Colin Weil
CS5350 Programming Assignment 1

Problem

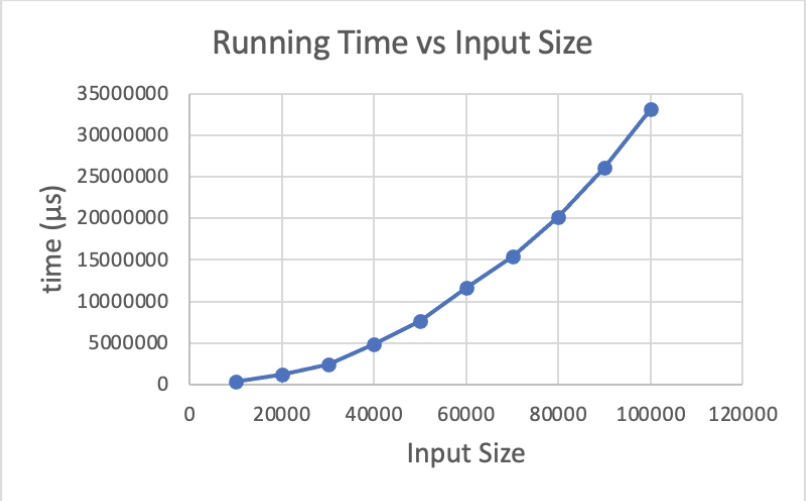
Write a program that takes a value "n" as input; produces "n" random numbers with a uniform distribution between 1 and n and places them in a linked list in sorted order. Place each one individually in the linked list where they belong to be in order, do not sort the list after placing them there.

```
void createSortedListOfSizeN(int n){
    LinkedList<int> list;
    for(int i = 0; i < n; i++){    /* Loop run 'n' times */
        int num = rand() % n + 1; /* rand is constant time  */
        list.insertSorted(num);    /* insertSorted is Θ(n) */
    }
}
```

The loop to insert the random number will run 'n' times and inserting into the sorted list will be also take 'n' time to traverse the list and find the correct position to insert into. Therefore, the time to run is $f(n) = n^2$ which is $\Theta(n^2)$

Here is the running tie chart and graph:

n	time (µs)
10000	295189
20000	1190486
30000	2401469
40000	4804386
50000	7606907
60000	11624549
70000	15384895
80000	20108989
90000	26067431
100000	33119165



The table and graph support my analysis of $\Theta(n^2)$ since as the size of n doubles (from 10,000 to 20,000) the running time also essential quadruples since $2^2 = x4$ (from 295,189 to 1,190,486).

Here is the code in C++: The highlighted code is the tested code. The rest of the code is scaffolding code that can be common for all problems or supplementary code to make the linked list class function correctly.

Main.cpp

```
#include <iostream>
#include <chrono>
#include <stdlib.h>
#include "LinkedList.h"
#include <fstream>
using namespace std;

void createSortedListOfNSize(int n){
    LinkedList<int> list;
    for(int i = 0; i < n; i++){
        int num = rand() % n + 1;
        list.insertSorted(num);
    }
}

int main() {
    ofstream output;
    output.open("../output.csv");
    output << "n" << "," << "Time (ns)" << endl;

    for(int i = 10000; i <= 100000; i+=10000)
    {
        auto beginTime = chrono::high_resolution_clock::now();
        createSortedListOfNSize(i);
        auto endTime = chrono::high_resolution_clock::now();
        auto runTime =
chrono::duration_cast<chrono::microseconds>(endTime -
beginTime);
        output << i << "," << runTime.count() << endl;
        cout << i << "," << runTime.count() << endl;
    }

    output.close();
    return 0;
}
```

LinkedList.h

/* MOST ALL CODE IN THIS CLASS CAME FROM MY DATA STRUCTURES PROJECT 4 – FLIGHT PLANNER
THE ONLY CHANGES WERE ADDING THE INSERT SORTED FUNCTION AND REMOVING UNNESSECARY FUNCTIONS FOR THIS ASSIGNMENT */

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

template<class T>
class ListNode{
public:
    T data;
    ListNode* next;
    ListNode* previous;

    ListNode(T d, ListNode<T>* n = nullptr, ListNode<T>* p =
nullptr){
        data = d;
        next = n;
        previous = p;
    }

    bool operator==(const ListNode& n) const { return data ==
n.data; }
    T& getData(){ return data; }
    ListNode* getNext(){ return next; }
    ListNode* getPrevious(){ return previous; }
};

template<class T>
class LinkedList {
private:
    ListNode<T> *head;
public:
    LinkedList();
    LinkedList(T data);
    LinkedList(const LinkedList &list);
    void insertSorted(T data);
    void print(ostream &out);
    ~LinkedList();
    LinkedList &operator=(const LinkedList &l);
    int getSize();
};
```

```
template<class T>
void LinkedList<T>::insertSorted(T data) {
    // No data
    if(head == nullptr) {
        head = new ListNode<T>(data);
    }
    // Data is smaller than head
    else if(data < head->data) {
        ListNode<T> *temp = new ListNode<T>(data, head,
nullptr);
        head->previous = temp;
        head = temp;
    }
    else{
        ListNode<T> *cur = head;
        while(cur->getNext() != nullptr && data > cur-
>getNext()->getData()){
            cur = cur->getNext();
        }
        // Data is new TAIL
        if(cur->getNext() == nullptr){
            ListNode<T> *temp = new ListNode<T>(data, nullptr,
cur);
            cur->next = temp;
        }
        // Middle of list
        else {
            ListNode<T> *temp = new ListNode<T>(data, cur->next,
cur);
            cur->next = temp;
            cur->next->previous = temp;
        }
    }
}
```

```
template<class T>
LinkedList<T>::LinkedList() {
    head = nullptr;
}

template<class T>
LinkedList<T>::LinkedList(T data) {
    head = new ListNode<T>(data);
}
```

```
template<class T>
LinkedList<T>::LinkedList(const LinkedList &list){
    head = nullptr;
    ListNode<T> *cur = list.head;
    while (cur != nullptr) {
        insertSorted(cur->data);
        cur = cur->next;
    }
}

template<class T>
LinkedList<T>& LinkedList<T>::operator=(const LinkedList<T>&
list){
    head = nullptr;
    ListNode<T> *cur = list.head;
    while (cur != nullptr) {
        insertSorted(cur->data);
        cur = cur->next;
    }
    return *this;
}

template<class T>
void LinkedList<T>::print(ostream& out) {
    if(head == nullptr){
        out << "empty" << endl;
    }
    ListNode<T>* cur = head;
    while(cur->next->next != nullptr){
        out << cur->data << " -> " ;
        cur = cur->next;
    }
    out << cur->next->data;
    out << endl;
}
template<class T>
LinkedList<T>::~LinkedList() {
    ListNode<T>* temp;

    if(head != nullptr) {
        while (head != nullptr) {
            temp = head;
            head = head->next;
            delete temp;
```

```
        }
    }
}

template<class T>
int LinkedList<T>::getSize(){
    if(head == nullptr){
        return 0;
    }
    int size = 0;
    ListNode<T>* cur = head;
    while(cur != nullptr){
        cur = cur->next;
        size++;
    }
    return size;
}
```