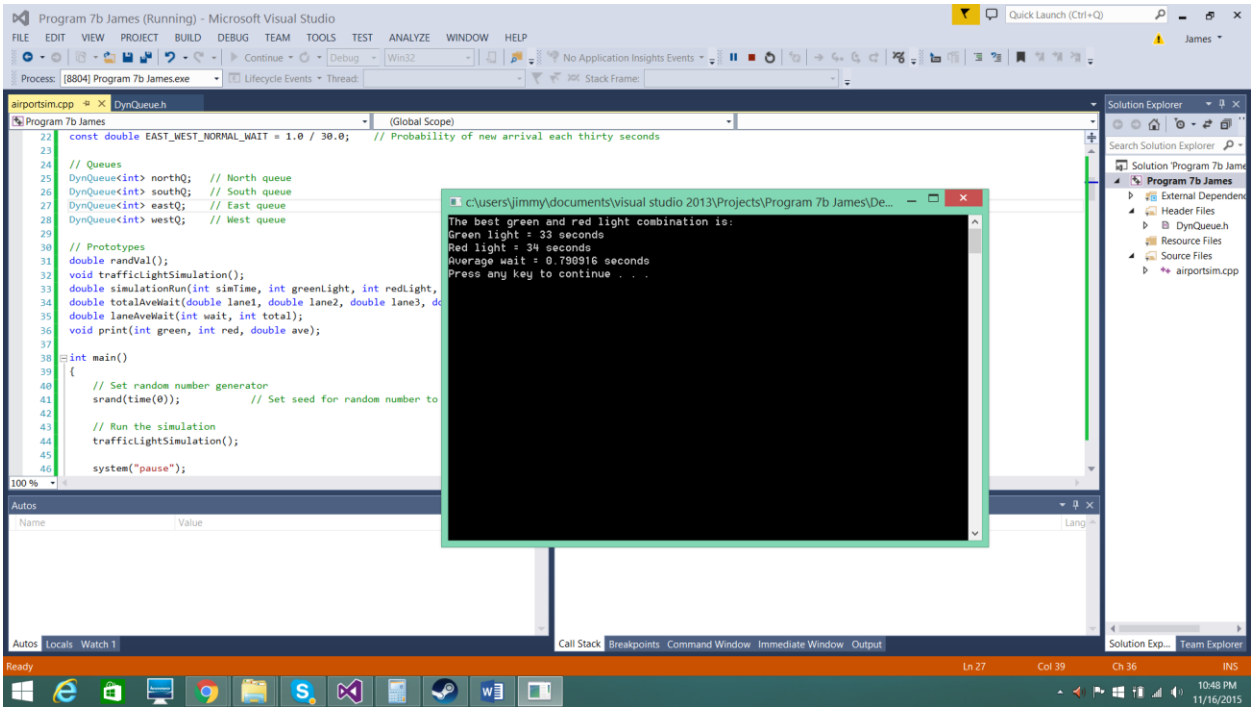
James Wetters Program 7 CST 280




```
/* The traffic light was simulated with about 570
possible combinations of green and red lights */


Run 1


The best green and red light combination is:

Green light = 33 seconds

Red light = 34 seconds

Average wait = 0.790916 seconds

Press any key to continue . . .
```

Run 2

```
The best green and red light combination is:
Green light = 17 seconds
Red light = 28 seconds
Average wait = 0.734116 seconds
Press any key to continue . . .
```

```
// This program calculates the average wait time for a four way traffic at an
// intersection with multiple green and red light times to find the fastest
// green and red light combination
//
// Author James Wetters
#include <iostream>
#include <cstdlib>    // for random number functions
#include <ctime>      // for clock functions
using namespace std;

// Queue library
#include "DynQueue.h"

// Constants
const int SIM_START = 16;          // Lowest number of seconds tested
const int SIM_END = 40;                   // Higest number of seconds tested
const int SIM_RUSH_TIME = 7200;                  // 2 hours in seconds
const int SIM_NORMAL_TIME = 21600;        // 6 hours in seconds
const double NORTH_SOUTH_RUSH_WAIT = 1.0 / 5.0;        // Probability of new arrival each
five seconds
const double EAST_WEST_RUSH_WAIT = 1.0 / 20.0;         // Probability of new arrival each
twenty seconds
const double NORTH_SOUTH_NORMAL_WAIT = 1.0 / 10.0;     // Probability of new arrival each
ten seconds
const double EAST_WEST_NORMAL_WAIT = 1.0 / 30.0;// Probability of new arrival each thirty
seconds

// Queues
DynQueue<int> northQ;   // North queue
DynQueue<int> southQ;   // South queue
DynQueue<int> eastQ; // East queue
DynQueue<int> westQ; // West queue

// Prototypes
double randVal();
void trafficLightSimulation();
```

```
double simulationRun(int simTime, int greenLight, int redLight, double northSouthWait,
double eastWestWait);
double totalAveWait(double lane1, double lane2, double lane3, double lane4);
double laneAveWait(int wait, int total);
void print(int green, int red, double ave);

int main()
{
	// Set random number generator
	srand(time(0));			// Set seed for random number to clock

	// Run the simulation
	trafficLightSimulation();

	system("pause");
	return 0;
}

//--------------------------------------------------------------------------
// Traffic Light Simulation
//
//	The simulation simulates the traffic light hundreds of times and
//	compares the results to find the fastest time
//--------------------------------------------------------------------------
void trafficLightSimulation()
{
	// Initilize Variables
	double firstRushRun, normalRun, secoundRushRun, currentRunAve;
	double bestAverage = 99999;
	int bestGreen, bestRed;

	// Simulation
	// Go and stop correspond to North and Souths light
	// Go == Green and Stop == Red
	for (int go = SIM_START; go < SIM_END; go++)				// Green
	{
		for (int stop = SIM_START; stop < SIM_END; stop++)		// Red
		{

			// 2 hours of high traffic
			firstRushRun = abs(simulationRun(SIM_RUSH_TIME, go, stop,
NORTH_SOUTH_RUSH_WAIT, EAST_WEST_RUSH_WAIT));
			// 6 hours of normal traffic
			normalRun = abs(simulationRun(SIM_NORMAL_TIME, go, stop,
NORTH_SOUTH_NORMAL_WAIT, EAST_WEST_NORMAL_WAIT));
			// 2 hours of high traffic
			secoundRushRun = abs(simulationRun(SIM_RUSH_TIME, go, stop,
NORTH_SOUTH_RUSH_WAIT, EAST_WEST_RUSH_WAIT));

			//cout << firstRushRun << endl;
			//cout << normalRun << endl;
			//cout << secoundRushRun << endl;

			// Get the overall average for the current run
			currentRunAve = (firstRushRun + normalRun + secoundRushRun) / 3;

			// Find lowest Average
			if (currentRunAve < bestAverage)
```

```
                        {
                                bestAverage = currentRunAve;
                                bestGreen = go;
                                bestRed = stop;
                        }
                }
        }

        // Write the results
        print(bestGreen, bestRed, bestAverage);
}


//---------------------------------------------------------------------------
// Simulation Run
//
//      Each run recives the total time in seconds the green and red lights in
//      seconds and the probability of traffic for north/south and east/west.
//
//      The average wait time for each lane is averaged and returned as a
//      double in seconds.
//---------------------------------------------------------------------------
double simulationRun(int simTime, int greenLight, int redLight, double northSouthWait,
double eastWestWait)
{
        // Initilize variables
        int time;                 // Time clock for simulation

        int northTraffic = 0;
        int southTraffic = 0;
        int eastTraffic = 0;
        int westTraffic = 0;

        // Traffic light
        // If green is true North and South have a green light
        bool green = true;

        // light lasts for 30 secounds
        int light = 30;

        // Current Traffic for each direction
        int n = 0, s = 0, e = 0, w = 0;

        // Total number of cars for each direction
        int nTotal = 0;
        int sTotal = 0;
        int eTotal = 0;
        int wTotal = 0;

        // Total wait time in secounds for each direction
        int nTotalWait = 0;
        int sTotalWait = 0;
        int eTotalWait = 0;
        int wTotalWait = 0;

        // Average wait time in secounds for each direction
        double northAveWait;
        double southAveWait;
```

```
	double eastAveWait;
	double westAveWait;

	int wait;		// Wait time of traffic exiting queues

	// Running the simulation of the intersection for 2 or 6 hours in seconds
	for (time = 1; time <= simTime; time++)
	{
		if (randVal() <= northSouthWait)		// New arrival
		{
			northQ.enqueue(time);					// enqueue
			northTraffic++;						//
Increment traffic
		}
		if (randVal() <= northSouthWait)		// New arrival
		{
			southQ.enqueue(time);					// enqueue
			southTraffic++;						//
Increment traffic
		}
		if (randVal() <= eastWestWait)		// New arrival
		{
			eastQ.enqueue(time);				// enqueue
			eastTraffic++;						//
Increment traffic
		}
		if (randVal() <= eastWestWait)		// New arrival
		{
			westQ.enqueue(time);				// enqueue
			westTraffic++;						//
Increment traffic
		}

		// If the light for North and south is green
		if (green == true)
		{
			// If two seconds have passed
			if (time % 2 == 0)
			{
				// If the queue isn't empty then dequeue
				if (!northQ.isEmpty())
				{
					n = northQ.dequeue();			// Get time
from queue
					wait = time - n;			// Calculate
wait time in queue
					nTotalWait += wait;			// Sum total
wait time
					nTotal++;					//
Increment total landed
				}
				// If the queue isn't empty then dequeue
				if (!southQ.isEmpty())
				{
					s = southQ.dequeue();			// Get time
from queue
					wait = time - s;			// Calculate
wait time in queue
```

```
                                    sTotalWait += wait;                         // Sum total
wait time
                                    sTotal++;
                             }
                      }
               }
               // If the light is red for North and South then East and West have a green
light
               else if (time % 2 == 0)
               {
                      // If the queue isn't empty then dequeue
                      if (!eastQ.isEmpty())
                      {
                             e = eastQ.dequeue();                       // Get time from
queue
                             wait = time - e;                                   // Calculate
wait time in queue
                             eTotalWait += wait;                                // Sum total
wait time
                             eTotal++;                                                 //
Increment total landed
                      }
                      // If the queue isn't empty then dequeue
                      if (!westQ.isEmpty())
                      {
                             w = westQ.dequeue();                       // Get time from
queue
                             wait = time - w;                                   // Calculate
wait time in queue
                             wTotalWait += wait;                                // Sum total
wait time
                             wTotal++;
                      }
               }

               // Decrement one second from the light
               light--;

               // If the light has zero seconds left
               if (light == 0)
               {
                      // If the light is green
                      if (green)
                      {
                             // Change the light
                             green = false;
                             // Asign the light time
                             light = greenLight;
                      }
                      else
                      {
                             // Change the light
                             green = true;
                             // Asign the light time
                             light = redLight;
                      }
               }
       }
```

```
	// Calculate the average wait time from each direction in secounds
	northAveWait = laneAveWait(nTotalWait, nTotal);
	southAveWait = laneAveWait(sTotalWait, sTotal);
	eastAveWait = laneAveWait(eTotalWait, eTotal);
	westAveWait = laneAveWait(wTotalWait, wTotal);

	// Calculate and return the total average wait in secounds
	return totalAveWait(northAveWait, southAveWait, eastAveWait, westAveWait);
}


//-------------------------------------------------------------------------
// Random Value
//
// This function returns as random number between 0.0 and 1.0
//-------------------------------------------------------------------------
double randVal()
{
	return double(rand()) / double(RAND_MAX);
}

//-------------------------------------------------------------------------
// Lane Average Wait
//
//	Recivies the wait time in seconds and total number of cars
//
//	Calculates average wait time for each car and returns the average
//-------------------------------------------------------------------------
double laneAveWait(int wait, int total)
{
	// Initilize
	double average = 0;

	// Calculate Average
	average = (double) wait / (double) total;

	// Return average
	return average;
}

//-------------------------------------------------------------------------
// Total Average Wait
//
//	Recivies the wait time in seconds for four lanes of traffic
//
//	Calculates average wait time for all four lanes of traffic and
//	returns the total average in seconds
//-------------------------------------------------------------------------
double totalAveWait(double lane1, double lane2, double lane3, double lane4)
{
	// Initilize
	double totalAverage = 0;

	// Calculate Average
	totalAverage = (lane1 + lane2 + lane3 + lane4) / 4;

	// Return average
```

```
	return totalAverage;
}


//------------------------------------------------------------------------
// Print
//
//     Recives the best green light time, red light time and the average wait
//     time in seconds
//
//     Prints the best green light time the red light time and the average wait
//     time to the screen
//------------------------------------------------------------------------
void print(int green, int red, double ave)
{
	cout << "The best green and red light combination is: " << endl;
	cout << "Green light = " << green << " seconds" << endl;
	cout << "Red light = " << red << " seconds" << endl;
	cout << "Average wait = " << ave << " seconds" << endl;
}
```

```
#ifndef DYNQUEUE_H
#define DYNQUEUE_H

template <class ItemType>
class DynQueue
{
private:
	struct NodeType
	{
		ItemType info;
		NodeType *next;
	};

	NodeType *front;
	NodeType *rear;
	int numItems;
public:
	DynQueue();
	~DynQueue();
	void enqueue(ItemType);
	ItemType dequeue();
	bool isEmpty();
	bool isFull();
	void clear();
};

#endif

//=================================================
//Implementation for Dynamic Queue class
//=================================================

#include <iostream>
```

```
using namespace std;

//************************
// Constructor           *
//************************

template <class ItemType>
DynQueue<ItemType>::DynQueue()
{
	front = NULL;
	rear = NULL;
	numItems = 0;
}

//************************
// Destructor            *
//************************

template <class ItemType>
DynQueue<ItemType>::~DynQueue()
{
	clear();
}

//*****************************************
// Function enqueue inserts the value in num *
// at the rear of the queue.                 *
//*****************************************

template <class ItemType>
void DynQueue<ItemType>::enqueue(ItemType item)
{
	NodeType *newNode;

	newNode = new NodeType;
	newNode->info = item;
	newNode->next = NULL;
	if (isEmpty())
	{
		front = newNode;
		rear = newNode;
	}
	else
	{
		rear->next = newNode;
		rear = newNode;
	}
	numItems++;
}

//*****************************************
// Function dequeue removes the value at the   *
// front of the queue, and copies it into num. *
// PRECONDITION:  Queue is not empty           *
//*****************************************

template <class ItemType>
ItemType DynQueue<ItemType>::dequeue()
```

```
{
	NodeType *temp;
	ItemType returnItem;

	returnItem = front->info;
	temp = front;
	front = front->next;
	delete temp;
	numItems--;
	return returnItem;
}

//*********************************************
// Function isEmpty returns true if the queue *
// is empty, and false otherwise.              *
//*********************************************

template <class ItemType>
bool DynQueue<ItemType>::isEmpty()
{
	bool status;

	if (numItems > 0)
		status = false;
	else
		status = true;
	return status;
}

//*****************************************************
// Member function isFull is assumed to be false.     *
// Tailor to local operating environment.             *
//*****************************************************

template <class ItemType>
bool DynQueue<ItemType>::isFull()
{
	return false;
}

//*******************************************
// Function clear dequeues all the elements  *
// in the queue.                              *
//*******************************************

template <class ItemType>
void DynQueue<ItemType>::clear()
{
	ItemType value;   // Dummy variable for dequeue

	while (!isEmpty())
		value = dequeue();
}
```