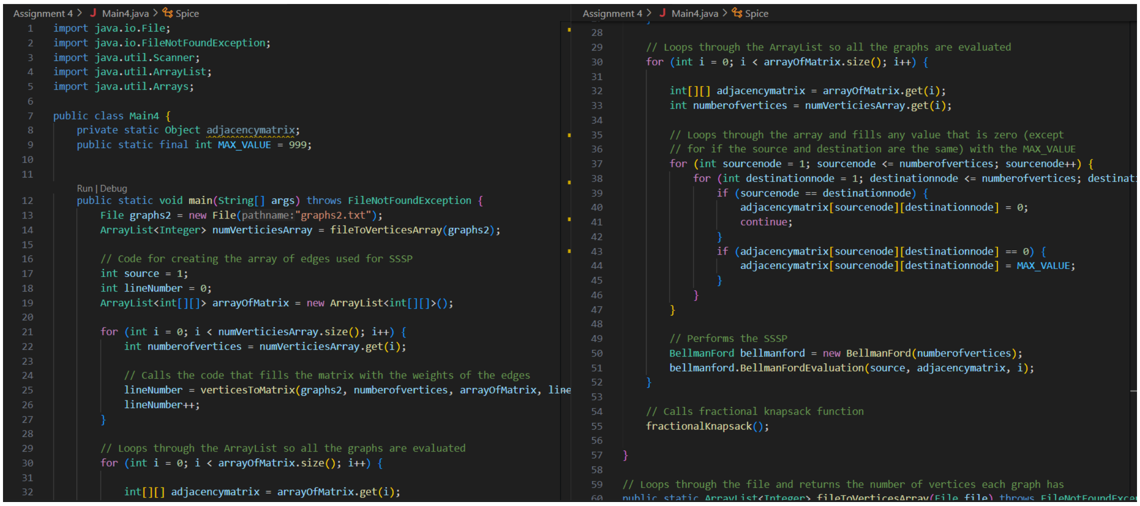
Algorithms Assignment 4

Thomas Breaton

December 2023

1 Main Function



```java
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
import java.util.ArrayList;
import java.util.Arrays;

public class Main4 {
    private static Object adjacencymatrix;
    public static final int MAX_VALUE = 999;


    public static void main(String[] args) throws FileNotFoundException {
        File graphs2 = new File(pathname:"graphs2.txt");
        ArrayList<Integer> numVerticiesArray = fileToVerticesArray(graphs2);

        // Code for creating the array of edges used for SSSP
        int source = 1;
        int lineNumber = 0;
        ArrayList<int[][]> arrayOfMatrix = new ArrayList<int[][]>();

        for (int i = 0; i < numVerticiesArray.size(); i++) {
            int numberofvertices = numVerticiesArray.get(i);

            // Calls the code that fills the matrix with the weights of the edges
            lineNumber = verticesToMatrix(graphs2, numberofvertices, arrayOfMatrix, line
            lineNumber++;
        }

        // Loops through the ArrayList so all the graphs are evaluated
        for (int i = 0; i < arrayOfMatrix.size(); i++) {

            int[][] adjacencymatrix = arrayOfMatrix.get(i);
            int numberofvertices = numVerticiesArray.get(i);

            // Loops through the array and fills any value that is zero (except
            // for if the source and destination are the same) with the MAX_VALUE
            for (int sourcenode = 1; sourcenode <= numberofvertices; sourcenode++) {
                for (int destinationnode = 1; destinationnode <= numberofvertices; destinat
                    if (sourcenode == destinationnode) {
                        adjacencymatrix[sourcenode][destinationnode] = 0;
                        continue;
                    }
                    if (adjacencymatrix[sourcenode][destinationnode] == 0) {
                        adjacencymatrix[sourcenode][destinationnode] = MAX_VALUE;
                    }
                }
            }

            // Performs the SSSP
            BellmanFord bellmanford = new BellmanFord(numberofvertices);
            bellmanford.BellmanFordEvaluation(source, adjacencymatrix, i);
        }

        // Calls fractional knapsack function
        fractionalKnapsack();

    }

    // Loops through the file and returns the number of vertices each graph has
```

Figure 1: Image of main function

This section of code consists of what is being imported, constants, and the code that I want to run. The main function in this assignment calls for files to be scanned, the array for Bellman-Ford SSSP to be created and filled, the SSSP to be executed, and the fractional knapsack function to be called.

2 File To Vertices Array

Assignment 4 > Main4.java > Spice

```java
    // Loops through the file and returns the number of vertices each graph has
    public static ArrayList<Integer> fileToVerticesArray(File file) throws FileNotFoundException {
        Scanner scan = new Scanner(file);
        int numVertices = 0;
        ArrayList<Integer> numVerticesArray = new ArrayList<Integer>();

        // Scans each line looking for a specific string
        while (scan.hasNextLine()) {
            String data = scan.nextLine();
            String delims = "[ ]+";
            String[] tempString = data.split(delims);
            // There is a new line so the graph is done being created, it can be added to
            // the ArrayList
            if (tempString[0].compareTo(anotherString:"") == 0) {
                numVerticesArray.add(numVertices);
                numVertices = 0;
                // Creates the new graph
            } else if (tempString[0].compareTo(anotherString:"new") == 0) {

            } else if (tempString[0].compareTo(anotherString:"--") == 0) {
                // Adds the vertexes to the graph
            } else if (tempString[1].compareTo(anotherString:"vertex") == 0) {
                numVertices++;
                // Adds the edges to the graph
            } else if (tempString[1].compareTo(anotherString:"edge") == 0) {

            }
        }
        numVerticesArray.add(numVertices);
        scan.close();
        return numVerticesArray;
    }
```

Figure 2: Image of verticesToArray function

This function reads the file graphs2 and keeps a count of how many vertices are in each graph. Then that number is added to an ArrayList which is returned and used to create the array for Bellman-Ford to be implemented.

3 Vertices To Matrix

Assignment 4 > Main4.java > Main4 > fileToVerticesArray(File)

```java
    // Takes the information in the file and fills out an array, then adds that array to an ArrayList
    // Uses line number so the same graph isn't put into the arrayList twice
    public static int verticesToMatrix(File file, int numVertices, ArrayList<int[][]> arrayOfMatrix, int lineNumber)
            throws FileNotFoundException {
        Scanner scan = new Scanner(file);
        int adjacencymatrix[][] = new int[numVertices + 1][numVertices + 1];

        // Scans each line looking for a specific string
        for (int i = 0; i < lineNumber; i++) {
            if (scan.hasNextLine()) {
                scan.nextLine();
            }
        }
        while (scan.hasNextLine()) {
            String data = scan.nextLine();
            String delims = "[ ]+";
            String[] tempString = data.split(delims);
            // There is a new line so the graph is done being created, it can be added to
            // the ArrayList
            if (tempString[0].compareTo(anotherString:"") == 0) {
                arrayOfMatrix.add(adjacencymatrix);
                return lineNumber;
                // Creates the new graph
            } else if (tempString[0].compareTo(anotherString:"new") == 0) {

            } else if (tempString[0].compareTo(anotherString:"--") == 0) {
                // Adds the vertexes to the graph
            } else if (tempString[1].compareTo(anotherString:"vertex") == 0) {
                // Adds the edges to the graph
            } else if (tempString[1].compareTo(anotherString:"edge") == 0) {
                adjacencymatrix[Integer.parseInt(tempString[2])][Integer.parseInt(tempString[4])] = Integer
                        .parseInt(tempString[5]);
            }
            lineNumber++;
        }
        // Adds the final graph from the file to the ArrayList
        arrayOfMatrix.add(adjacencymatrix);
        scan.close();
        return lineNumber;
    }
```

Figure 3: Image of verticesToMatrix function

This function is takes the number of vertices and an ArrayList and reads the file graphs2 again. This time, the weights of the edges are put into the matrix. The function uses an int to track the line number so that graphs aren't added multiple times into the ArrayList.

4 File To Spice Info

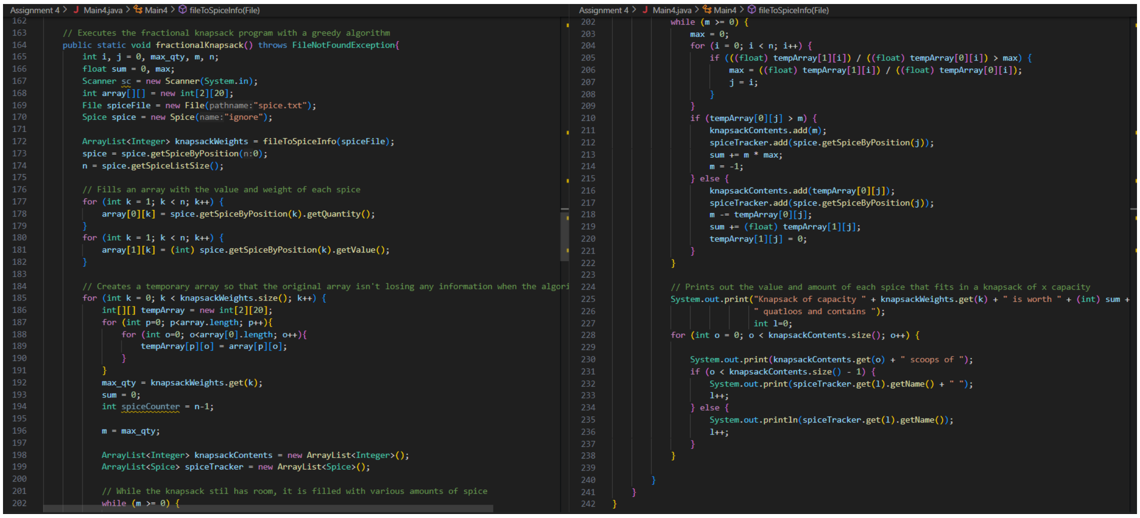
Assignment 4 > Main4.java > Main4 > verticesToMatrix(File, int, ArrayList<int[][]>, int)

```java

    // Loops through the file and puts all the information about the spices into an ArrayList
    // Returns the max weights each knapsack can hold
    public static ArrayList<Integer> fileToSpiceInfo(File file) throws FileNotFoundException {
        Scanner scan = new Scanner(file);
        ArrayList<Integer> knapsackWeights = new ArrayList<Integer>();

        while (scan.hasNextLine()) {
            String data = scan.nextLine();
            String delim1 = ";";
            String delim2 = "[ ]+";
            String[] tempString1 = data.split(delim1);
            String[] tempString2 = tempString1[0].split(delim2);
            String tempString3 = tempString2[0];
            if (tempString3.compareTo(anotherString:"") == 0) {

            } else if (tempString3.compareTo(anotherString:"--") == 0) {

            } else if (tempString3.compareTo(anotherString:"spice") == 0) {
                Spice tempSpice = new Spice(tempString2[3]);
                tempString2 = tempString1[1].split(delim2);
                tempSpice.setValue(Float.parseFloat(tempString2[3]));
                tempString2 = tempString1[2].split(delim2);
                tempSpice.setQuantity(Integer.parseInt(tempString2[3]));
            } else if (tempString3.compareTo(anotherString:"knapsack") == 0) {
                knapsackWeights.add(Integer.parseInt(tempString2[3]));
            }
        }
        return knapsackWeights;
    }
```

Figure 4: Image of fileToSpiceInfo function

This function reads through the spice file and creates the spices using their name, value, and quantity. These spices are added to an ArrayList in the Spice class. This function also creates an ArrayList of the different max weights that a knapsack can hold.

5 Fractional Knapsack



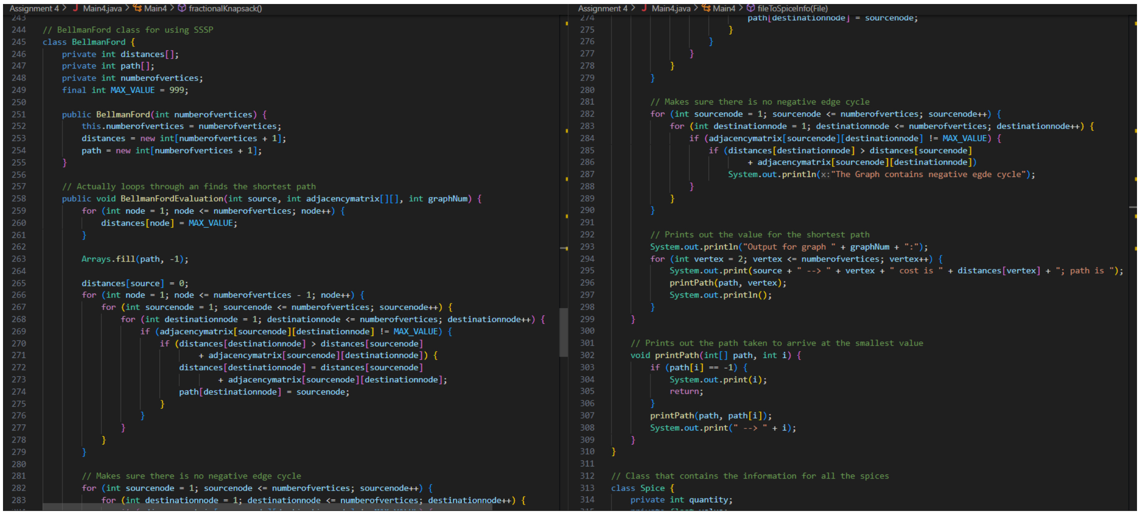
```
// Executes the fractional knapsack program with a greedy algorithm
public static void fractionalKnapsack() throws FileNotFoundException{
    int i, j = 0, max_qty, m, n;
    float sum = 0, max;
    Scanner sc = new Scanner(System.in);
    int array[][] = new int[2][20];
    File spiceFile = new File(pathname:"spice.txt");
    Spice spice = new Spice(name:"ignore");

    ArrayList<Integer> knapsackWeights = fileToSpiceInfo(spiceFile);
    spice = spice.getSpiceByPosition(n:0);
    n = spice.getSpiceListSize();

    // Fills an array with the value and weight of each spice
    for (int k = 1; k < n; k++) {
        array[0][k] = spice.getSpiceByPosition(k).getQuantity();
    }
    for (int k = 1; k < n; k++) {
        array[1][k] = (int) spice.getSpiceByPosition(k).getValue();
    }

    // Creates a temporary array so that the original array isn't losing any information when the algori
    for (int k = 0; k < knapsackWeights.size(); k++) {
        int[][] tempArray = new int[2][20];
        for (int p=0; p<array.length; p++){
            for (int o=0; o<array[0].length; o++){
                tempArray[p][o] = array[p][o];
            }
        }
        max_qty = knapsackWeights.get(k);
        sum = 0;
        int spiceCounter = n-1;

        m = max_qty;

        ArrayList<Integer> knapsackContents = new ArrayList<Integer>();
        ArrayList<Spice> spiceTracker = new ArrayList<Spice>();

        // While the knapsack stil has room, it is filled with various amounts of spice
        while (m >= 0) {
```

```
        while (m >= 0) {
            max = 0;
            for (i = 0; i < n; i++) {
                if (((float) tempArray[1][i]) / ((float) tempArray[0][i]) > max) {
                    max = ((float) tempArray[1][i]) / ((float) tempArray[0][i]);
                    j = i;
                }
            }
            if (tempArray[0][j] > m) {
                knapsackContents.add(m);
                spiceTracker.add(spice.getSpiceByPosition(j));
                sum += m * max;
                m = -1;
            } else {
                knapsackContents.add(tempArray[0][j]);
                spiceTracker.add(spice.getSpiceByPosition(j));
                m -= tempArray[0][j];
                sum += (float) tempArray[1][j];
                tempArray[1][j] = 0;
            }
        }

        // Prints out the value and amount of each spice that fits in a knapsack of x capacity
        System.out.print("Knapsack of capacity " + knapsackWeights.get(k) + " is worth " + (int) sum +
                        " quatloos and contains ");
                        int l=0;
        for (int o = 0; o < knapsackContents.size(); o++) {

            System.out.print(knapsackContents.get(o) + " scoops of ");
            if (o < knapsackContents.size() - 1) {
                System.out.print(spiceTracker.get(l).getName() + " ");
                l++;
            } else {
                System.out.println(spiceTracker.get(l).getName());
                l++;
            }
        }

    }
  }
}
```

Figure 5: Image of fractionalKnapsack function

This section of code is what is used to determine how much of each spice can be taken in different knapsacks. The code creates an array that stores the weight and value of each spice and then loops through that array and finds the maximum value that can be taken while not going over the weight limit of the knapsack. The running time for this function is O(nlogn).

6 Bellman-Ford SSSP



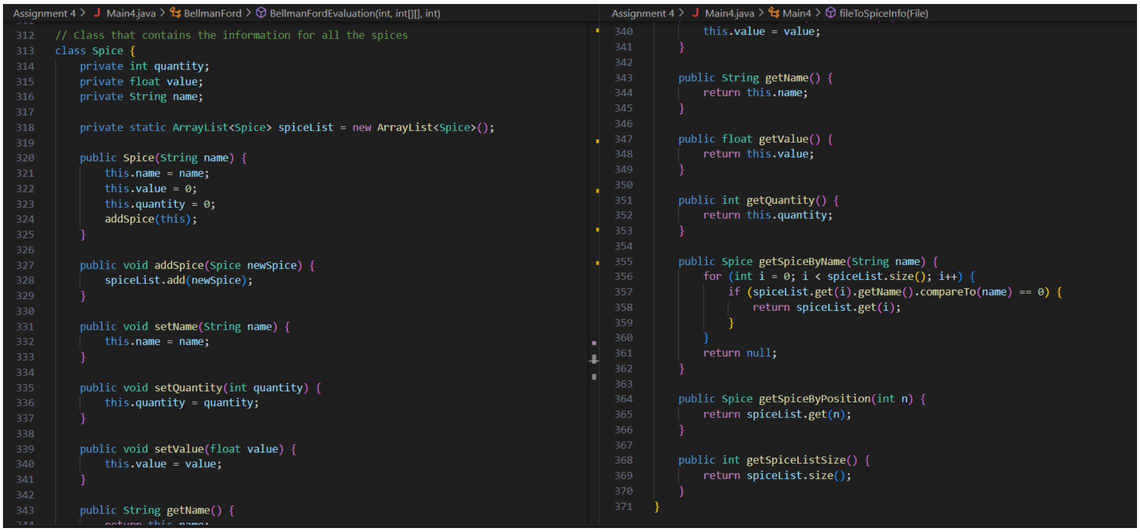

```
// BellmanFord class for using SSSP
class BellmanFord {
    private int distances[];
    private int path[];
    private int numberofvertices;
    final int MAX_VALUE = 999;

    public BellmanFord(int numberofvertices) {
        this.numberofvertices = numberofvertices;
        distances = new int[numberofvertices + 1];
        path = new int[numberofvertices + 1];
    }

    // Actually loops through an finds the shortest path
    public void BellmanFordEvaluation(int source, int adjacencymatrix[][], int graphNum) {
        for (int node = 1; node <= numberofvertices; node++) {
            distances[node] = MAX_VALUE;
        }

        Arrays.fill(path, -1);

        distances[source] = 0;
        for (int node = 1; node <= numberofvertices - 1; node++) {
            for (int sourcenode = 1; sourcenode <= numberofvertices; sourcenode++) {
                for (int destinationnode = 1; destinationnode <= numberofvertices; destinationnode++) {
                    if (adjacencymatrix[sourcenode][destinationnode] != MAX_VALUE) {
                        if (distances[destinationnode] > distances[sourcenode]
                                + adjacencymatrix[sourcenode][destinationnode]) {
                            distances[destinationnode] = distances[sourcenode]
                                    + adjacencymatrix[sourcenode][destinationnode];
                            path[destinationnode] = sourcenode;
                        }
                    }
                }
            }
        }

        // Makes sure there is no negative edge cycle
        for (int sourcenode = 1; sourcenode <= numberofvertices; sourcenode++) {
            for (int destinationnode = 1; destinationnode <= numberofvertices; destinationnode++) {
```

```
                            path[destinationnode] = sourcenode;
                        }
                    }
                }
            }
        }

        // Makes sure there is no negative edge cycle
        for (int sourcenode = 1; sourcenode <= numberofvertices; sourcenode++) {
            for (int destinationnode = 1; destinationnode <= numberofvertices; destinationnode++) {
                if (adjacencymatrix[sourcenode][destinationnode] != MAX_VALUE) {
                    if (distances[destinationnode] > distances[sourcenode]
                            + adjacencymatrix[sourcenode][destinationnode])
                        System.out.println(x:"The Graph contains negative egde cycle");
                }
            }
        }

        // Prints out the value for the shortest path
        System.out.println("Output for graph " + graphNum + ":");
        for (int vertex = 2; vertex <= numberofvertices; vertex++) {
            System.out.print(source + " --> " + vertex + " cost is " + distances[vertex] + "; path is ");
            printPath(path, vertex);
            System.out.println();
        }
    }

    // Prints out the path taken to arrive at the smallest value
    void printPath(int[] path, int i) {
        if (path[i] == -1) {
            System.out.print(i);
            return;
        }
        printPath(path, path[i]);
        System.out.print(" --> " + i);
    }
}

// Class that contains the information for all the spices
class Spice {
    private int quantity;
```

Figure 6: Image of BellmanFord Class

This section of code contains the Bellman-Ford class that is used to implement SSSP. This class has an array for the distance from the source to the destination, an array for the path that is taken to get there, and an int to store how many vertices are in the graph. When the evaluation function is called, the matrix that was filled with the edge weights is looped through and each vertex that isn't at infinity, or in this case 999, and is larger than the edge weight, it is changed to the lower value. This is repeated for the number of vertices that are in the graph. Then the function loops through the graph again and check for a negative edge cycle. Finally, the function prints out the value of the shortest path and what that path actually is. The running time for this algorithm is O(V * E) so it depends on how many vertices and edges are in your graph.

7 Spice Class




```java
// Class that contains the information for all the spices
class Spice {
    private int quantity;
    private float value;
    private String name;

    private static ArrayList<Spice> spiceList = new ArrayList<Spice>();

    public Spice(String name) {
        this.name = name;
        this.value = 0;
        this.quantity = 0;
        addSpice(this);
    }

    public void addSpice(Spice newSpice) {
        spiceList.add(newSpice);
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }

    public void setValue(float value) {
        this.value = value;
    }

    public String getName() {
```

```java
        this.value = value;
    }

    public String getName() {
        return this.name;
    }

    public float getValue() {
        return this.value;
    }

    public int getQuantity() {
        return this.quantity;
    }

    public Spice getSpiceByName(String name) {
        for (int i = 0; i < spiceList.size(); i++) {
            if (spiceList.get(i).getName().compareTo(name) == 0) {
                return spiceList.get(i);
            }
        }
        return null;
    }

    public Spice getSpiceByPosition(int n) {
        return spiceList.get(n);
    }

    public int getSpiceListSize() {
        return spiceList.size();
    }
}
```

Figure 7: Image of Spice Class

This section of code contains the Spice class. This class is used to store information about each of the spices like name, value, and weight. There are also get functions for easy access to the data. You can find spices by their name, or their position in the ArrayList of spices.

8 Results

```
PS C:\Users\Tom\Documents\GitHub\Algorithms>  & 'C:\Program Files\Java\jdk-21\bin\java.exe' '--enable-preview' '-XX:+ShowCodeDetailsInExceptionMessages'
ming\Code\User\workspaceStorage\4ec5a952ceb7004b334f690f214cc08b\redhat.java\jdt_ws\Algorithms_95ea08bd\bin' 'Main4'
Output for graph 0:
1 --> 2 cost is 2; path is 1 --> 4 --> 3 --> 2
1 --> 3 cost is 4; path is 1 --> 4 --> 3
1 --> 4 cost is 7; path is 1 --> 4
1 --> 5 cost is -2; path is 1 --> 4 --> 3 --> 2 --> 5
Output for graph 1:
1 --> 2 cost is 999; path is 2
1 --> 3 cost is 999; path is 3
1 --> 4 cost is 999; path is 4
1 --> 5 cost is 999; path is 5
1 --> 6 cost is 999; path is 6
1 --> 7 cost is 999; path is 7
Output for graph 2:
1 --> 2 cost is 1; path is 1 --> 2
1 --> 3 cost is 2; path is 1 --> 2 --> 3
1 --> 4 cost is 3; path is 1 --> 2 --> 3 --> 4
1 --> 5 cost is 1; path is 1 --> 5
1 --> 6 cost is 1; path is 1 --> 6
1 --> 7 cost is 2; path is 1 --> 5 --> 7
Output for graph 3:
1 --> 2 cost is 2; path is 1 --> 2
1 --> 3 cost is 6; path is 1 --> 2 --> 5 --> 3
1 --> 4 cost is 7; path is 1 --> 2 --> 5 --> 3 --> 4
1 --> 5 cost is 1; path is 1 --> 2 --> 5
1 --> 6 cost is 3; path is 1 --> 6
1 --> 7 cost is 2; path is 1 --> 2 --> 5 --> 7
Knapsack of capacity 1 is worth 9 quatloos and contains 1 scoops of orange
Knapsack of capacity 6 is worth 38 quatloos and contains 2 scoops of orange 4 scoops of blue
Knapsack of capacity 10 is worth 58 quatloos and contains 2 scoops of orange 8 scoops of blue 0 scoops of green
Knapsack of capacity 20 is worth 74 quatloos and contains 2 scoops of orange 8 scoops of blue 6 scoops of green 4 scoops of red 0 scoops of red
Knapsack of capacity 21 is worth 74 quatloos and contains 2 scoops of orange 8 scoops of blue 6 scoops of green 4 scoops of red 1 scoops of red
PS C:\Users\Tom\Documents\GitHub\Algorithms>
```

Figure 8: Image of results

These are the results of the Bellman-Ford SSSP and the fractional knapsack problem.