Compiling command for this project is -std=c++11, and used compiler is g++.

Implementation 1:

In the first implementation (1.cpp), I tried array based brute force approach. The time complexity for this method is O(n^2).

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

void initialize(string& expression, int& i, int& n, int arr[10000]) {
    while (i < expression.length()) {
        int sign = 1;
        if (expression[i] == '-') {
            sign = -1;
            i++;
        }
        int num = 0;
        while (i < expression.length() && expression[i] >= '0' && expression[i] <= '9') {
            num = num * 10 + (expression[i] - '0');
            i++;
        }
        arr[n++] = sign * num;
        i++;
    }
}

void printRes(int res[], int count) {
    if (count > 0) {
        cout << res[0];
        for (int i = 1; i < count; i++) {
            cout << " " << res[i];
        }
        cout << endl;
    }

}

void solution(string expression) {
    int n = 0;
    int i = 0;
    int arr[10000];
    int res[10000];
    int count = 0;
    initialize(expression, i, n, arr);

    // uses array implementation and brute force;
    bool visited[10000] = { false };
    for (int i = 0; i < n; i++) {
        bool flag = false;
        if (visited[i]) {
            continue;
        }
        for (int j = i + 1; j < n; j++) {
            if (arr[i] == arr[j]) {
                visited[j] = true;
                flag = true;
            }
        }
        if (!flag) {
            res[count++] = arr[i];
        }
    }
    printRes(res, count);

}

int main() {
    string expression;
    getline(cin, expression);
    while (expression != "") {
        solution(expression);
        getline(cin, expression);
    }
}
```

Explanation:
This C++ program is designed to process a string of numbers, possibly with negative signs, and output the numbers that only appear once in the string. The main logic is in the solution function.

The initialize function processes the input string, converting sequences of digits into integers and storing them in an array. It also handles negative numbers by checking for a '-' sign before the number. The time complexity of this function is O(n), where n is the length of the string.

The solution function uses a brute force approach to find numbers that appear only once in the array. It iterates over the array, and for each number, it checks all the numbers that come after it in the array. If it finds a duplicate, it marks the duplicate as visited and sets a flag. If the flag is not set (meaning no duplicate was found), it adds the number to the result array. This function has a time complexity of O(n^2), where n is the number of elements in the array, because of the nested loops.

The printRes function simply prints the numbers in the result array. Its time complexity is O(n), where n is the number of elements in the result array.

The main function reads lines from the standard input until it encounters an empty line, and for each line, it calls the solution function. The time complexity of the main function depends on the number of lines and their lengths.

In summary, the overall time complexity of the program is dominated by the O(n^2) time complexity of the solution function. The space complexity is O(n), where n is the length of the input string, because of the arrays used to store the numbers and the results.

Worst Case:

The worst-case time complexity for this brute force array implementation is encountered when all the numbers in the input are distinct. This is because the inner loop in the solution function has to traverse all the remaining elements of the array for each element, and no elements are skipped by the visited check. Here are some test cases that would result in the worst-case time complexity:

A sequence of distinct positive integers: "1 2 3 4 5 6 7 8 9 10"
A sequence of distinct negative integers: "-1 -2 -3 -4 -5 -6 -7 -8 -9 -10"
A sequence of distinct integers, both positive and negative: "1 -2 3 -4 5 -6 7 -8 9 -10"
A sequence of distinct integers in non-ascending order: "10 9 8 7 6 5 4 3 2 1"
A sequence of distinct integers in non-descending order: "-10 -9 -8 -7 -6 -5 -4 -3 -2 -1"

Submission #289043 - Accepted

未命名用户

Problem 897: Find All Unique Elements

Compiler: C++ 11, flag: -static -std=c++0x **Runtime:** 0 seconds **Memory:** 1,688 KBs

Copy to Clipboard Plain Text

```cpp
/******************************************
 * (This comment block is added by the Judge System)
 * Submission ID: 289043
 * Submitted at:  2024-04-23 16:31:31
 *
 * User ID:       2511
 * Username:      57854329
 * Problem ID:    897
 * Problem Name:  Find All Unique Elements
 */

#include <iostream>
#include <string>
using namespace std;

void initialize(string& expression, int& i, int& n, int arr[10000]) {
    while (i < expression.length()) {
        int sign = 1;
        if (expression[i] == '-') {
            sign = -1;
            i++;
        }
        int num = 0;
        while (i < expression.length() && expression[i] >= '0' && expression[i] <= '9') {
            num = num * 10 + (expression[i] - '0');
            i++;
        }
        arr[n++] = sign * num;
        i++;
    }
}

void printRes(int res[], int count) {
    if (count > 0) {
        cout << res[0];
        for (int i = 1; i < count; i++) {
            cout << " " << res[i];
```

Implementation 2:

In the second implementation (2.cpp), I tried binary search tree approach. The time complexity for this method is O(n^2).

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

class Node {
public:
    int val;
    Node* left;
    Node* right;
    int count;
    Node(int val) : val(val), left(NULL), right(NULL), count(1) {}

};

class BinarySearchTree {
public:
    Node* root;
    BinarySearchTree() : root(NULL) {}

    void insert(int val) {
        if (root == NULL) {
            root = new Node(val);
        }
        else {
            insert(root, val);
        }
    }

    void insert(Node* node, int val) {
        if (val < node->val) {
            if (node->left == NULL) {
                node->left = new Node(val);
            }
            else {
                insert(node->left, val);
            }
        }
        else if (val > node->val) {
```

```cpp
            if (node->right == NULL) {
                node->right = new Node(val);
            }
            else {
                insert(node->right, val);
            }
        }
        else {
            node->count++;
        }
    }

    Node* find(Node* node, int val) {
        if (node == NULL) {
            return NULL;
        }
        if (val < node->val) {
            return find(node->left, val);
        }
        else if (val > node->val) {
            return find(node->right, val);
        }
        else {
            return node;
        }
    }

    Node* find(int val) {
        return find(root, val);
    }
};



void initialize(string& expression, int& i, int& n, int arr[10000]) {
    while (i < expression.length()) {
        int sign = 1;
        if (expression[i] == '-') {
            sign = -1;
            i++;
        }
        int num = 0;
        while (i < expression.length() && expression[i] >= '0' && expression[i] <= '9') {
            num = num * 10 + (expression[i] - '0');
            i++;
        }
        arr[n++] = sign * num;
        i++;
    }
}

void printRes(int res[], int count) {
    if (count > 0) {
        cout << res[0];
        for (int i = 1; i < count; i++) {
            cout << " " << res[i];
        }
        cout << endl;
    }

}

void solution(string expression) {
    int n = 0;
    int i = 0;
    int arr[10000];
    int res[10000];
    int count = 0;
    initialize(expression, i, n, arr);

    // uses binary search tree to store the numbers
    BinarySearchTree bst;
    for (int i = 0; i < n; i++) {
        bst.insert(arr[i]);
    }
    for (int i = 0; i < n; i++) {
        Node* node = bst.find(arr[i]);
        if (node->count == 1) {
            res[count++] = arr[i];
        }
    }


    printRes(res, count);

}

int main() {
    string expression;
    getline(cin, expression);
    while (expression != "") {
        solution(expression);
        getline(cin, expression);
    }
}
```

Explanation:

This C++ program is similar to the previous one, but it uses a binary search tree (BST) to store the numbers instead of an array. The main logic is still in the solution function.

The Node class represents a node in the BST. Each node has a value, a count of how many times the value appears, and pointers to the left and right child nodes.

The BinarySearchTree class represents the BST. It has methods to insert a value into the tree and to find a value in the tree. The insert method checks if the value is less than, greater than, or equal to the value of the current node and either creates a new node in the appropriate place or increments the count of the current node. The find method searches the tree for a node with a specific value. Both methods have a time complexity of O(log n) in the average case and O(n) in the worst case, where n is the number of nodes in the tree.

The initialize function is the same as in the previous program, with a time complexity of O(n), where n is the length of the string.

The solution function initializes the BST with the numbers from the array, then iterates over the array again and checks the count of each number in the BST. If the count is 1, it adds the number to the result array. The time complexity of this function is O(n log n) in the average case and O(n^2) in the worst case, where n is the number of elements in the array.

The printRes function is also the same as in the previous program, with a time complexity of O(n), where n is the number of elements in the result array.

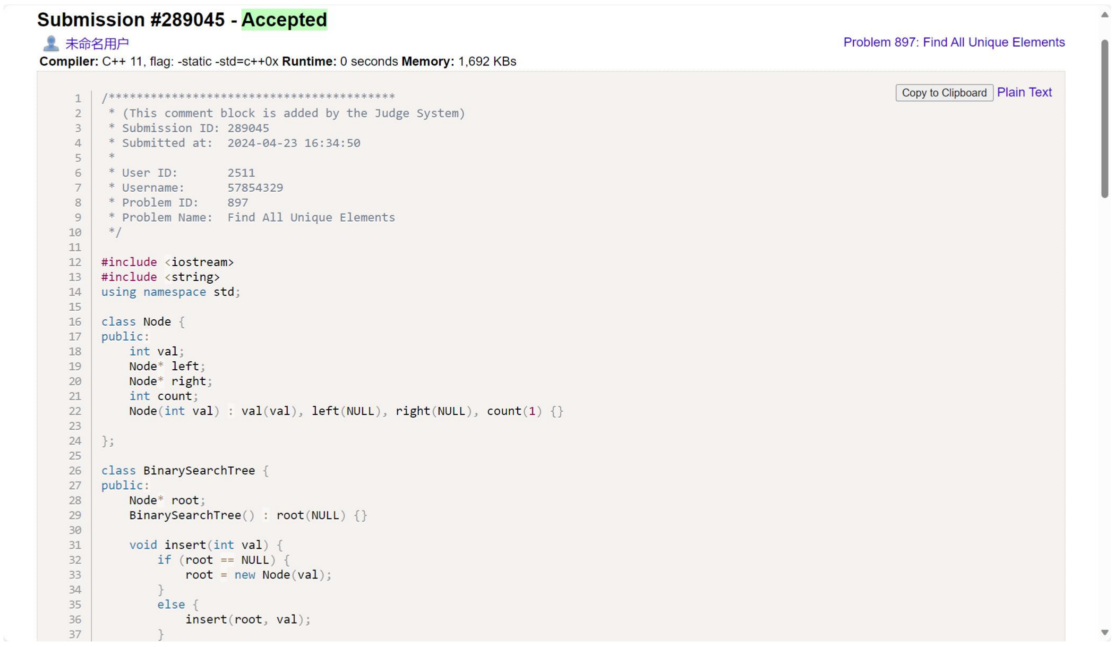
The main function is the same as in the previous program, with a time complexity that depends on the number of lines and their lengths.

In summary, the overall time complexity of the program is dominated by the O(n log n) time complexity of the solution function in the average case and the O(n^2) time complexity in the worst case. The space complexity is O(n), where n is the length of the input string, because of the arrays and the BST used to store the numbers and the results.

Worst Case:
The worst-case time complexity for this binary search tree implementation is encountered when the input numbers are in sorted order, either ascending or descending. This is because the tree

becomes unbalanced and degenerates into a linked list, causing the insert and find operations to have a time complexity of O(n) instead of O(log n). Here are some test cases that would result in the worst-case time complexity:

A sequence of ascending integers: "1 2 3 4 5 6 7 8 9 10"
A sequence of descending integers: "10 9 8 7 6 5 4 3 2 1"
A sequence of ascending negative integers: "-10 -9 -8 -7 -6 -5 -4 -3 -2 -1"
A sequence of descending negative integers: "-1 -2 -3 -4 -5 -6 -7 -8 -9 -10"
A sequence of alternating ascending and descending integers: "1 10 2 9 3 8 4 7 5 6"



Submission #289045 - Accepted

未命名用户 Problem 897: Find All Unique Elements

Compiler: C++ 11, flag: -static -std=c++0x **Runtime:** 0 seconds **Memory:** 1,692 KBs

```
/******************************************
 * (This comment block is added by the Judge System)
 * Submission ID: 289045
 * Submitted at:  2024-04-23 16:34:50
 *
 * User ID:       2511
 * Username:      57854329
 * Problem ID:    897
 * Problem Name:  Find All Unique Elements
 */

#include <iostream>
#include <string>
using namespace std;

class Node {
public:
    int val;
    Node* left;
    Node* right;
    int count;
    Node(int val) : val(val), left(NULL), right(NULL), count(1) {}

};

class BinarySearchTree {
public:
    Node* root;
    BinarySearchTree() : root(NULL) {}

    void insert(int val) {
        if (root == NULL) {
            root = new Node(val);
        }
        else {
            insert(root, val);
        }
```

Implementation 3:

In the third implementation (3.cpp), I tried linked list approach.
The time complexity for this method is O(n^2).

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

class Node {
public:
    int val;
    int cnt;
    Node* next;
    Node(int val, int cnt) {
        this->val = val;
        this->cnt = cnt;
        this->next = NULL;
    }

    Node() {
        this->val = 0;
        this->cnt = 0;
        this->next = NULL;
    }
};

class LinkedList {
public:
    Node* head;
    Node nodes[10000];
    int size;

    LinkedList() {
        this->head = NULL;
        this->size = 0;
    }

    Node* findNode(int val) {
        Node* cur = head;
        while (cur != NULL) {
            if (cur->val == val) {
                return cur;
            }
            cur = cur->next;
        }
        return NULL;
    }

    void addNode(int val) {
        Node* node = findNode(val);
        if (node == NULL) {
            if (this->head == NULL) {
                nodes[0] = Node(val, 1);
                size++;
                head = &nodes[0];
            }
            else {
                nodes[size] = Node(val, 1);
                size++;
                nodes[size - 2].next = &nodes[size - 1];
            }
        }
        else {
            node->cnt++;
            return;
        }
    }

    void res(int res[], int& count) {
        Node* cur = head;
        while (cur != NULL) {
            if (cur->cnt == 1) {
                res[count++] = cur->val;
            }
            cur = cur->next;
        }
    }
};

void initialize(string& expression, int& i, int& n, int arr[10000]) {
    while (i < expression.length()) {
        int sign = 1;
        if (expression[i] == '-') {
            sign = -1;
            i++;
        }
        int num = 0;
        while (i < expression.length() && expression[i] >= '0' && expression[i] <= '9') {
            num = num * 10 + (expression[i] - '0');
            i++;
        }
        arr[n++] = sign * num;
        i++;
    }
}

void printRes(int res[], int count) {
    if (count > 0) {
        cout << res[0];
        for (int i = 1; i < count; i++) {
            cout << " " << res[i];
        }
        cout << endl;
    }

}

void solution(string expression) {
    int n = 0;
    int i = 0;
    int arr[10000];
    int res[10000];
    int count = 0;
    initialize(expression, i, n, arr);

    //uses linked list to store the numbers
    LinkedList* list = new LinkedList();
    for (int i = 0; i < n; i++) {
        list->addNode(arr[i]);
    }
    list->res(res, count);

    printRes(res, count);

}

int main() {
    string expression;
    getline(cin, expression);
    while (expression != "") {
        solution(expression);
        getline(cin, expression);
    }
}
```

Explanation:
This C++ program is similar to the previous ones, but it uses a linked list to store the numbers instead of an array or a binary search tree. The main logic is still in the solution function.

The Node class represents a node in the linked list. Each node has a value, a count of how many times the value appears, and a pointer to the next node.

The LinkedList class represents the linked list. It has methods to find a node with a specific value (findNode) and to add a node to the list (addNode). If a node with the specified value already exists, it increments the count of that node; otherwise, it creates a new node at the end of the list. Both methods have a time complexity of O(n), where n is the number of nodes in the list, because in the worst case they need to traverse the entire list.

The res method of the LinkedList class iterates over the list and adds the values of the nodes with a count of 1 to the result array. Its time complexity is also O(n), where n is the number of nodes in the list.

The initialize function is the same as in the previous programs, with a time complexity of O(n), where n is the length of the string.

The solution function initializes the linked list with the numbers from the array, then calls the res method of the LinkedList class to get the numbers that appear only once. The time complexity of this function is O(n^2), where n is the number of elements in the array, because of the addNode method.

The printRes function is also the same as in the previous programs, with a time complexity of O(n), where n is the number of elements in the result array.

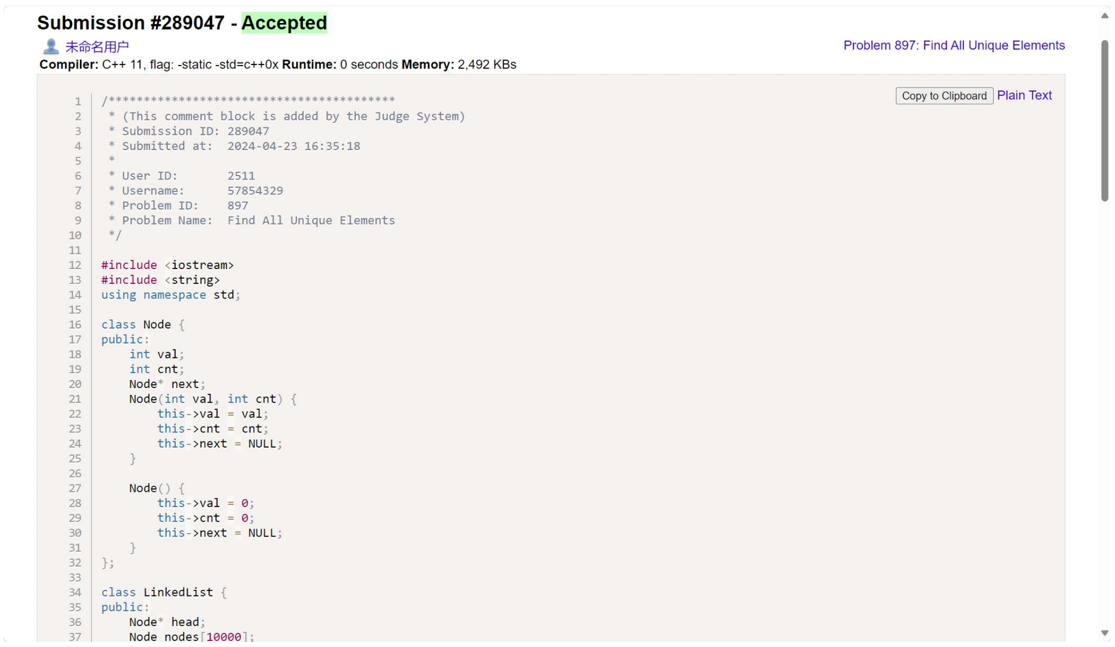
The main function is the same as in the previous programs, with a time complexity that depends on the number of lines and their lengths.

In summary, the overall time complexity of the program is dominated by the O(n^2) time complexity of the solution function. The space complexity is O(n), where n is the length of the input string, because of the arrays and the linked list used to store the numbers and the results.

Worst Case:
The worst-case time complexity for this linked list implementation is encountered when all the

numbers in the input are distinct. This is because the findNode method has to traverse the entire list for each number, and the list gets longer with each new number. Here are some test cases that would result in the worst-case time complexity:

A sequence of distinct positive integers: "1 2 3 4 5 6 7 8 9 10"
A sequence of distinct negative integers: "-1 -2 -3 -4 -5 -6 -7 -8 -9 -10"
A sequence of distinct integers, both positive and negative: "1 -2 3 -4 5 -6 7 -8 9 -10"
A sequence of distinct integers in non-ascending order: "10 9 8 7 6 5 4 3 2 1"
A sequence of distinct integers in non-descending order: "-10 -9 -8 -7 -6 -5 -4 -3 -2 -1"


Submission #289047 - Accepted
未命名用户
Problem 897: Find All Unique Elements
Compiler: C++ 11, flag: -static -std=c++0x **Runtime:** 0 seconds **Memory:** 2,492 KBs

```
/****************************************
 * (This comment block is added by the Judge System)
 * Submission ID: 289047
 * Submitted at:  2024-04-23 16:35:18
 *
 * User ID:       2511
 * Username:      57854329
 * Problem ID:    897
 * Problem Name:  Find All Unique Elements
 */

#include <iostream>
#include <string>
using namespace std;

class Node {
public:
    int val;
    int cnt;
    Node* next;
    Node(int val, int cnt) {
        this->val = val;
        this->cnt = cnt;
        this->next = NULL;
    }

    Node() {
        this->val = 0;
        this->cnt = 0;
        this->next = NULL;
    }
};

class LinkedList {
public:
    Node* head;
    Node nodes[10000];
```