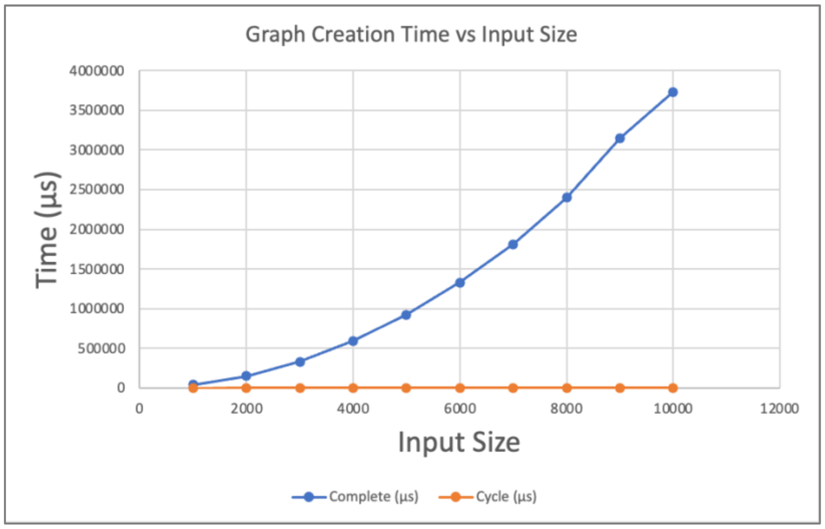
Colin Weil
CS5350 Programming Assignment 2

Problem
Create a program that accepts a number of vertices "V", create both an undirect complete graph and cycle graph with "V" vertices using an adjacency list data structure of your creation and then output the graph in the format below. Do not include the outputting of the graph in your timing analysis.

For creating the complete graph, every vertex connects to every other vertex. This is done in a double for-loop to V which is why creating a complete graph has the asymptotic notation of $\Theta(n^2)$.
For creating the cycle graph, every vertex only has to connect to two other vertexes. I did this simply by connecting every vertex to the number before it and after it in a single for loop. I then had to connect the last vertex to the first vertex also but that is constant time. Because this process only uses one for loop to V, the asymptotic notation of creating a cycle is $\Theta(n)$.

Here is the running time chart and graph:

n	Complete (µs)	Cycle (µs)
1000	39426	87
2000	150637	175
3000	332655	291
4000	596571	318
5000	921478	413
6000	1328686	475
7000	1812891	601
8000	2399988	642
9000	3150865	706
10000	3730386	811



The table and graph support my analysis of $\Theta(n^2)$ for the complete graph since as the size of n doubles (from 2000 to 4000) the running time also essential quadruples since $2^2 = x4$ (from 150,637 to 596,571).
The table and graph support my analysis of $\Theta(n)$ for the cycle graph since as the size of n doubles (from 1000 to 2000) the running time also essential doubles (from 87 to 175).

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.

The output for both of the graphs are below all of the code

Main.cpp

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

AdjacencyList createComplete(int V){
    AdjacencyList graph(V);
    for(int i = 0; i < V; i++){
        for(int j = 0; j < V; j++){
            if(i != j)
                graph.addSingleEdge(i, j);
        }
    }
    return graph;
}

AdjacencyList createCycle(int V){
    AdjacencyList graph(V);
    for(int i = 1; i < V; i++){
        graph.addEdge(i, i-1);
    }
    graph.addEdge(0, V-1);
    return graph;
}
```

```
int main() {
    ofstream output("../timings.csv");
    output << "n" << "," << "Complete" << "," << "Cycle" << endl;

    for(int i = 1000; i <= 10000; i+=1000)
    {
        auto beginTime = chrono::high_resolution_clock::now();
        createComplete(i);
        auto endTime = chrono::high_resolution_clock::now();
        auto completeRunTime =

chrono::duration_cast<chrono::microseconds>(endTime - beginTime);

        beginTime = chrono::high_resolution_clock::now();
        createCycle(i);
        endTime = chrono::high_resolution_clock::now();
        auto cycleRunTime =

chrono::duration_cast<chrono::microseconds>(endTime - beginTime);

        output << i << "," << completeRunTime.count() << "," <<
cycleRunTime.count() << endl;
        cout << i << "," << completeRunTime.count() << "," <<
cycleRunTime.count() << endl;
    }
    output.close();

    AdjacencyList com = createCycle(5);
    AdjacencyList cy = createComplete(5);

    com.OutputFile("../complete.txt");
    cy.OutputFile("../cycle.txt");

    return 0;
}
```

AdjacencyList.h
```
#ifndef PA2_ADJACENCYLIST_H
#define PA2_ADJACENCYLIST_H
#include <algorithm>
#include <iostream>
#include “LinkedList.h”
using namespace std;

class AdjacencyList{
private:
    int V;
    LinkedList<int>* adjLists;
public:
    AdjacencyList(int V);
    void addEdge(int src, int dest);
    void addSingleEdge(int src, int dest);
    ~AdjacencyList();
    void display();
    void OutputFile(string filename);
};

#endif //PA2_ADJACENCYLIST_H
```

AdjacencyList.cpp
```
#include "AdjacencyList.h"
#include <vector>
#include <fstream>
using namespace std;

AdjacencyList::AdjacencyList(int size){
    V = size;
    edges = 0;
    adjLists = new LinkedList<int>[V];
    for(int i = 0; i < V; i++) {
        LinkedList<int> temp;
        adjLists[i] = temp;
    }
}

void AdjacencyList::addEdge(int src, int dest){
    adjLists[src].push_back(dest);
    adjLists[dest].push_back(src);
    edges += 2;
}

void AdjacencyList::addSingleEdge(int src, int dest){
    adjLists[src].push_back(dest);
    edges++;
}

AdjacencyList::~AdjacencyList(){
    delete[] adjLists;
}

void AdjacencyList::display(){
    for(int i = 0; i < V; i++){
        cout << i+1;
        adjLists[i].printListPlusOne();
        cout << endl;
    }
}

void AdjacencyList::OutputFile(string filename){
    int outputSize = 1 + V + edges;
    int out[outputSize];
    out[0] = V;

    int index = 1 + V;
    for(int i = 0; i < V; i++) {
        out[i+1] = index;
        adjLists[i].ToOutputPlusOne(out, index);
    }

    ofstream outfile(filename);
    for(int i = 0; i < outputSize; i++)
        outfile << out[i] << endl;
    outfile.close();
}
```

LinkedList.h
```
#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;
    ListNode<T> *tail;
public:
    LinkedList();
    LinkedList(T data);
    LinkedList(const LinkedList &list);
    void ToOutputPlusOne(int* out, int &index);
    void printListPlusOne();
    ~LinkedList();
    LinkedList &operator=(const LinkedList &l);
    int getSize();
    void push_back(T data);
};

template<class T>
void LinkedList<T>::push_back(T data) {
    // No data
    if(head == nullptr) {
        head = new ListNode<T>(data);
        tail = head;
    }
    else{
        ListNode<T> *temp = new ListNode<T>(data, nullptr, tail);
        tail->next = temp;
        tail = temp;
    }
}

template<class T>
LinkedList<T>::LinkedList() {
    head = nullptr;
    tail = 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) {
        push_back(cur->data);
        cur = cur->next;
    }
    return *this;
}

template<class T>
void LinkedList<T>::ToOutputPlusOne(int* out, int &index) {
    ListNode<T>* cur = head;
    while(cur != nullptr){
        out[index] = cur->data+1;
        index++;
        cur = cur->next;
    }
}

template<class T>
void LinkedList<T>::printListPlusOne() {
    if(head == nullptr){
        return;
    }
    ListNode<T>* cur = head;
    while(cur->next != nullptr){
        cout << " -> " << cur->data;
        cur = cur->next;
    }
}

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;
}
```

Complete Output with V = 5

```
5
6
10
14
18
22
2
3
4
5
1
3
4
5
1
2
4
5
1
2
3
5
1
2
3
4
```

Cycle Output with V = 5

```
5
6
8
10
12
14
2
5
1
3
2
4
3
5
4
1
```