:

$$T(n) = \begin{cases} = 0, & \text{if } n = 0 \\ = T(n-1), & \text{if } n = n \end{cases} \quad (17)$$

Worst:

$$T(n) = \begin{cases} = 0, & \text{if } n = 0 \\ = T(n-1), & \text{if } n = n \end{cases} \quad (18)$$

- (b) Solve each recurrence relation you provided in part (a)
- (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	Picking the middle element as the pivot
Average	Having an arbitrary element as the pivot
Worst	Largest and smallest element 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.

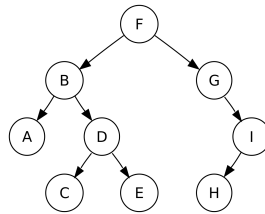


Figure 4: Calling `count_filled_nodes` on the root node F returns 3

- (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.

```

1 #include <vector>
2
3 template<typename T>
4 struct Node {
5     Node<T> *left , *right ;
6     T obj ;
7
8     Node(T obj , Node<T> * left = nullptr , Node<T> * right =
        nullptr)

```

```

9         : obj(obj), left(left), right(right)
10     { }
11 };
12
13 template<typename T>
14 int count_filled_nodes(const Node<T> * node) {
15     if (node && node->left && node->right) {
16         return 1 + count_filled_nodes(node->left) +
17             count_filled_nodes(node->right);
18     }
19     return 0;
20 }

```

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

The big-O notation is $O(n)$ since it is simply checking all possible nodes in any given tree for nodes with 2 children. There is only 1 operation being done at each call of the recursive function, which means for n nodes in a given tree, there are only n operations completed.

$$f(n) \in O(n) \quad (19)$$

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.

This is false because at any given red node, there cannot be a consecutive red node. Red nodes cannot have children, so at the worst case the number of nodes alternate in red and black order.

- (b) The sibling of an external node is either external or red.

This is true. An external node is such node that's null. This external node has to be black to begin with, as all external nodes are black. However, when calculating depth, all external nodes must have the same depth. If there were a black node that follows the original external node, this would break the depth rule. Thus, it can be possible for the sibling of an external node to be red as well.

- (c) There is a unique 2-4 tree associated with a given red-black tree.

This is true. There can be 2, 3, or 4 nodes in a 2-4 tree. For 2 nodes, we can have 2 pointers with 1 data element, which in that case is simply a node with no children. Likewise for 3 and 4 nodes, a red black tree with 1 and 2 red children respectively, satisfies this.

(d) There is a unique red-black tree associated with a given 2-4 tree.

This is false. While this applies for even numbered nodes, for a 3 node, this can be implemented in 2 seperate ways: black, red, black and red, black, red.

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	$+\infty$

-	0	-	-	-	-	-	+	~
-	0	-	17	-	-	-	+	~
-	0	-	17	-	-	42	+	~
-	0	-	17	-	-	42	+	~
-	0	12	17	-	3	42	+	~
-	0	17	17	20	7	42	+	~

insert(48, x)
x = coinflip = 5

-	0	-	-	-	-	48	+	~
-	0	-	17	-	-	48	+	~
-	0	-	17	-	42	48	+	~
-	0	-	17	-	42	48	+	~
-	0	12	17	-	42	48	+	~
-	0	17	17	20	42	48	+	~

insert(24, y)
y = coinflip = 1

-	0	-	-	-	-	48	+	~	
-	0	-	17	-	-	48	+	~	
-	0	-	17	-	-	42	48	+	~
-	0	-	17	-	-	42	48	+	~
-	0	12	17	-	24	42	48	+	~
-	0	17	17	20	24	42	48	+	~

erase(42)

-	0	-	-	-	-	48	+	~
-	0	-	17	-	-	48	+	~
-	0	-	17	-	-	48	+	~
-	0	-	17	-	-	48	+	~
-	0	12	17	-	24	48	+	~
-	0	17	17	20	24	48	+	~