0/1 Knapsack

Danielle Sarafian

16 March 2018

Abstract

This project focuses on and analyzes the strategies used to complete the 0/1 Knapsack problem including sorting by price density and taking the largest density first, sorting by price and taking the largest price first, using the brute force solution, and using the dynamic programming solution.

1 Motivation

The problem of finding the optimal solution is very important because the optimal solution is the best possible answer. Clearly, having the best answer can be useful in many cases. In some situations, it is important to consider the tradeoff between the optimal solution and the time it takes to find that optimal solution. If finding the optimal solution takes longer than finding a solution that is close to the optimal solution, is it worth it to spend the time finding the optimal solution?

2 Background

The 0/1 knapsack problem, as defined by Cormen in *Introduction to Algorithms* is as follows: "A thief robbing a store finds *n* items. The *i*th item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers. The thief wants to take as valuable a load as possible, but he can carry at most *W* pounds in his knapsack, for some integer *W*. Which items should he take?" [1]. This analysis could be applied to a real-world scenario when considering items to save during a fire or, like Cormen described, when a thief is taking as valueable a load as possible. Of course, in these situations, the highest price may be a lower priority than the time available, so, an analysis of different methods of taking these items is in order.

The goal of this project is to understand the pros and cons of different solutions to the 0/1 Knapsack problem. I predict that the price density solution will be the fastest and the brute force solution will take the longest time. In addition, I predict that the brute force will always find the optimal solution, but the other solutions will not always find it, though they may some of the time.

3 Procedure

A computer program was written in order to test this problem. This program consisted of different classes for each possible solution: brute force, greedy 1 (taking the item with the largest price first), greedy 2 (using price density), and the dynamic programming solution. For all solutions, the precondition exists that the weights of the items are not larger than the capacity. This is accounted for in the Knapsack class in which random weights and prices are generated. If the random weight exceeds the capacity, the number is regenerated. A class called Item was created in order to represent an item that has a weight and a price. When performing the two greedy solutions, the items needed to be sorted. Heap sort was used to sort these items because its time is $O(nlgn)$ and is the fastest sort, as proven in Project 1. The algorithms for the solutions are below.

Algorithm 1 GREEDYDENSITY()

```
1:  procedure GREEDYDENSITY
            ▷ All variables are instance variables initialized in the constructor for
    this class.
2:      for i = 0 to items.length do      ▷ items is an array containing objects of
    type Item
3:          temp = items[i].getPriceDensity()
4:          priceDensity[i] = temp
5:      end for
6:      finalKnapsack is a new array of size items.length
7:      priceDensity = sort.reverseHeapSort(priceDensity)
8:      for index = 0 to priceDensity.length do
9:          if items[index].getWeight() <= capacity then
10:             finalKnapsack[index] = 1
11:             c = −items[index].getWeight()
12:             capacityFilled = +items.getWeight()
13:             finalPrice = +items[index].getPrice()
14:         end if
15:     end for
16: end procedure
```

Algorithm 2 DYNAMICPROGRAMMING(profit, weight, c, array)

procedure DYNAMICPROGRAMMING(profit, weight, c, array)
    $numObj = profit.length$
    $yMax = min(weight[numObj - 1], c)$
    for $y = 0$ to $yMax$ **do**
        $array[numObj][y] = 0$
    end for
    for $y = weight[numObj$ to c **do**
        $array[numObj][y] = profit[numObj]$
    end for
    for $i = numObj - 1$ down to 0 **do**
        $yMax = min(weight[i] - 1, c)$
        for $y = 0$ to $yMax$ **do**
            $array[i][y] = array[i + 1][y]$
        end for
        for $y = weight[i]$ to c **do**
            $array[i][y] = max(array[i + 1][y], array[i + 1][y - weight[i]] + profit[i])$
        end for
    end for
    $array[1][c] = array[2][c]$
    if $c >= weight[1]$ **then**
        $array[1][c] = max(array[1][c], array[2][c - weight[1]] + profit[1])$
    end if
    solution is a new array of length profit.length
    traceback(array, weight, profit, c, solution)
                                                ▷ Algorithm from Sahni
end procedure

Algorithm 3 TRACEBACK(array, weight, price, c, solution)

```
1:  procedure TRACEBACK(array, weight, price, c, solution)
2:      numObj = weight.length − 1
3:      for i = 0 to numObj − 1 do
4:          if array[i][c] == array[i + 1][c] then
5:              solution[i] = 0;
6:          end if
7:          else
8:          solution[i] = 1
9:          c− = weight[i]
10:         end else
11:     end for
12:     solution[numObj] = (array[numObj][c] > 0)?1 : 0
                                                  ▷ Algorithm from Sahni
13: end procedure
```

Algorithm 4 BRUTEFORCE(num)

```
1:  procedure BRUTEFORCE(num)
2:      if num < 0 then
3:          weight = 0
4:          price = 0
5:          for i = 0 to numItems do
6:              if current[i] then
7:                  weight = weight + items[i].getWeight()
8:                  price = price + items[i].getPrice()
9:              end if
10:         end for
11:         if weight <= capacity and price > best then
12:             best = price
13:             capacityFilled = weight
14:             copySolution()
15:         end if
16:         return
17:     end if
18:     current[num] = true
19:     solve(num-1)
20:     current[num] = false
21:     solve(num-1)
                                          ▷ Based on code from GitHub, Inc.
22: end procedure
```

Algorithm 5 GREEDYPRICE()

```
1: procedure GREEDYPRICE
                ▷ PRECONDITION: price array is sorted lowest to highest value
2:     for i = sortedPrice.length − 1 down to 0 do
3:         if (currentC is nonnegative, currentC is less than c, and c-currentC is
   nonnegative) then
4:             solution[i] = 1
5:             currentC = currentC − (int)(sortedPrice[i].getWeight())
6:         end if
7:         else
8:         solution[i] = 0
9:         end else
10:    end for
11: end procedure
```

4 Testing

4.1 Testing Plan and Results

In order to test this, three trials were run for all of the data and for three of the solutions (price density, dynamic programming, and largest price). Brute force began to take too long after the test with 25 elements. The solution methods were run with capacities of 5, 15, 25, 35, 45, 55, 65, 75 , 85, and 95. The highest price and weight possible were set at 100, though if the random weight generated exceeded the capacity for the knapsack, it would be regenerated. The solutions were tested with an array of 10 random items.

BRUTE FORCE				
# OF ELEMENTS	TRIAL 1	TRIAL 2	TRIAL 3	AVERAGE
5	3892146	3278403	4919792	4030113.667
10	2467007	1455042	2085924	2002657.667
15	11741350	10214832	11456176	11137452.67
20	122044833	105454073	160820425	129439777
25	2858550232	2963866094	2981978995	2934798440

Figure 1: Results from Brute Force Solution

DYNAMIC PROGRAMMING				
# OF ELEMENTS	TRIAL 1	TRIAL 2	TRIAL 3	AVERAGE
5	995555	2122027	753777	1290453
15	330758	917150	565242	604383.3333
25	247612	886153	368319	500694.6667
35	759247	670996	173584	534609
45	156444	100649	678290	311794.3333
55	573994	513094	382906	489998
65	195828	564512	354826	371722
75	130188	610097	227191	322492
85	731532	466780	321276	506529.3333
95	836923	146233	254177	412444.3333

Figure 2: Results from Dynamic Programming Solution

GREEDY SOLUTION #1: LARGEST PRICE				
# OF ELEMENTS	TRIAL 1	TRIAL 2	TRIAL 3	AVERAGE
5	1017800	964558	854062	945473.3333
15	768729	1365333	479179	871080.3333
25	413174	945230	545185	634529.6667
35	868649	4796533	306689	1990623.667
45	160820	136752	638906	312159.3333
55	644011	526586	395304	521967
65	711475	727521	247248	562081.3333
75	111954	516011	301219	309728
85	789879	629059	441618	620185.3333
95	1871498	80593	249071	733720.6667

Figure 3: Results from Greedy 1: Largest Price Solution

GREEDY SOLUTION #2: PRICE DENSITY				
# OF ELEMENTS	TRIAL 1	TRIAL 2	TRIAL 3	AVERAGE
5	1698278	1706301	1470358	1624979
15	568159	563418	313254	481610.3333
25	603532	1019623	428125	683760
35	749402	895635	256000	633679
45	428855	253447	233755	305352.3333
55	195099	497048	371601	354582.6667
65	762530	732262	433595	642795.6667
75	127636	820512	250165	399437.6667
85	663703	543727	494496	567308.6667
95	576547	336957	380717	431407

Figure 4: Results from Greedy 2: Price Density Solution

4.2 Problems Encountered

An issue that was encountered with this experiment was with the use of Sahni's algorithm for the dynamic programming solution. He refers to his arrays as being 1-relative, so, it caused errors when his code was copied exactly as it was in the PDF scan provided. In addition, Sahni's code will not work if the first item checked has a weight larger than the capacity. Finally, it seems that Sahni's code does not always return the optimal value when compared to that of brute force. This can be seen because the results yielded by Sahni's code often take up a smaller capacity than that of brute force.

5 Experimental Analysis

When comparing these four strategies, the dynamic programming method seems to be the fastest with a total average time of 534512 nanoseconds. The second fastest seems to be Greedy 2 with a total average time of about 612491 nanoseconds. The third fastest seems to be Greedy 1 with a total average time of about 750154 nanoseconds. Finally, the slowest solution method is the brute force with a total average time of 616281688 nanoseconds. While the dynamic programming method was the fastest, it did not seem to return the optimal solution as given by the brute force method. I believe that the results for the dynamic programming solution may be off because, based on further reading and research, it should yield the optimal solution. The graph for the dynamic programming average times, shown in Figure 6 shows the total time decreasing as the capacity increases. This may be because, with this type of solution, values do not have to be recalculated, as they do in brute force. Instead, they are saved in a 2-dimensional array which represents a table of values and can be referred to when other values are being calculated. According to Cormen in *Introduction to Algorithms*, the price density algorithm should take $O(nlgn)$ time to run, as seen in the graph in Figure 10. The dynamic programming method should take about $O(nm)$ where m is the number of items in the array of possible items. In this case, $m = 10$ and $O(n * 10)$ can be simplified to $O(n)$ as seen in the graph in Figure 9. In the brute force solution, the for loop runs approximately 2^n times, giving it a larger time complexity than any of the other solutions, as shown by the fewer results able to be calculated from that solution.

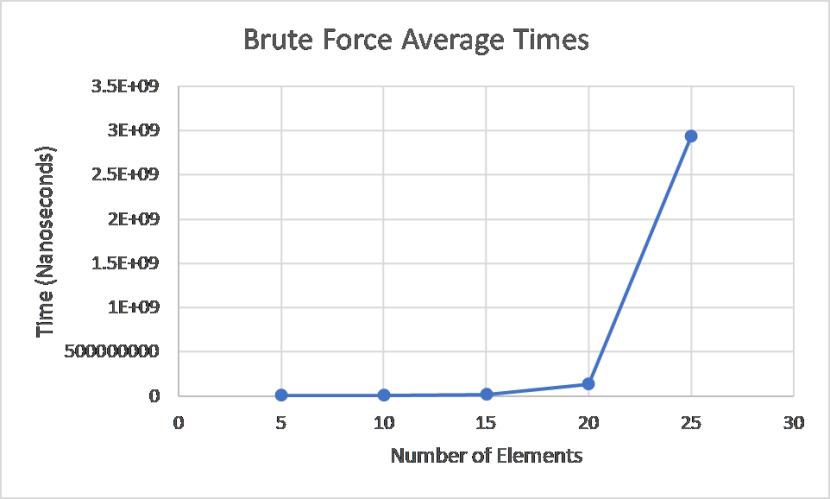

Figure 5: Graph for Brute Force Solution

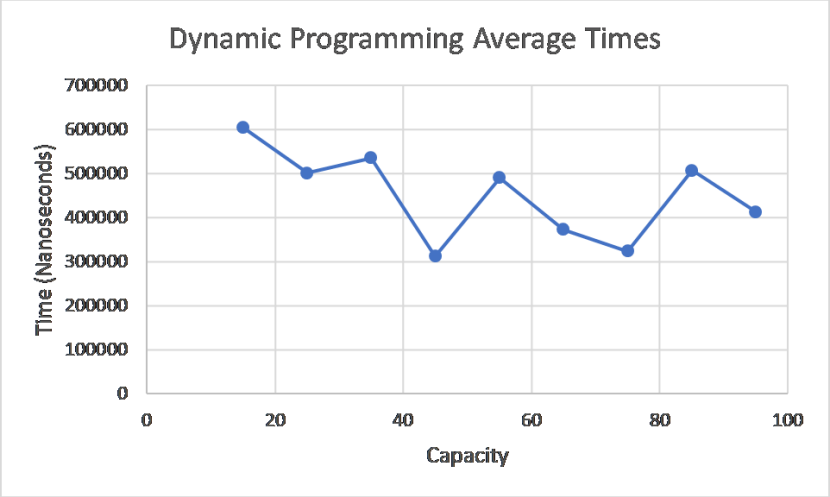

Figure 6: Graph for Dynamic Programming Solution

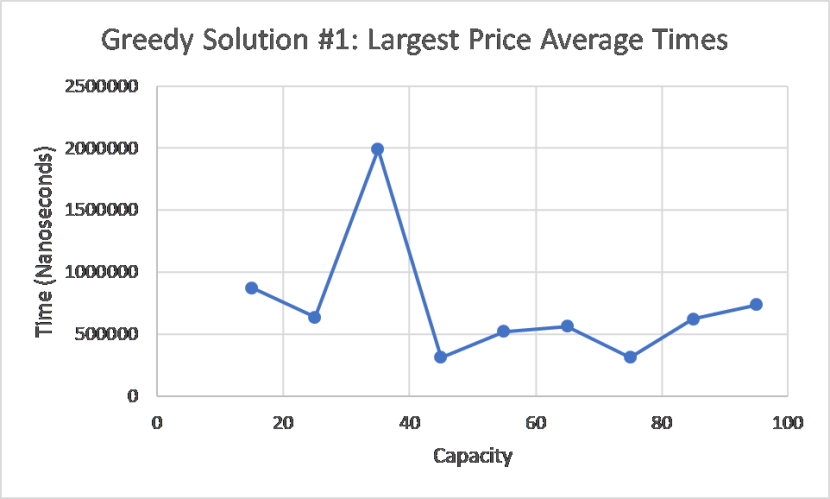

Figure 7: Graph for Greedy 1: Largest Price Solution

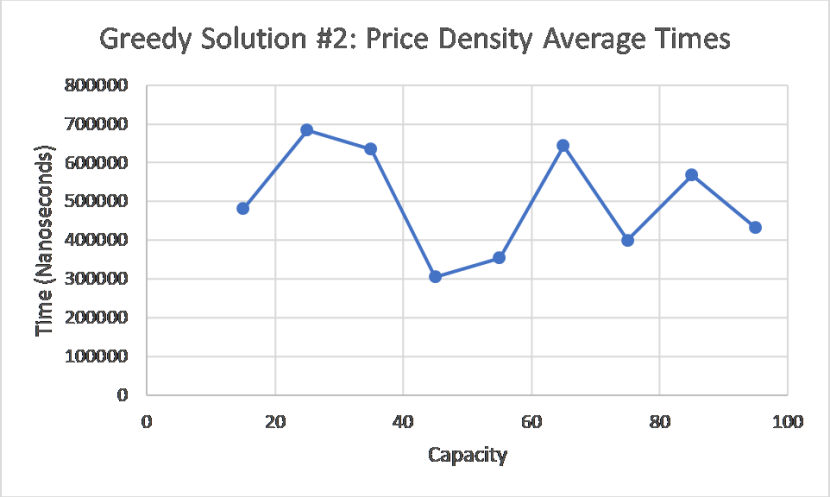

Figure 8: Graph for Greedy 2: Price Density Solution

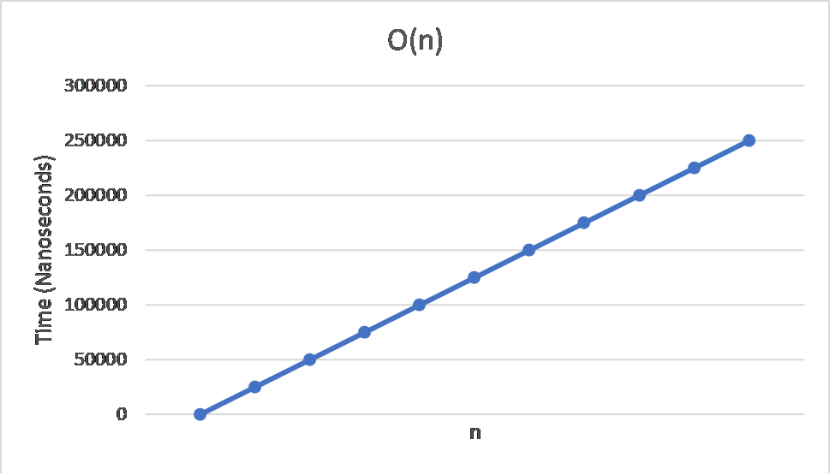

Figure 9: Graph of $O(n)$

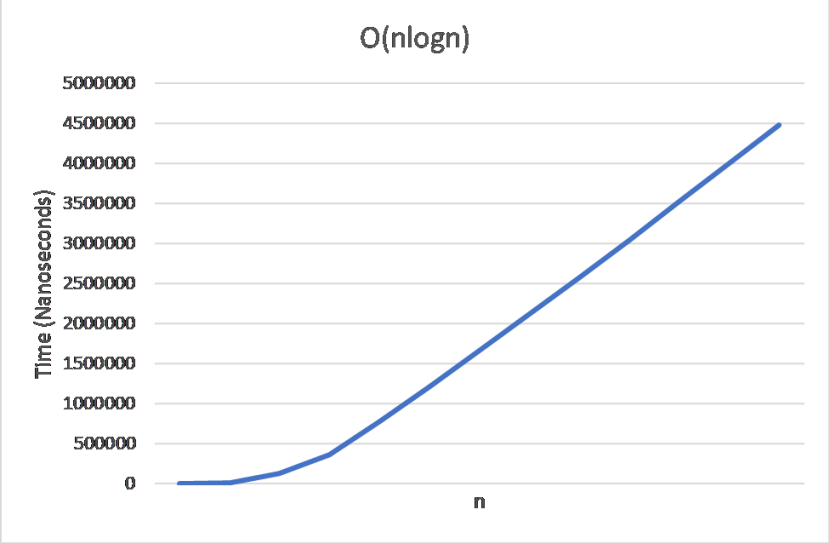

Figure 10: Graph of $O(nlogn)$

6 Conclusions

If the knapsack capacity is smaller, it may be feasible to use the brute force solution, but, as the capacity grows larger, so does the time it takes for this solution

to finish. The best solution time-wise and accuracy-wise is the dynamic programming solution because, even though, from these results, it does not always provide the optimal solution, the total time appears to decrease as the capacity of the knapsack increases and the solutions seem to be most similar to those of the brute force method. Another testing method that may be helpful in order to yield more results is keeping the capacity constant and changing the number of possible items to see how long it takes to fit more and more items into a specific capacity with each solution method. It is also important to consider the positive aspects of greedy algorithms and the fact that they are more likely to be used in real-world situations, in which a "good enough" solution could be acceptable, than the more tedious solutions, like brute force and dynamic programming, which would give more exact solutions.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein., *Introduction to Algorithms, 3rd Edition*. MIT press Cambridge, 2001.

Appendix A

JAVA Source Code: **BruteForce.java**

```
/**
 *  Programming Project 2: 0/1 Knapsack<br>
 *
 *  This class represents a brute force solution to the 0/1 knapsack.
 *
 *  <br> <br>
 *  Created: <br>
 *     8 March 2018, Danielle Sarafian<br>
 *  Modifications: <br>
 *     12 March 2018, Danielle Sarafian, greedyDensity<br>
 *     14 March 2018, Danielle Sarafian, restructure to use Items
 *
array in constructor rather
 *
than 2 arrays for price and weight<br>
 *
 *  @author Danielle Sarafian with assistance from Nora Wichmann and GitHub, Inc.
 */

public class BruteForce {
        private Item items[];
        private int numItems;
        private int capacity;
        private double best;
        private boolean solution[];
        private boolean current[];
        private double capacityFilled;


        /**
         * Constructs an object of this class
         *              @param  allItems        an array of Item objects to find the
         *              @param  c                       the capacity
         */
        public BruteForce(Item[] allItems, int c)
        {
                System.out.println("Brute Force Solution");

                // number of items
                numItems = allItems.length;

                // capacity of knapsack
```

```
        capacity = c;

        capacityFilled = 0;

        items = allItems;

        // populate items array
        /*System.out.println("Item Weight Value");
        for (int i = 0; i < numItems; i++)
        {
                System.out.println(i + "\t" + items[i].getWeight() + "\t" +
        }*/
        best = Integer.MIN_VALUE;
        solution = new boolean[numItems];
        current = new boolean[numItems];

        solve(numItems-1);
        printSolution();
}

/**
 * Solves this solution to the knapsack problem
 *              @param num      number of items to solve for
 */
private void solve(int num)
{
        // check that there are items to solve for
        if (num < 0)
        {
                double weight = 0;
                double price = 0;
                double prevWeight = 0;
                double prevPrice = 0;

                // go through all items
                // INVARIANT: if the item is selected, weight-prevWeight = v
                // vacuously true
                for (int i = 0; i < numItems; i++)
                {
                        // check if the current item is being taken
                        if (current[i])
                        {
                                // increase weight and price counters
                                prevWeight = weight;
                                weight = weight + items[i].getWeight();
```

```
                                prevPrice = price;
                                price = price + items[i].getPrice();
                        }
                        //assert(solveInvariant(prevWeight, prevPrice, weigh
                }
                //assert(solveInvariant(prevWeight, prevPrice, weight, price


                // check that the item will fit in the knapsack and that the
                // than the current best price
                if ((weight <= capacity) && (price > best))
                {
                        // update info
                        best = price;
                        capacityFilled = weight;

                        // copy current best solution
                        copySolution();
                }
                return;
        }

        // check when this item is true
        current[num] = true;
        solve(num-1);

        // check when this item is false
        current[num] = false;
        solve(num-1);
}

/**
 * Copies the current best solution into the solution array
 */
private void copySolution()
{
        for (int i = 0; i<numItems; i++)
        {
                solution[i] = current[i];
        }
}


/**
 * Prints the solution
```

```
         */
        private void printSolution()
        {
                System.out.println(" Item Weight Value");
                for (int i = 0; i < numItems; i++)
                {
                        if (solution[i])
                        {
                                System.out.println(i + "\t" + (int) items[i].getWeig
                        }
                }
                System.out.println("Total Price: " + best);
                System.out.println("Capacity Used: " + capacityFilled);
                System.out.println("");
        }

        private boolean solveInvariant(double prevWeight, double prevPrice, double v
        {
                if (current[index])
                {
                        if ((items[index].getWeight() != weight-prevWeight) && (item
                        {
                                return false;
                        }

                }
                return true;
        }
}
```

JAVA Source Code: **DP2.java**

```
/***
 *
 *  Programming Project 2: 0/1 Knapsack<br>
 *
 *  This class represents dynamic programming solution based on
 *  Sahni's code.
 *
 *  <br> <br>
 *  Created: <br>
 *     13 March 2018, Danielle Sarafian<br>
 *  Modifications: <br>
 *     14 March 2018, Danielle Sarafian, fix off-by-one error<br>
 *
 *  @author Danielle Sarafian, with assistance from Nora Wichmann
 */
```

```
public class DP2
{

        public DP2()
        {
                System.out.println("Dynamic Programming");
        }


        // Methods
        /**
         *  This method finds the 2D array, array, to solve the knapsack problem ite
         *    @param     profit array of object values
         *    @param     weight array of object weights
         *    @param     c capacity of  knapsack
         *    @param     array 2D array of profits and weights of eligible items
         **/
        public void solve(int[] profit, int[]weight, int c, int[][] array)
        {
                int numObj = profit.length-1;

                //initialize array[numObj][]
                int yMax = Math.min(weight[numObj]-1, c);

                for(int y = 0; y <= yMax; y++)
                {
                        array[numObj][y]=0;
                }

                int y1 = weight[numObj];
                for(; y1 <= c; y1++)
                {
                        array[numObj][y1]=profit[numObj];
                        //assert(invariantEquality(array, profit, y1, numObj));
                }
                //assert(invariantEquality(array, profit, y1, numObj));


                //compute array[i][y], 1<i<numObj
                for(int i = numObj-1; i > 0; i--)
                {
                        yMax = Math.min(weight[i]-1, c);
                        for(int y = 0; y <= yMax; y++)
                        {
                                array[i][y] = array[i+1][y];
```

```
                }
                for(int y = weight[i]; y <= c; y++)
                {
                        array[i][y] = Math.max(array[i+1][y], array[i+1][y-v
                }
                //assert(yMaxInvariant(yMax, c));
        }
        //assert(yMaxInvariant(yMax, c));


        array[1][c] = array[2][c];
        if(c >= weight[1])
        {
                array[1][c] = Math.max(array[1][c], array[2][c-weight[1]]+pr
        }

        int[] solution = new int[profit.length];
        traceback(array, weight, profit, c, solution);
}

/**
 * Traces the table back
 * @param array         2d array to represent table
 * @param weight        array of weights
 * @param c                     capacity
 * @param solution      array to hold solution
 */
public void traceback(int[][] array, int[] weight, int[] price, int c, int[]
{
        int numObj = weight.length-1;
        for(int i = 0; i < numObj-1; i++)
        {
                if(array[i][c] == array[i+1][c])
                {
                        //don't include i
                        solution[i] = 0;
                }
                else
                {
                        //include i
                        solution[i] = 1;
                        c -= weight[i];
                }
        }
        solution[numObj] = (array[numObj][c]>0) ? 1 : 0;
```

```
        printSolution(solution, weight, price);
    }

    private void printSolution(int[] s, int[] weight, int[] price)
    {
        int currentPrice = 0;
        int currentWeight = 0;
        System.out.println("Item Weight Value");
        for (int i = 0; i < s.length; i++)
        {
            if (s[i] == 1)
            {
                System.out.println(i + "\t" + weight[i] + "\t" + pri
                currentPrice = currentPrice + price[i];
                currentWeight = currentWeight + weight[i];
            }
        }
        System.out.println("Total Price: " + currentPrice);
        System.out.println("Capacity Used: " + currentWeight);
    }

    /**
     * Checks the invariant that the last row of the table should equal
     * the items array.
     *
     * @param array2d       the 2d array
     * @param iArray        the array of items
     * @param index         the current index
     * @param num           the number of objects
     * @return      false if the invariant fails, otherwise true
     */
    private boolean invariantEquality(int[][] array2d, int[] profit, int index,
    {
        if (array2d[num][index] != (int)profit[num])
        {
            return false;
        }
        return true;
    }

    /**
     * Checks the invariant yMax <= c
     * @param yMax          the yMax value
     * @param c                     the capacity
     * @return      false if the invariant fails, otherwise true
```

```
         */
        private boolean yMaxInvariant(int yMax, int c)
        {
                if (yMax > c)
                {
                        return false;
                }
                return true;
        }
}
```

JAVA Source Code: **GreedyDensity.java**

```
// import java.awt.Color;
// import java.util.ArrayList;
// import java.util.Random;
/**
 *  Programming Project 2: 0/1 Knapsack<br>
 *
 *  This class represents the greedy solution using
 *  price density for a 0/1 Knapsack.
 *
 *  <br> <br>
 *  Created: <br>
 *     8 March 2018, Danielle Sarafian<br>
 *  Modifications: <br>
 *     12 March 2018, Danielle Sarafian, solve<br>
 *     13 March 2018, Danielle Sarafian, printSolution<br>
 *     14 March 2018, Danielle Sarafian, make methods work for arrays of Item object
 *
 *  @author Danielle Sarafian with assistance from Nora Wichmann and GitHub, Inc.
 */
public class GreedyDensity
{
        // State: instance variables and shared class variables go here.
        private int c;
        private Sort sort;
        private double[] finalKnapsack;
        private double[] priceDensity;
        private Item[] items;
        private int capacityFilled;
        private int finalPrice;

        // Constructors


        /**
```

```
 * Constructs a new object of this class.
 *      @param   paramArray    array of Item to find the solution for
 *      @param   capacity          the capacity of the knapsack
 */
public GreedyDensity(Item[] paramArray, int capacity)
{
        System.out.println("Greedy Solution: Price Density Array");

        c = capacity;
        sort = new Sort();
        priceDensity = new double[paramArray.length];
        items = paramArray;
        capacityFilled = 0;
        finalPrice = 0;
        solve();
}

// Methods

/**
 * Greedy algorithm using price density array
 *
 * @return array of solutions 1 if the item was taken, 0 if it wasn't
 */
public void solve()
{
        // calculate price densities and fill array
        //System.out.println("Original Array");
        //System.out.println("Item Weight Value");
        for (int i = 0; i < items.length; i++)
        {
                double temp = items[i].getPriceDensity();
                priceDensity[i] = temp;
                //System.out.println(i + "\t" + items[i].getWeight() + "\t"
        }

        // create array to represent final knapsack
        finalKnapsack = new double[items.length];

        // sort the price densities
        priceDensity = sort.reverseHeapSort(priceDensity);

        //System.out.println("Sorted by Price Density");
        //System.out.println("Price Densities");
        /*for (int i = 0; i < priceDensity.length; i++)
```

```
        {
                System.out.println(priceDensity[i]);
        }*/

        // check if each item fits in the knapsack
        // INVARIANT: the space in the knapsack is never < 0
        //assert(spaceInvariant(c));
        for (int index = 0; index < priceDensity.length; index++)
        {
                if (items[index].getWeight() <= c)
                {
                        finalKnapsack[index] = 1;
                        c = c - (int) items[index].getWeight();
                        capacityFilled = capacityFilled + (int)items[index].
                        finalPrice = finalPrice + (int)items[index].getPrice
                }
                //assert(spaceInvariant(c));
        }
        //assert(spaceInvariant(c));

        printSolution();
}

/**
 * Prints the solution to this problem
 */
private void printSolution()
{
        System.out.println("  Item  Weight  Value");
        for (int i = 0; i < items.length; i++)
        {
                if (finalKnapsack[i] == 1.0)
                {
                        System.out.println(i + "\t" + items[i].getWeight() +
                }

        }
        System.out.println("Total Price: " + finalPrice);
        System.out.println("Capacity Used: " + capacityFilled);
        System.out.println("");
}

/**
 * checks that c >=0
 * @param c     capacity
```

```
         * @return false if invariant fails, otherwise true
         */
        private boolean spaceInvariant(int c)
        {
                if (c<0)
                {
                        return false;
                }
                return true;
        }
}
```

JAVA Source Code: **GreedyPrice.java**

```
/**
 *  Project 2: 0/1 Knapsack<br>
 *
 *  The <code>GreedyPrice</code> class provides a GreedyPrice object
 *  which will find the solution for the knapsack when taking the largest price firs
 *
 *  <br> <br>
 *  Created: <br>
 *     14 March 2018, Danielle Sarafian<br>
 *
 *  @author Danielle Sarafian
 */
public class GreedyPrice {
        Item[] items;
        double[] solution;
        int c;
        int currentC;
        int totalCUsed;
        int currentP;
        Sort sort;
        Item[] sortedPrice;

        /**
         * Creates a new object of this class
         * @param allItems      array of Items to find the best knapsack for
         * @param capacity      capacity of knapsack
         */
        public GreedyPrice(Item[] allItems, int capacity)
        {
                System.out.println("Greedy Solution: Largest Price First");

                c = capacity;
                currentC = c;
```

```
        currentP = 0;
        totalCUsed = 0;
        items = allItems;

        solution = new double[allItems.length];
        sort = new Sort();
        sortedPrice = sort.heapSort(items);

        /*System.out.println("Item  Weight  Value");
        for (int i = 0; i < sortedPrice.length; i++)
        {
                System.out.println(i + "\t" + sortedPrice[i].getWeight() + "
        }*/

        solve();
}

/**
 * Solves this knapsack problem
 */
public void solve()
{
        // go through list of sorted prices from the highest price first
        // INVARIANT: the space in the knapsack is never < 0
        //assert(spaceInvariant(currentC));
        for (int i = sortedPrice.length-1; i >= 0; i--)
        {
                // check if the current capacity is greater than 0 and less
                // check that the item will fit in the knapsack
                if (currentC >= 0 && currentC <= c && (currentC - (int)(sort
                {
                        // take the item
                        solution[i] = 1;

                        // update price and capacity counters
                        currentC = currentC - (int)(sortedPrice[i].getWeight
                        currentP = currentP + (int)(sortedPrice[i].getPrice(
                        totalCUsed = totalCUsed + (int)(sortedPrice[i].getW
                }
                else
                {
                        // don't take the item
                        solution[i] = 0;
                }
                //assert(spaceInvariant(currentC));
```

```
                }
                //assert(spaceInvariant(currentC));
                printSolution();

        }

        /**
         * Prints this knapsack solution.
         */
        private void printSolution()
        {
                System.out.println("Item Weight Value");
                for (int i = 0; i < sortedPrice.length; i++)
                {
                        if (solution[i] == 1)
                        {
                                System.out.println(i + "\t" + sortedPrice[i].getWeig
                        }
                }
                System.out.println("Total Price: " + currentP);
                System.out.println("Capacity Used: " + totalCUsed);
                System.out.println("");
        }

        /**
         * checks that c >=0
         * @param c     capacity
         * @return false if invariant fails, otherwise true
         */
        private boolean spaceInvariant(int c)
        {
                if (c<0)
                {
                        return false;
                }
                return true;
        }
}
```

JAVA Source Code: **Item.java**

```
/**
 *  Project 2: 0/1 Knapsack<br>
 *
 *  The <code>Item</code> class represents an item that can be put in a knapsack
 *
 *  <br> <br>
```

```
 *  Created: <br>
 *     5 March 2018, Danielle Sarafian<br>
 *
 *  Modifications: <br>
 *     14 March 2018, Danielle Sarafian, add toString method<br>
 *
 *  @author Danielle Sarafian
 */
public class Item {
        double weight;
        double price;

        /**
         * Constructs a new item of this class
         * @param w     the weight of the item
         * @param p     the price of the item
         */
        public Item(double w, double p)
        {
                weight = w;
                price = p;
        }

        /**
         *
         * @return      the weight of this item
         */
        public double getWeight()
        {
                return weight;
        }

        /**
         *
         * @return      the price of this item
         */
        public double getPrice()
        {
                return price;
        }

        /**
         *
         * @return      the price density of this item
         */
```

```
        public double getPriceDensity()
        {
                return (price/weight);
        }

        /**
         * @return      this item as a string
         */
        public String toString()
        {
                return (weight + "\t" + price);
        }
}
```

JAVA Source Code: **Knapsack.java**

```
// Import statements go here.  For example,
// import java.awt.Color;
// import java.util.ArrayList;
// import java.util.Random;
import java.util.Arrays;
import java.util.Random;

/**
 *  Programming Project 2: 0/1 Knapsack<br>
 *
 *  This class represents a 0/1 knapsack.
 *
 *  <br> <br>
 *  Created: <br>
 *     8 March 2018, Danielle Sarafian<br>
 *  Modifications: <br>
 *     12 March 2018, Danielle Sarafian, randomValues<br>
 *
 *  @author Danielle Sarafian
 */
public class Knapsack
{
        // State: instance variables and shared class variables go here.
        int n;
        Item items[];
        int maxP;
        int maxW;
        int c;

        // Constructors
```

```
        /**
         * Constructs a new object of this class.
         *      @param    size                size of array
         *      @param    highestPrice    the highest price possible to generate
         *      @param    highestWeight   the highest weight possible to generate
         *      @param    capacity                the capacity of the knapsack
         */
        public Knapsack (int size, int highestPrice, int highestWeight, int capacity
        {
                n = size;
                items = new Item[size];
                maxP = highestPrice;
                maxW = highestWeight;
                c = capacity;
        }

        // Methods

        /**
         * Generate random price and weight values
         *
         * @return array of Item with random price and weight values
         */
        public Item[] randomValues()
        {
                Random generator = new Random();
                int randPrice;
                int randWeight;
                int i = 0;

                while (i <n)
                {
                        randPrice = generator.nextInt(maxP) + 1;
                        randWeight = generator.nextInt(maxW) + 1;
                        if (randWeight <= c)
                        {
                                items[i] = new Item(randWeight, randPrice);
                                i++;
                        }
                }
                return items;
        }

}
```

JAVA Source Code: **Simulation.java**

```
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;

/**
 *  Project 2: 0/1 Knapsack<br>
 *
 *  The <code>Simulation</code> class provides a main method
 *  for a program that determines the best items to put in a 0/1 knapsack
 *  in order to have the highest cost.
 *
 *  <br> <br>
 *  Created: <br>
 *     5 March 2018, Danielle Sarafian<br>
 *
 *
 *  @author Danielle Sarafian
 */
public class Simulation<T>
{
        /**
         *  The main function initiates execution of this program.
         *    @param    String[] args not used in this program
         **/
        public static void main(String[] args) throws IOException
        {
                System.out.println ("Welcome to Project 2.");

                //int capacity = 175;
                //Knapsack knapsack = new Knapsack(10, 10, 10);
                //knapsack = new Knapsack(3, 4, 4);
                //Item[] items = knapsack.randomValues();
                //DynamicProgramming dp = new DynamicProgramming(items, 40);
                //BruteForce bf = new BruteForce(items, 40);
                //GreedyDensity gd = new GreedyDensity(items, 5);
                //GreedyPrice gp = new GreedyPrice(items, 5);

                // create list of different lengths to test
                ArrayList<Integer> capacities = new ArrayList<Integer>();

                for (int i = 5; i <30; i = i+10)
                {
                        capacities.add(i);
                }
```

```
Knapsack knapsack;
int highestPrice = 100;
int highestWeight = 100;

PrintWriter out = new PrintWriter(new FileWriter("C:\\Users\\owner\\

// create list for the times it takes to sort different sizes
ArrayList<Long> bfTimes = new ArrayList<Long>();
ArrayList<Long> gdTimes = new ArrayList<Long>();
ArrayList<Long> gpTimes = new ArrayList<Long>();
ArrayList<Long> dpTimes = new ArrayList<Long>();


for (Integer i : capacities)
{
        System.out.println("capacity: " + i);
        System.out.println("");
        knapsack = new Knapsack(10, highestPrice, highestWeight, i);

        Item[] items = knapsack.randomValues();
        int profits[] = new int[items.length];
        int weights[] = new int[items.length];


        for (int j = 0; j < items.length; j++)
        {
                profits[j] = (int)items[j].getPrice();
                weights[j] = (int)items[j].getWeight();
        }
        int array[][] = new int[profits.length+1][i+1];

        // time for brute force
        if (i <=25)
        {
                long bfStartTime = System.nanoTime();
                BruteForce bf = new BruteForce(items, i);
                long bfEndTime = System.nanoTime();
                long bfDuration = bfEndTime - bfStartTime;
                bfTimes.add(bfDuration);
                System.out.println("brute force time: " + bfDuration
        }
        else
        {
                bfTimes.add(Long.MAX_VALUE);
```

```
            }

            // time for greedy solution with price density
            /*long gdStartTime = System.nanoTime();
            GreedyDensity greedyDensity = new GreedyDensity(items, i);
            long gdEndTime = System.nanoTime();
            long gdDuration = gdEndTime-gdStartTime;
            gdTimes.add(gdDuration);
            System.out.println("greedy price density time: " + gdDuratio

            // time for greedy solution based on price
            long gpStartTime = System.nanoTime();
            GreedyPrice gp = new GreedyPrice(items, i);
            long gpEndTime = System.nanoTime();
            long gpDuration = gpEndTime-gpStartTime;
            gpTimes.add(gpDuration);
            System.out.println("greedy price time: " + gpDuration);

            // time for dynamic programming
            long dpStartTime = System.nanoTime();
            DP2 dp = new DP2();
            dp.solve(profits, weights, i, array);
            long dpEndTime = System.nanoTime();
            long dpDuration = dpEndTime-dpStartTime;
            dpTimes.add(dpDuration);
            System.out.println("dynamic programming time: " + dpDuration
    }

    // print results into a txt file with tab deliminators
    out.println("SOLUTION TYPE \t NUM ELEMENTS \t TIME");
    for (int i = 0; i < bfTimes.size(); i++)
    {
            out.println("BRUTE FORCE \t" + capacities.get(i) + "\t" + bf
    }
    /*for (int i = 0; i < gdTimes.size(); i++)
    {
            out.println("GREEDY PRICE DENSITY \t" + capacities.get(i) +
    }
    for (int i = 0; i < gpTimes.size(); i++)
    {
            out.println("GREEDY PRICE \t" + capacities.get(i) + "\t" + g
    }
    for (int i = 0; i < dpTimes.size(); i++)
    {
            out.println("DYNAMIC PROGRAMMING \t" + capacities.get(i) + "
```

```
                }
                out.close();*/



                System.out.println ("Program done.");

        }//end main
}//end class
```

JAVA Source Code: **Sort.java**

```
import java.util.ArrayList;

/**
 *  Project 2: 0/1 Knapsack<br>
 *
 *  Sorts an ArrayList using heap sort
 *
 *  <br> <br>
 *  Created: <br>
 *     8 March 2018, Danielle Sarafian<br>
 *
 *  Modifications: <br>
 *     8 March 2018, Danielle Sarafian, switch methods from using ArrayLists to usin
 *
 *  @author Danielle Sarafian
 */
public class Sort
{
        // State: instance variables and shared class variables go here.

        // Constructors

        /**
         * Constructs a new object of this class.
         */
        public Sort()
        {
        }

        // Methods
        /**
         * This method will find the left child of a given node in the max heap
         *
         *              @param  index   the index of the element in the heap to find
```

```
     *              @return the index in the ArrayList where the left child node
     */
    private int left(int index)
    {
            return 2*index;
    }

    /**
     * This method will find the right child of a given node in the max heap
     *
     *              @param  index   the index of the element in the heap to find
     *              @return the index in the ArrayList where the right child nod
     */
    private int right(int index)
    {
            return ((2*index)+1);
    }

    /**
     * This method will swap the items given as parameters
     *
     *              @param  list    the ArrayList that contains the values that
     *              @param  index1  the index of one of the values that is being
     *              @param  index2  the index of the other value that is being s
     *              @return the ArrayList with the swapped values
     */
    private double[] swap (double[] list, int index1, int index2)
    {
            // set item at index1 to a temp
            double temp = list[index1];

            // set item at index2 in index1
            list[index1] = list[index2];

            // set temp at index2
            list[index2] = temp;

            return list;
    }

    /**
     * This method will make sure that the heap is a max heap and if it isn't,
     * the method will rearrange the terms to make it a max heap
     *
     *              @param  list            ArrayList to sort
```

```
     *              @param  index           index to make max heap from
     *              @param  endIndex        the last index to use in the heap
     *              @return an ArrayList that represents a max heap
     */
    private double[] minHeapify(double[] list, int index, int endIndex)
    {
            // get indices of left and right children
            int left = left(index);
            int right = right(index);

            // set largest to index
            int smallest = index;

            // check if largest is smaller than the left child

            if (left <= endIndex && (list[smallest] > list[left]))
            {
                    // set largest equal to the left child
                    smallest = left;
            }

            // check if largest is smaller than the right child
            if ((right <= endIndex) && (list[smallest] > list[right]))
            {
                    // set largest equal to the right child
                    smallest = right;
            }

            // if the item at index isn't the largest
            if (smallest != index)
            {
                    // swap the item at index with the largest item
                    list = swap(list, index, smallest);

                    // recursively check that the next heap is a max heap
                    list = minHeapify(list, smallest, endIndex);
            }
            return list;
    }

    /**
     * Makes the ArrayList a max heap
     *
     *              @param  list            the ArrayList to make a max heap
     *              @param  endIndex        the last index to use in the heap
```

```
     *              @return the ArrayList as a max heap
     */
    private double[] buildMinHeap(double[] list, int endIndex)
    {
            int heapSize = list.length;

            // vacuously true
            //assert(heapAssert(list, heapSize));

            // make sure the ArrayList represents a max heap
            while (heapSize > 0)
            {
                    //assert(heapAssert(list, heapSize));
                    minHeapify(list, heapSize-1, endIndex);
                    heapSize--;
            }

            // check invariant before returning list
            //assert(heapAssert(list, heapSize));
            return list;
    }

    /**
     * Takes an unsorted list and sorts it
     *
     *              @param  list    the ArrayList to sort
     *              @return the sorted ArrayList
     */
    public double[] reverseHeapSort(double[] list)
    {
            // set variable for list size
            int size = list.length;

            // set variable to count index
            int i = size-1;

            // create a max heap with the list
            buildMinHeap(list, i);

            while (i > 0)
            {
                    // switch largest number (located at beginning index)
                    // and smallest number (located at ending index)
                    list = swap(list, 0, i);
```

```
                // remove A[n]
                i--;

                // reconfigure into another max heap
                list = minHeapify(list, 0, i);
        }
        return list;
}
////////////////////////////////////////////
/**
 * This method will make sure that the heap is a max heap and if it isn't,
 * the method will rearrange the terms to make it a max heap
 *
 *              @param  list            ArrayList to sort
 *              @param  index           index to make max heap from
 *              @param  endIndex        the last index to use in the heap
 *              @return an ArrayList that represents a max heap
 */
private Item[] maxHeapify(Item[] list, int index, int endIndex)
{
        // get indices of left and right children
        int left = left(index);
        int right = right(index);

        // set largest to index
        int largest = index;

        // check if largest is smaller than the left child

        if (left <= endIndex && (list[largest].getPrice() < list[left].getPr
        {
                // set largest equal to the left child
                largest = left;
        }

        // check if largest is smaller than the right child
        if ((right <= endIndex) && (list[largest].getPrice() < list[right].g
        {
                // set largest equal to the right child
                largest = right;
        }

        // if the item at index isn't the largest
        if (largest != index)
        {
```

```
                // swap the item at index with the largest item
                list = swapItem(list, index, largest);

                // recursively check that the next heap is a max heap
                list = maxHeapify(list, largest, endIndex);
        }
        return list;
}

/**
 * Makes the ArrayList a max heap
 *
 *              @param  list            the ArrayList to make a max heap
 *              @param  endIndex        the last index to use in the heap
 *              @return the ArrayList as a max heap
 */
private Item[] buildMaxHeap(Item[] list, int endIndex)
{
        int heapSize = list.length;

        // vacuously true
        //assert(heapAssert(list, heapSize));

        // make sure the ArrayList represents a max heap
        while (heapSize > 0)
        {
                //assert(heapAssert(list, heapSize));
                maxHeapify(list, heapSize-1, endIndex);
                heapSize--;

        }

        // check invariant before returning list
        //assert(heapAssert(list, heapSize));
        return list;
}

/**
 * Takes an unsorted list and sorts it
 *
 *              @param  list    the ArrayList to sort
 *              @return the sorted ArrayList
 */
public Item[] heapSort(Item[] list)
{
```

```
        // set variable for list size
        int size = list.length;

        // set variable to count index
        int i = size-1;

        // create a max heap with the list
        buildMaxHeap(list, i);

        while (i > 0)
        {
                // switch largest number (located at beginning index)
                // and smallest number (located at ending index)
                list = swapItem(list, 0, i);

                // remove A[n]
                i--;

                // reconfigure into another max heap
                list = maxHeapify(list, 0, i);
        }
        return list;
}
/**
 * This method will swap the items given as parameters
 *
 *              @param  list    the ArrayList that contains the values that
 *              @param  index1  the index of one of the values that is being
 *              @param  index2  the index of the other value that is being s
 *              @return the ArrayList with the swapped values
 */
private Item[] swapItem (Item[] list, int index1, int index2)
{
        // set item at index1 to a temp
        Item temp = list[index1];

        // set item at index2 in index1
        list[index1] = list[index2];

        // set temp at index2
        list[index2] = temp;

        return list;
}
```

```
}
```