Omar Alzibdeh
1945261

3320 Assignment 2

Leiss

s.omaralzibdeh@gmail.com

Q1a:
In the sense that we have to check each linear lists elements to see if they match, In the worst worst worst possible case in which we have to check each element we would have a lower bound of **O(n)**
Q1b:
-***Begin by creating a duplicate of Matrix A, labeling it as Matrix D.***

```
int[][] D = new int[n][n];
for (int i = 0; i < n; i++) {
   for (int j = 0; j < n; j++) {
      D[i][j] = A[i][j];
   }
}
```

-***Traverse through Matrix B, reducing each element by a single unit.***

```
for (int i = 0; i < n; i++) {
   for (int j = 0; j < n; j++) {
      B[i][j]--;
   }
}
```

-***Perform an element-wise comparison between Matrix A and the duplicate Matrix D to determine if they occupy distinct memory spaces.***

```
boolean isMemoryShared = false;
for (int i = 0; i < n; i++) {
   for (int j = 0; j < n; j++) {
      if (D[i][j] != A[i][j]) {
         isMemoryShared = true;
         break;
      }
   }
   if (isMemoryShared) {
      break;
   }
}
```

-***Reinstate the original values of Matrix B by incrementing the decremented elements.***

```
for (int i = 0; i < n; i++) {
   for (int j = 0; j < n; j++) {
      B[i][j]++;
   }
}
```

This algorithm has a lower bound complexity of **O(n)**, which makes it the same as our previous lower bound.

Q2:
In order to get a better version of the established method we used in class, we will use a refined method for the Insert and Delete procedures.The new algorithm will use K a bit differently by cutting it to one third of n. We then will enhance the insert and delete to work with this adjustment.

```
int thirdIndex = n / 3;
void nodeAdjustment(node* currentNode, int thirdIndex)
{
  if (thirdIndex < currentNode->value)
  {
    nodeAdjustment(currentNode->left, thirdIndex); // Recurse on left child
    // Operations to move the current node to the right subtree are performed here
    // The left child node is promoted to the root position
    // The tree is balanced while the root remains fixed
  }
  else if (thirdIndex > currentNode->value)
  {
    nodeAdjustment(currentNode->right, thirdIndex); // Recurse on right child
    // Operations to move the current node to the left subtree are performed here
    // The right child node is promoted to the root position
    // The tree is balanced while the root remains fixed
  }
  else
  {
    // If the current node is at the thirdIndex, no adjustment is needed
    return;
  }
}
```

The complexity of both space and time in the insert/delete function is O(1) because of its access to the value. This makes it better than the previous complexity by a large margin and grants us faster times. We are also granted a find function which also changes to O(1).

Q3:
What is “average work”?
Average work in terms of this question would be the hypothesized mean number of scalar multiplications; it would then need to be averaged over all possible ways the matrices can be parenthesized. In order to get the average work, we have to consider every possible way to fully parenthesize the product, get the number of scalar multiplications for each of the parenthesizations, then get the sum of the multiplications, and finally divide the sum to get the average. This will result in your “average work” term.

What is its time complexity?
O(n^3)

What is its space complexity?
O(n^2)
Here is the algorithm to solve the average work:
—---
Using the textbook, we know to calculate the total using
total[i][k] + total[k+1][j] + p[i-1]*p[k]*p[j];

```
// Assume p[0...n] is filled with the matrix dimensions
int p[n+1];
double total[n+1][n+1]; // To store the total scalar multiplications
int count[n+1][n+1]; // To store the number of parenthesizations
// Function to compute the total multiplications and number of parenthesizations
void compute(int i, int j) {
    if (i == j) {
        count[i][j] = 1;
        total[i][j] = 0;
    } else if (count[i][j] == 0) { // Uncomputed cell
        total[i][j] = 0;
        count[i][j] = 0;
        for (int k = i; k < j; k++) {
            compute(i, k);
            compute(k+1, j);
            // Add the number of multiplications for this parenthesization
            double multiplications = total[i][k] + total[k+1][j] + p[i-1]*p[k]*p[j];
            // Multiply the number of ways to parenthesize the left and right sub chains
            int ways = count[i][k] * count[k+1][j];
            total[i][j] += multiplications * ways;
            count[i][j] += ways;
        }
    }
}
// To calculate the average work
compute(1, n);
double averageWork = total[1][n] / count[1][n];
```

Q4:

N	M=1677721600	M=13421772800
16	25.16	206.40
64	24.86	204.21
256	26.57	206.45
1024	32.03	255.94
4096	116.71	954.04
16384	142.19	1147.04

Seconds

Quick comment:
I specifically chose c++ to run this code because I knew it was going to need to be faster than something like python; my computer cannot handle an algorithm like this very easily. My reasoning for this code taking a long time for some of the times is because i had to run this code through a linux virtual machine and not through windows. Running a system through another system often leads to a lot of background interference, which is what I suspect is the cause for such high times. In some of the instances, I believe that simply closing out tabs and programs on my computer may have caused the times to have big differences between them.
Explanation:
Now for the actual explanation of what is happening in the code. The reason for the sudden increase after N=1024 is because at some point, the IDE (OS?) starts to substitute virtual memory for physical memory. It essentially has to do this because there simply isn't enough to go around for the code to continue running. Hence, the longer times. With these times, it becomes pretty clear that we have a computational complexity of O(m) since we have to go through and loop ‘m’ times to be able to get our sum.

Q4 code:

```
#include <iostream>
#include <vector>
#include <random>
#include <chrono>

class Matrix {
public:
    Matrix(size_t size) : size_(size), data_(size, std::vector<int>(size, 0)) {}

    void AddValue(int a, int b, int x) {
        data_[a][b] += x;
    }
```

```
private:
   size_t size_;
   std::vector<std::vector<int>> data_;
};

int main() {
   std::vector<int> n_values = {16, 64, 256, 1024, 4096, 16384};
   std::default_random_engine generator(static_cast<unsigned
int>(std::chrono::system_clock::now().time_since_epoch().count()));
   std::uniform_int_distribution<int> distribution(1, 100);

   for (int n : n_values) {
      Matrix matrix(n);
      const size_t m = 134217728; //or 1677721600

      auto start_time = std::chrono::high_resolution_clock::now();
      for (size_t i = 0; i < m; ++i) {
         int x = distribution(generator);
         int a = generator() % n;
         int b = generator() % n;
         matrix.AddValue(a, b, x);
      }

      auto end_time = std::chrono::high_resolution_clock::now();
      std::chrono::duration<double> diff = end_time - start_time;
      std::cout << "Time for n size " << n << ": " << diff.count() << " seconds\n";
   }

   return 0;
}
```

Q5:

Quick comment:

This was by far the worst question personally. I think because the question was so long, I kept having to reread and redo my code constantly. I did eventually get it to work, it only took a couple days but it does work now. Also, for the life of me I could not get this code to be short. I don't know if it is because I used python instead of C++ (my preferred language). But I simply could not get it to be short compared to my other codes.

Explanation:

```
[Running] python -u "/home/vboxuser/c++/3320a2q5.py"
average initial insertion time: 4.0657 milliseconds
average insertion time: 2.1215 milliseconds
average deletion time: 0.9015 milliseconds
```

I ran all of this through a linux VM (not a VMM on windows) because linux seems to be much faster in coding memory. I also used python because I wanted to see if it would be slow compared to a language like c++. From the results,it does not seem as though the code was really that much slower, It really just seemed about the same.

This code helps us understand what the average time is to put in the first 50 nodes into the code. The program is meant to create memory fragmentation which means the code needs memory compaction to fix these gaps that occur. In theory, when the code starts to edit these nodes, the average insertion time should start to increase compared to the initial insertion time;however this does not happen. More than likely this is due to my computer having a really good memory management system that stops such a syntax from happening. Also, deletion times are consistently faster than the initial times. This makes perfect sense considering that deallocating memory will always be faster than allocating memory (as long as it's not interfered with by adding overhead).

WIth this, we can easily say that our code helped prove the experiment. We found memory fragmentation has occurred, which then triggered garbage collection and also led to memory compaction

Q5 Code:

(partially borrowed code from Geeks4Geeks)

```
import numpy as np
import random
import time

class TreeNode:
   def __init__(self, key, val):
      self.key = key
      self.val = val
      self.matrix = np.zeros((128, 128))
      self.left = None
      self.right = None
      self.height = 1

class AVLTree:
   def insert(self, root, key, val):
      if not root:
         return TreeNode(key, val)
      if key < root.key:
         root.left = self.insert(root.left, key, val)
      else:
         root.right = self.insert(root.right, key, val)
      root.height = 1 + max(self.getHeight(root.left), self.getHeight(root.right))
      b = self.getBalance
      if b(root) > 1 and key < root.left.key: return self.rightRotate(root)
      if b(root) < -1 and key > root.right.key: return self.leftRotate(root)
      if b(root) > 1 and key > root.left.key: root.left = self.leftRotate(root.left); return
self.rightRotate(root)
      if b(root) < -1 and key < root.right.key: root.right = self.rightRotate(root.right);
return self.leftRotate(root)
      return root

   def delete(self, root, key):
      if not root: return root
      if key < root.key: root.left = self.delete(root.left, key)
      elif key > root.key: root.right = self.delete(root.right, key)
      else:
         if not root.left: return root.right
         elif not root.right: return root.left
         t = self.getMinValueNode
         root.key = t(root.right).key
         root.right = self.delete(root.right, t(root.right).key)
```

```
        root.height = 1 + max(self.getHeight(root.left), self.getHeight(root.right))
        b = self.getBalance
        if b(root) > 1 and b(root.left) < 0: root.left = self.leftRotate(root.left); return self.rightRotate(root)
        if b(root) < -1 and b(root.right) <= 0: return self.leftRotate(root)
        if b(root) > 1 and b(root.left) >= 0: return self.rightRotate(root)
        if b(root) < -1 and b(root.right) > 0: root.right = self.rightRotate(root.right); return self.leftRotate(root)
        return root

    def balance(self, root):
        root.height = 1 + max(self.getHeight(root.left), self.getHeight(root.right))
        b = self.getBalance
        if b(root) > 1 and root.left and root.left.key > root.key: return self.rightRotate(root)
        if b(root) < -1 and root.right and root.right.key < root.key: return self.leftRotate(root)
        if b(root) > 1 and root.left and root.left.key < root.key: root.left = self.leftRotate(root.left); return self.rightRotate(root)
        if b(root) < -1 and root.right and root.right.key > root.key: root.right = self.rightRotate(root.right); return self.leftRotate(root)
        return root

    def leftRotate(self, z):
        y = z.right
        T2 = y.left
        y.left = z
        z.right = T2
        z.height = 1 + max(self.getHeight(z.left), self.getHeight(z.right))
        y.height = 1 + max(self.getHeight(y.left), self.getHeight(y.right))
        return y

    def rightRotate(self, y):
        x = y.left
        T2 = x.right
        x.right = y
        y.left = T2
        y.height = 1 + max(self.getHeight(y.left), self.getHeight(y.right))
        x.height = 1 + max(self.getHeight(x.left), self.getHeight(x.right))
        return x

    def getHeight(self, root):
        return root.height if root else 0

    def getBalance(self, root):
```

```
        return self.getHeight(root.left) - self.getHeight(root.right) if root else 0

    def getMinValueNode(self, root):
        return self.getMinValueNode(root.left) if root and root.left else root

avl = AVLTree()
root = None

inserted_keys = []

insert_times = []
delete_times = []

for _ in range(1000):
    key = random.randint(0, 10000)
    val = random.randint(0, 299)
    inserted_keys.append(key)
    start_time = time.time()
    root = avl.insert(root, key, val)
    end_time = time.time()
    insert_times.append((end_time - start_time) * 1000)
while len(inserted_keys) > 50:
    key_to_delete = random.choice(inserted_keys)
    start_time = time.time()
    root = avl.delete(root, key_to_delete)
    end_time = time.time()
    delete_times.append((end_time - start_time) * 1000)
    inserted_keys.remove(key_to_delete)

avg_initial_insert_time = sum(insert_times[:50]) / 50
avg_insert_time = sum(insert_times) / len(insert_times)
avg_delete_time = sum(delete_times) / len(delete_times)

print("average initial insertion time: {:.6f} milliseconds".format(avg_initial_insert_time))
print("average insertion time: {:.6f} milliseconds".format(avg_insert_time))
print("average deletion time: {:.6f} milliseconds".format(avg_delete_time))
```

Q6:

Output:

Cache Size: 0.5M
Available Physical Memory: 26754273280
Available Page File: 9504972800
Available Virtual Memory: 140733134274560
Time elapsed: 15 microseconds
Cache Size: 0.6M
Available Physical Memory: 26754273280
Available Page File: 9504972800
Available Virtual Memory: 140733134274560
Time elapsed: 20 microseconds
Cache Size: 0.7M
Available Physical Memory: 26754273280
Available Page File: 9504972800
Available Virtual Memory: 140733134274560
Time elapsed: 11 microseconds
Cache Size: 0.8M
Available Physical Memory: 26754273280
Available Page File: 9504972800
Available Virtual Memory: 140733134274560
Time elapsed: 11 microseconds
Cache Size: 0.9M
Available Physical Memory: 26754273280
Available Page File: 9504972800
Available Virtual Memory: 140733134274560
Time elapsed: 15 microseconds
Cache Size: 0.95M
Available Physical Memory: 26754273280
Available Page File: 9504972800
Available Virtual Memory: 140733134274560
Time elapsed: 9 microseconds
Cache Size: 0.99M
Available Physical Memory: 26754273280
Available Page File: 9504972800
Available Virtual Memory: 140733134274560
Time elapsed: 8 microseconds
Cache Size: 1M
Available Physical Memory: 26754273280
Available Page File: 8517857280
Available Virtual Memory: 140732149690368
Time elapsed: 583923 microseconds
Cache Size: 1.01M
Available Physical Memory: 26769551360

Available Page File: 8515907584
Available Virtual Memory: 140732149673984
Time elapsed: 612808 microseconds
Cache Size: 1.1M
Available Physical Memory: 26774609920
Available Page File: 8515231744
Available Virtual Memory: 140732149653504
Time elapsed: 596981 microseconds
Cache Size: 1.5M
Available Physical Memory: 26774224896
Available Page File: 8515440640
Available Virtual Memory: 140732149682176
Time elapsed: 595297 microseconds
Cache Size: 2M
Available Physical Memory: 26773028864
Available Page File: 7527600128
Available Virtual Memory: 140731165085696
Time elapsed: 1184618 microseconds
Cache Size: 5M
Available Physical Memory: 26770812928
Available Page File: 8870707200
Available Virtual Memory: 140732506435584
Time elapsed: 400801 microseconds
Cache Size: 10M
Available Physical Memory: 26776363008
Available Page File: 8243937280
Available Virtual Memory: 140731878596608
Time elapsed: 739357 microseconds
Cache Size: 50M
Available Physical Memory: 26782232576
Available Page File: 7521292288
Available Virtual Memory: 140731150913536
Time elapsed: 1207178 microseconds

Explanation:

With this much ram, it was hard to be able to see thrashing actually occur; though, it does occur. Looking at the results, it is obvious that thrashing occurs. Given that the 'Available Physical Memory' does not change significantly across the different cache sizes, the sharp increase in operation time when the cache size exceeds 1 M indicates that the system has started to swap heavily to maintain the working set of the program. This is a strong indicator of thrashing, as the system is likely continually moving data between the disk (swap space) and the physical memory, leading to high latency in memory access and a decrease in overall system performance.

Q6 code:

```
#include <iostream>
#include <chrono>
#include <unistd.h>
#include <sys/sysinfo.h>
#include <climits>
#include <vector>

using namespace std;
using namespace std::chrono;

constexpr int CACHE_SIZES_COUNT = 15;
const double CACHE_SIZES_MB[CACHE_SIZES_COUNT] = {0.5, 0.6, 0.7, 0.8, 0.9, 0.95,
0.99, 1.0, 1.01, 1.1, 1.5, 2, 5, 10, 50};
constexpr double MB_TO_BYTES = 1024 * 1024;

void printSysInfo() {
   struct sysinfo info;
   sysinfo(&info);
   cout << "Available Physical Memory: " << info.freeram << endl;
   cout << "Available Swap Space: " << info.freeswap << endl;
}

void simulateOperationsOnArray(int* array, int size) {
   for (int i = 0; i < size; i++) {
      array[i] = i;
   }
}

int main() {
   printSysInfo();

   for (int i = 0; i < CACHE_SIZES_COUNT; i++) {
      double cacheSizeBytes = CACHE_SIZES_MB[i] * MB_TO_BYTES;
      cout << "\nCache Size: " << CACHE_SIZES_MB[i] << " MB (" << cacheSizeBytes <<
" Bytes)" << endl;

      long numBytes = static_cast<long>(cacheSizeBytes);
      if (numBytes <= 0 || numBytes > LONG_MAX / sizeof(int)) {
         cerr << "Invalid number of bytes: " << numBytes << endl;
         continue;
      }
```

```
        int arraySize = numBytes / sizeof(int);
        if (arraySize <= 0 || arraySize > INT_MAX) {
            cerr << "Invalid array size: " << arraySize << endl;
            continue;
        }

        vector<int> numArray(arraySize);
        simulateOperationsOnArray(&numArray[0], arraySize);

        auto startTime = high_resolution_clock::now();
        auto endTime = high_resolution_clock::now();

        cout << "Time elapsed for operations: "
             << duration_cast<milliseconds>(endTime - startTime).count()
             << " milliseconds" << endl;
    }

    cout << "\nProcess completed. Press enter to exit.";
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
    return 0;
}
```

Q7:
To write this code, i borrowed an implementation from geeksforgeeks (https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/#). I wrote it in C++ just for the sake of personal preference. I personally know C++, therefore it was easier for me to adjust and write the code for me.

What is dijkstra's algorithm?
To put it in terms that anyone can understand, dijkstra's algorithm is essentially a treasure hunt game. If there are multiple spots to travel to in order to find the treasure, It is best to find a path of least time taken in order to be as efficient as possible. If there are multiple treasure spots, the game is over when the quickest route to all the spots has been achieved.

Example 1:

```
g.addEdge(0, 1, 4);
g.addEdge(0, 7, 8);
g.addEdge(1, 2, 8);
g.addEdge(1, 7, 11);
g.addEdge(2, 3, 7);
g.addEdge(2, 8, 2);
g.addEdge(2, 5, 4);
g.addEdge(3, 4, 9);
g.addEdge(3, 5, 14);
g.addEdge(4, 5, 10);
g.addEdge(5, 6, 2);
g.addEdge(6, 7, 1);
g.addEdge(6, 8, 6);
g.addEdge(7, 8, 7);
```

Output:

Vertex	Distance	Path
0 -> 0	0	0
0 -> 1	4	0 1
0 -> 2	12	0 1 2
0 -> 3	19	0 1 2 3
0 -> 4	28	0 1 2 3 4
0 -> 5	16	0 1 2 5
0 -> 6	18	0 1 2 5 6
0 -> 7	8	0 7
0 -> 8	14	0 1 2 8

Example 2:

```
g.addEdge(0, 1, 6);
g.addEdge(0, 2, 3);
g.addEdge(1, 3, 5);
g.addEdge(1, 4, 3);
g.addEdge(2, 3, 7);
g.addEdge(2, 5, 8);
```

```
g.addEdge(3, 4, 2);
g.addEdge(3, 5, 4);
g.addEdge(4, 5, 6);
g.addEdge(5, 7, 3);
g.addEdge(5, 6, 9);
g.addEdge(6, 7, 6);
g.addEdge(6, 8, 2);
g.addEdge(7, 8, 7);
```

Output:

Vertex	Distance	Path
0 -> 0	0	0
0 -> 1	6	0 1
0 -> 2	3	0 2
0 -> 3	10	0 2 3
0 -> 4	9	0 1 4
0 -> 5	11	0 2 5
0 -> 6	20	0 2 5 6
0 -> 7	14	0 2 5 7
0 -> 8	21	0 2 5 7 8

Example 3:

```
g.addEdge(0, 1, 2);
g.addEdge(0, 3, 1);
g.addEdge(1, 2, 7);
g.addEdge(1, 4, 3);
g.addEdge(1, 5, 8);
g.addEdge(2, 6, 4);
g.addEdge(3, 7, 5);
g.addEdge(4, 8, 3);
g.addEdge(5, 6, 2);
g.addEdge(6, 8, 9);
g.addEdge(7, 8, 6);
```

Output:

Vertex	Distance	Path
0 -> 0	0	0
0 -> 1	2	0 1
0 -> 2	9	0 1 2
0 -> 3	1	0 3
0 -> 4	5	0 1 4
0 -> 5	10	0 1 5
0 -> 6	12	0 1 5 6
0 -> 7	6	0 3 7
0 -> 8	8	0 1 4 8

Example 4:

```
g.addEdge(0, 4, 5);
```

g.addEdge(0, 3, 9);
g.addEdge(0, 1, 10);
g.addEdge(1, 2, 1);
g.addEdge(1, 3, 3);
g.addEdge(1, 4, 4);
g.addEdge(2, 5, 7);
g.addEdge(3, 5, 2);
g.addEdge(3, 6, 3);
g.addEdge(4, 6, 2);
g.addEdge(5, 7, 1);
g.addEdge(6, 7, 11);
g.addEdge(6, 8, 1);
g.addEdge(7, 8, 6);

Output:

Vertex	Distance	Path
0 -> 0	0	0
0 -> 1	10	0 1
0 -> 2	11	0 1 2
0 -> 3	9	0 3
0 -> 4	5	0 4
0 -> 5	11	0 3 5
0 -> 6	7	0 4 6
0 -> 7	12	0 3 5 7
0 -> 8	8	0 4 6 8

Example 5:

g.addEdge(0, 2, 2);
g.addEdge(0, 5, 9);
g.addEdge(0, 6, 14);
g.addEdge(1, 0, 3);
g.addEdge(1, 3, 4);
g.addEdge(2, 1, 8);
g.addEdge(2, 3, 7);
g.addEdge(3, 4, 1);
g.addEdge(4, 5, 5);
g.addEdge(5, 3, 2);
g.addEdge(6, 2, 6);
g.addEdge(6, 7, 2);
g.addEdge(7, 8, 3);
g.addEdge(8, 6, 4);

Output:

Vertex	Distance	Path
0 -> 0	0	0
0 -> 1	10	0 2 1
0 -> 2	2	0 2

0 -> 3	9	0 2 3
0 -> 4	10	0 2 3 4
0 -> 5	9	0 5
0 -> 6	14	0 6
0 -> 7	16	0 6 7
0 -> 8	19	0 6 7 8

Example 6:

```
g.addEdge(0, 1, 2);
g.addEdge(0, 3, 6);
g.addEdge(1, 4, 3);
g.addEdge(3, 5, 1);
g.addEdge(4, 6, 1);
g.addEdge(5, 7, 5);
g.addEdge(6, 8, 4);
g.addEdge(7, 2, 2);
```

Output:

Vertex	Distance	Path
0 -> 0	0	0
0 -> 1	2	0 1
0 -> 2	14	0 3 5 7 2
0 -> 3	6	0 3
0 -> 4	5	0 1 4
0 -> 5	7	0 3 5
0 -> 6	6	0 1 4 6
0 -> 7	12	0 3 5 7
0 -> 8	10	0 1 4 6 8

Example 7:

```
g.addEdge(0, 2, 6);
g.addEdge(0, 1, 7);
g.addEdge(0, 3, 9);
g.addEdge(1, 3, 3);
g.addEdge(1, 4, 9);
g.addEdge(2, 1, 2);
g.addEdge(2, 4, 1);
g.addEdge(3, 4, 2);
g.addEdge(3, 5, 1);
g.addEdge(4, 5, 4);
g.addEdge(5, 6, 2);
g.addEdge(6, 4, 7);
g.addEdge(6, 7, 4);
g.addEdge(7, 5, 3);
g.addEdge(4, 8, 5);
g.addEdge(8, 7, 1);
```

Output:

Vertex	Distance	Path
0 -> 0	0	0
0 -> 1	7	0 1
0 -> 2	6	0 2
0 -> 3	9	0 3
0 -> 4	7	0 2 4
0 -> 5	10	0 3 5
0 -> 6	12	0 3 5 6
0 -> 7	13	0 2 4 8 7
0 -> 8	12	0 2 4 8

Q7 code:

```
#include <iostream>
#include <vector>
#include <queue>
#include <limits>
#include <utility>
Using namespace std;


class Graph {
   int numVertices; // Number of vertices
   vector<vector<pair<int, int>>> adjList; // Adjacency list to store nodes and weights

public:
   explicit Graph(int V) : numVertices(V), adjList(V) {}

   void addEdge(int u, int v, int w) {
      adjList[u].emplace_back(v, w);
      // For undirected graph, add: adjList[v].emplace_back(u, w);
   }

   void printPath(const vector<int>& predecessor, int j) const {
      if (predecessor[j] == -1) return;
      printPath(predecessor, predecessor[j]);
      cout << j << " ";
   }

   void dijkstra(int src) const {
      priority_queue<pair<int, int>, vector<pair<int, int>>, std::greater<>> pq;
      vector<int> dist(numVertices, std::numeric_limits<int>::max());
      vector<int> predecessor(numVertices, -1);

      pq.emplace(0, src);
```

```
        dist[src] = 0;

        while (!pq.empty()) {
            int u = pq.top().second;
            pq.pop();

            for (const auto& [v, weight] : adjList[u]) {
                if (dist[v] > dist[u] + weight) {
                    dist[v] = dist[u] + weight;
                    predecessor[v] = u;
                    pq.emplace(dist[v], v);
                }
            }
        }

        cout << "Vertex\tDistance\tPath" << endl;
        for (int i = 0; i < numVertices; ++i) {
            cout << src << " -> " << i << "\t" << dist[i] << "\t\t" << src << " ";
            printPath(predecessor, i);
            cout << endl;
        }
    }
};

int main() {
    int V = 9;
    Graph g(V);
    // Add edges as needed
    // Example: g.addEdge(0, 1, 4);

    g.dijkstra(0);
    return 0;
}
```