

Programming Assignment 3: 0/1 Knapsack

Melany Diaz

February 16, 2017

Abstract

This report will be validating and analyzing the time complexity of three different strategies used for 0/1 Knapsack. We hope to compare how close each strategy gets to the most optimal solution and each strategy's execution time. The three strategies used are brute force, dynamic programming, and greedy.

MOTIVATION AND BACKGROUND

The 0/1 Knapsack problem is a classic problem in Computer Science whose applications can be found in multiple fields, such as Operations Research, Process Scheduling and Computer Networking. It is also a problem that appears in real-world decision-making, as the one used to describe its premise: imagine a thief, whose knapsack holds only a certain amount of weight (defined as capacity c). The thief has found n items that they could steal, where each item has a certain weight and value. Unfortunately for the thief, their knapsack cannot hold all n items and they must choose the perfect combination of items to steal to gain the most value from their theft. This is called a 0/1 Knapsack problem because each item may either be taken by the thief or left behind.

There also exists a fractional knapsack problem, where our thief may take parts of items, rather than exclusively the whole item. To use another analogy, this problem could be illustrated by a test-taker determining which questions to answer. Questions that take less time to answer have a smaller weight in terms of points than questions that take longer. If the test-taker *must* complete their answer to receive points, we would have a 0/1 knapsack problem. If they may receive points for partial credit, then this would be a fractional knapsack problem.

Both 0/1 Knapsack and Fractional Knapsack illustrate the optimal-substructure property required for both greedy algorithms and dynamic programming. In this project, we will be focusing solely on the take-it-or-leave-it version of the problem. We will implement and compare three different solutions: greedy, dynamic, and by brute force. The goal of this report is to address two major issues: how close does each strategy come to the optimal solution, and how do the three strategies compare in terms of execution time.

To better understand the terminology used, we provide the following definitions to our strategies:

- **Brute Force:** A trial and error method used to find solutions through exhaustive effort. In other words, running through all possible choices to find the most optimal solution.
- **Greedy Solution:** We can assemble a globally optimal solution by making a locally optimal solution, called the greedy choice. In other words, choosing the option that looks best in the current position, without consideration to the results from sub-problems.
- **Dynamic Programming:** Like divide-and-conquer, dynamic programming solves problems by combining the solutions to sub-problems. This is especially useful when sub-problems share sub-problems.

The 0/1 Knapsack problem is classified as NP-Complete, meaning that it is nondeterministic in polynomial time. Due to this property, we do not expect a low-cost solution to the problem, and we can predict that each solution strategy will yield a different result.

We also predict, *a priori*, that the greedy strategy will not be very successful for the 0/1 Knapsack problem. This is because the greedy strategy solution finds the most optimal item *at the time* for the thief to choose, without consideration of other items, or combination of items, that may be better.

PROCEDURE

In order to implement the three solutions (brute force, dynamic, and greedy) it is important to understand some details about each. These include their descriptions and applications to the problem, pre and post conditions, and their invariants.

Brute Force

As the definition indicates, the brute force strategy tries all possible solutions and then chooses the most optimal. In this case, we must go through all of the items in the list, and find the combination of items that will accumulate the maximum value, but still fit within the capacity of the knapsack. For implementation and simplicity purposes, we will be limiting the amount of items in the list to $n = 15$.

Preconditions The sizes of the price and weight arrays must be equal and greater than 0 and their elements must be positive integers.

Postconditions The maximum value attainable with the given capacity.

Pseudocode:

RESET(x)

```

let i = 0
let overflow = false

while i < x.length
    x[i] = 0
    i++

```

BUMP(x)

```

let i = x.length - 1
let overflow = true

while ((i >= 0 and overflow))
    if x[i] = 1
        x[i] = 0

    else
        x[i] = 1
        overflow = false

    i--

```

BRUTE-FORCE(c, p, w, n)

let p[0...n-1] be a **new** array

RESET(p)

```

While the counter hasn't overflowed
    for (int i = 0 to n-1)
        if p[i] == 1
            currentWeight = currentWeight + w[i]
            if currentWeight <= c
                currentValue = currentValue + p[i]
                if currentValue > maxValue

```

```

        .....maxValue -= currentValue
        .....else
        .....currentValue -= 0

        .....currentWeight -= 0
        .....currentValue -= 0

        .....if the counter has't overflowed
            BUMP(p)

package MDiaz; /*
    Class: Bruteforce

    Author: Melany Diaz
    with assistance from: Gerry Howser

    Creation date: 3/5/2016

    Modifications:
        Date      Name      reason
        03/09/2016 Melany Diaz Implemented Binary Counter
*/

import java.util.ArrayList;

/**
    This will implement three solutions to 0/1 Knapsack and compare
    the quality of those solutions
    and the run-time cost of their solutions.

    This class will implement the Brute force solution: Try all
    possible
    combinations of xi and take the maximum value that fits within
    the
    given capacity.
*/
public class BruteForce{

    //instance variables
    private static int capacity;
    private static int [] prices;
    private static Integer [] weight;
    private static int numItems;

    private static int currentValue = 0;

```

```
private static int maxValue = 0;
private static int currentWeight = 0;

//the knapsack
public static Integer[] permutations = new Integer[numItems];

public static boolean overflow = false;

//Constructors
public BruteForce()
{
}

// Methods

/**
 * finds the solution to the 0/1 knapsack problem
 *
 * Pre-condition: Array has a length > 0, the integers in the
 *               price and weight array are positive, and the
 *               prices array has the same number of
 *               items as the weights array
 * Post condition: The returned array is filled with integer
 *               "0" and
 *               overflow is set to "false".
 * @returns the value of the best combination
 */
public static int bruteForce(int capacity, int[] prices, int[]
    weight, int numItems) {
    //checking preconditions
    assert (numItems > 0);
    assert (prices.length == weight.length);
    for (int i = 0; i < numItems; i++){
        assert (prices[i] >= 0);
        assert (weight[i] >= 0);
    }

    int[] permutable = new int[numItems];
    permutable = Reset(permutable);    //resets permutable to
        all 0's

    while (!overflow) {
        //go through the array of permutations
```

```

        for (int i = 0; i < permutable.length; i++){

            //if there is an item the thief is taking, add it '
            //s weight
            //to the weight the thief is considering taking
            if(permutable[i] == 1) {
                currentWeight += weight[i];

                //if that weight fits in the knapsack, add the
                //values of the prices
                if (currentWeight <= capacity) {
                    currentValue += prices[i];
                    //find the most optimal value for the
                    //thief
                    if(currentValue > maxValue)
                        maxValue = currentValue;
                }
                //if the weight was too much for the knapsack '
                //s capacity, value is 0
                else
                    currentValue = 0;
            }
        }

//        //to print all of the combinations of the knapsack
//        considered
//        //the last one printed is the most optimal
//        if(currentValue == maxValue)
//        System.out.println("the permutation is now: " +
//        toString(permutable) + " Weight: " + currentWeight + " value:
//        " + currentValue + "\t");

        currentWeight = 0;
        currentValue = 0;

        if (!overflow) {
            permutable = Bump(permutable);
        }
    }

    return maxValue;
}

public long TimeToFind(int capacity, int[] prices, int[]

```

```
weight, int numItems){
    long start = System.nanoTime();
    int value = bruteForce(capacity, prices, weight, numItems)
    ;
    long end = System.nanoTime();
    long duration = end - start;
    System.out.println("value for brute force: " + value);
    return duration;
}

/**resets the permutation array to all 0's.
 * Pre-condition: Array has a length > 0
 * Post condition: The returned array is filled with integer
 * "0" and
 * overflow is set to "false".
 */
public static int[] Reset(int[] x)
{
    assert (x.length > 0);
    int i = 0;
    overflow = false;
    while (i < x.length)
    {
        x[i] = 0;
        i++;
    }
    return x;
}

/**returns a permutation of all possible combinations of "1"s
 and "0"s that an array of
 * size n can have
 *
 * Pre-condition: Array contains only "0" and "1" and length
 > 0
 * Post condition: The returned array is "bumped" by 1 as a
 binary counter
 *
 * If the binary counter overflows, overflow is
 set to
 *
 * "true" otherwise overflow is set to "false"
 */
public static int[] Bump(int[] x)
{
    assert (x.length > 0);
```

```
    assert (isBinary(x));
    int i = x.length - 1;
    overflow = true;
    while ((i >= 0) && (overflow))
    {
        if (x[i] == 1)
        {
            x[i] = 0;
        }
        else
        {
            x[i] = 1;
            overflow = false;
        }
        i--;
    }
    return x;
}

//takes the array of permutaitons and transforms it to a
//printable string
public static String toString(int[] x)
{
    String result = " ";
    int i = 0;
    while (i < x.length)
    {
        result = result + x[i];
        i++;
    }
    return result;
}

//Prints out the different permutations of a binary array
public void printPermutations(int[] permutable) {
    permutable = this.Reset(permutable);
    while (!this.overflow) {
        System.out.println("the permutation is now: " + this.
            toString(permutable) + "\t");
        if (!this.overflow) {
            permutable = this.Bump(permutable);
        }
    }
}
```



```
//assert methods
//confirms that the integers in the permutation array are just
  0s and 1s
public static boolean isBinary(int [] x)
{
    boolean result = true;
    int i = 0;
    while ((i <= x.length) && (result))
    {
        if ((x[i] != 0) && (x[i] != 1))
        {
            result = false;
        }
        i++;
    }
    return result;
}
}
```

Invariant: Since the invariant for this requires that we know the most optimal solution to begin with (in other words, run this method for the invariant) it would be redundant to use. So, we will assume that the invariant is already asserted, thus confirming the initialization, maintenance and termination of it.

Now that we know the implementation details, we conclude by writing an appropriate class that we may use for our study. An example of such class is shown in the figure bellow.

```

/**
 * finds the solution to the 0/1 knapsack problem
 *
 * Pre-condition: Array has a length > 0, the integers in the price and
 * weight array are positive, and the prices array has the same number of
 * items as the weights array Post condition: The returned array is filled
 * with integer "0" and
 *
 * overflow is set to "false".
 * @returns the value of the best combination
 */
public static int bruteForce(int capacity, int[] prices, int[] weight, int numItems)
    //checking preconditions
    assert (numItems > 0);
    assert (prices.length == weight.length);
    for (int i = 0; i < numItems; i++){
        assert (prices[i] >= 0);
        assert (weight[i] >= 0);
    }

    int[]permutable = new int[numItems];
    permutable = Reset(permutable); //resets permutable to all 0's

    while (!overflow) {
        //go through the array of permutations
        for (int i = 0; i < permutable.length; i++){

            //if there is an item the thief is taking, add it's weight
            //to the weight the thief is considering taking
            if(permutable[i] == 1) {
                currentWeight += weight[i];

                //if that weight fits in the knapsack, add the
                //values of the prices
                if (currentWeight <= capacity) {
                    currentValue += prices[i];
                    //find the most optimal value for the thief
                    if(currentValue > maxValue)
                        maxValue = currentValue;
                }

                //if the weight was too much for the knapsack's capacity, value
                //is 0
                else
                    currentValue = 0;
            }
        }

        currentWeight = 0;
        currentValue = 0;

        if (!overflow) {
            permutable = Bump(permutable);
        }

    }

    return maxValue;
}

```

```
/**resets the permutation array to all 0's.  
* Pre-condition: Array has a length > 0  
* Post condition: The returned array is filled with integer "0" and  
* overflow is set to "false".  
*/  
public static int[] Reset(int[] x)  
{  
    assert (x.length > 0);  
    int i = 0;  
    overflow = false;  
    while (i < x.length)  
    {  
        x[i] = 0;  
        i++;  
    }  
    return x;  
}  
  
/**returns a permutation of all possible combinations of "1"s and "0"s  
* that an array of size n can have  
*  
* Pre-condition: Array contains only "0" and "1" and length > 0  
* Post condition: The returned array is "bumped" by 1 as a binary counter  
* If the binary counter overflows, overflow is set to  
* "true" otherwise overflow is set to "false"  
*/  
public static int[] Bump(int[] x)  
{  
    assert (x.length > 0);  
    assert (isBinary(x));  
    int i = x.length - 1;  
    overflow = true;  
    while ((i >= 0) && (overflow))  
    {  
        if (x[i] == 1)  
        {  
            x[i] = 0;  
        }  
        else  
        {  
            x[i] = 1;  
            overflow = false;  
        }  
        i--;  
    }  
    return x;  
}
```

Dynamic Programming

Preconditions The sizes of the price and weight arrays must be equal and greater than 0 and their elements must be positive integers.

Postconditions The maximum value attainable with the given capacity.

Pseudocode:

DYNAMIC (c, p, w, n)

let f[0...n+1][0...c+1] be a **new** array of integers
 let k[0...n+1][0...c+1] be a **new** array of booleans

```
for (i = 1 to n)
  for (w = 1 to c)
    f1 = f[i - 1][w]

    f2 = -infinity
    if (w[i] <= w)
      f2 = p[i] + f[i - 1][w - w[i]]
    f[i][w] = max(f1, f2)
    k[i][w] = f2 > f1
```

```
let b[0...n+1] be a new array of booleans
for (int m = n to 0)
  if k[m][c]
    t[m] = true
    c = c - w[m]
  else
    t[m] = false
```

```
for (int i = 0 to t.length)
  if t[i]
    maxValue = maxValue + p[i]
```

Invariant: Since the invariant for this requires that we know the most optimal solution to begin with (in other words, run this method for the invariant) it would be redundant to use. So, we will assume that the invariant is already asserted, thus confirming the initialization, maintenance and termination of it.

Now that we know the implementation details, we conclude by writing an appropriate class that we may use for our study. An example of such class is shown in the figure bellow.

```

/**
 * finds the solution to the 0/1 knapsack problem
 *
 * Pre-condition: Array has a length > 0, the integers in the price and weight array are positive,
 *               the prices array has the same number of items as the weights array, and the
 *               weights are integers
 *
 * Post condition: The return value is the max value with the capacity allotted.
 */
public static int dynamic(int capacity, int[] prices, int[] weight, int numItems) {
    //checking preconditions
    assert (numItems > 0);
    assert (prices.length == weight.length);
    for (int i = 0; i < numItems; i++){
        assert (prices[i] >= 0);
        assert (weight[i] >= 0);
    }

    int[][] f = new int[numItems + 1][capacity + 1];
    boolean[][] knapsack = new boolean[numItems + 1][capacity + 1];

    //Build table k[][] in bottom up manner
    for (int i = 1; i <= numItems; i++) {
        for (int w = 1; w <= capacity; w++) {
            //don't take the item
            int f1 = f[i - 1][w];

            //take it
            int f2 = Integer.MIN_VALUE;
            if (weight[i] <= w) {
                f2 = prices[i] + f[i - 1][w - weight[i]];
            }

            //select the better of two options
            f[i][w] = Math.max(f1, f2);
            knapsack[i][w] = (f2 > f1);
        }
    }

    //determine which items to take
    boolean[] take = new boolean[numItems + 1];
    for (int n = numItems, w = capacity; n > 0; n--) {
        if (knapsack[n][w]) {
            take[n] = true;
            w = w - weight[n];
        } else
            take[n] = false;
    }

    //print results
    System.out.println("item" + "\t" + "profit" + "\t" + "weight" + "\t" + "take");
    for (int n = 1; n < numItems+1; n++)
        System.out.println(n + "\t" + prices[n] + "\t" + weight[n] + "\t" + take[n]);

    //finds the max value of the most optimal knapsack the thief can fill
    for(int i = 0; i < take.length; i++){
        if(take[i]) {
            maxVal += prices[i];
        }
    }
    return maxVal;
}

```

Greedy Algorithm

As the definition indicated, the greedy algorithm will choose the most optimal solution at the current moment. In this case, we will form a price density array in non-ascending order, and then find the greatest value for the knapsack using the greedy premise.

In order to form the price density array in non-ascending order, we will be using the HeapSort sorting algorithm written for a previous report. This sorting algorithm was chosen due to its faster run-time in comparison to other sorting methods, such as MergeSort, QuickSort, or InsertionSort. HeapSort, like MergeSort, has the same time complexity for its best, worst, and random cases ($O(n * \ln_2(n))$). The reason that HeapSort was chosen over MergeSort, however is that even with the same time complexity, HeapSort tends to terminate before MergeSort, due to its leading constants. QuickSort was also a strong choice, however the worst-case for QuickSort yields a time complexity of ($O(n^2)$), therefore making HeapSort the strongest choice for this scenario.

Preconditions The sizes of the price and weight arrays must be equal and greater than 0 and their elements must be positive integers.

Postconditions The maximum value attainable with the given capacity.

Pseudocode:

GREEDY(*c*, *p*, *w*, *n*)

sort the *p* in non-ascending order and rearrange *w* accordingly
so price index alligns with weight index

let currentWeight = 0
let w = 0

while currentWeight <= *c* AND *w* < *n*
 if newWeights[*w*] <= (*c* - currentWeight)
 currentWeight = currentWeight + newWeights[*w*]
 maxValue = maxValue + *p*[*w*]
 w++

Invariant: Since the invariant for this requires that we know the most optimal solution to begin with (in other words, run this method for the invariant) it would be redundant to use. So, we will assume that the invariant is already asserted, thus confirming the initialization, maintenance and termination of it.

Now that we know the implementation details, we conclude by writing an appropriate class that we may use for our study. An example of such class is shown in the figure bellow.

```

/**
 * finds the solution to the 0/1 knapsack problem
 *
 * Pre-condition: Array has a length > 0, the integers in the price and weight array are positive, and the
 *                prices array has the same number of items as the weights array
 *
 * Post condition: The return value is the max value with the capacity allotted.
 */
public static int greedy(int capacity, int[] prices, int[] weight, int numItems) {
    //checking preconditions
    assert (numItems > 0);
    assert (prices.length == weight.length);
    for (int i = 0; i < numItems; i++){
        assert (prices[i] >= 0);
        assert (weight[i] >= 0);
    }

    //sort price list in non-ascending order
    HeapSort h = new HeapSort();
    int[] OGPrices = Arrays.copyOf(prices, numItems);

    h.heapSort(prices);
    /    System.out.println(Arrays.toString(prices));

    //rearrange weight list accordingly
    orderedWeights = newWeights(prices, OGPrices, weight, numItems);
    /    System.out.println(Arrays.toString(orderedWeights));

    //fill the knapsack using the greedy idea
    currentWeight = 0;
    int w = 0;
    while(currentWeight <= capacity && w < numItems){
        if(orderedWeights[w] <= (capacity - currentWeight)) {
            /    System.out.print(prices[w] + " ");
            currentWeight += orderedWeights[w];
            maxValue += prices[w];
        }
        w++;
    }
    return maxValue;
}

```

Method used for Comparisons

In order to compare the three methods, we will be generating a random list of items (defined by their price, p_i , and their weight w_i) and testing each solution using this list. To generate a reasonable solution, this list will be made of integers ranging from 5 to 100.

As per the knapsack, we will use a constant capacity c set to a weight of 100 ($c = 100$.)

TESTING

Program testing is the process used to help identify the correctness, completeness, and quality of a class. The process of testing involves executing a program with the intent of finding errors and bugs. One of the goals for this project was to find a way to create a test driver so that it exercised the program. The following table shows the tests the program went through, the expected results, the actual results, and the solution.

Test	Expected Result	Actual Result	Remedy
Duplicate integers in price array	Program executes as usual	Issues with the reordering of the weight array (as explained in the Problems encountered)	modify the code responsible for re-ordering the weight array according with the price array
Duplicate integers in weight array	Program executes as usual	as expected	N/A
Having 0 items	Program executes as usual	As expected. This was useful for seeing which approach was initially quicker (when $n = 0$.) The results showed brute force as the fastest and greedy as the slowest	N/A
Having negative items	A negative array exception	As expected	N/A
Having negative integers in the price Array	An exception	Surprisingly, the array still compiled. Brute force and Dynamic gave 0 as the most optimal choice. Greedy, on the other hand, has our thief paying the store to steal it, as it showed the most optimal solution to be \$ -101	N/A
Having negative integers in the weights array	An exception	As expected, an index out of bounds exception	N/A

PROBLEMS ENCOUNTERED

As with any programming project, a programmer must be able to keep track of any problems encountered and of the solutions found to counter them. There were two main problems that were encountered during the development process. Following is a description of these and how I chose to tackle them.

The first problem began with translating the pseudocode of the Dynamic Programming approach into java. The dynamic approach must remain one-relative in order to properly execute. However, I had mistakenly changed all of the arrays to be zero-relative, therefore finding inaccurate results whenever I ran my code. After learning that the relative-ness of the arrays were so specific for this approach, it was an easy error to find and fix.

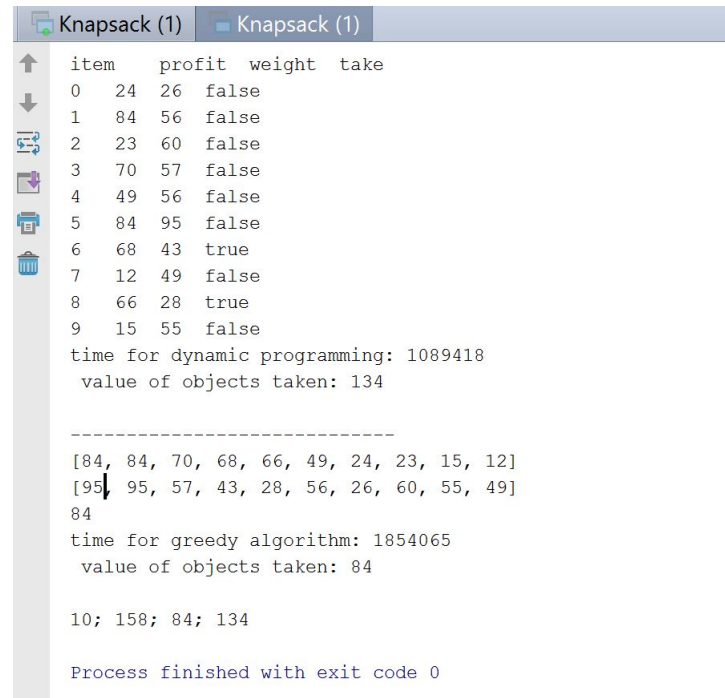
The second problem occurred during the Greedy algorithm approach. This approach requires that the price array be sorted into non-ascending order. Because of this rearrangement of the elements, it is required, consequently, that the weight array be rearranged appropriately. I felt confident in the code I had originally written to do this, until I realized that this code would be ineffective if the price array ever had a duplicate integer. The original code can be seen in figure 1.

```
//used to rearrange weight array to match new, ordered, price array
public static int[] newWeights(int[] newPrices, int[] OGPrices, int[] weight, int numItems){
    for(int i = 0; i < numItems; i++){
        int w = 0;
        while( w < numItems) {
            if (newPrices[i] == OGPrices[w]) {
                nw[i] = weight[w];
            }

            w++;
        }
    }
    return nw;
}
```

Figure 1: Original code that gave erroneous results

The error occurred when there was a duplicate in the price array can be seen in the following figure. The upper half shows the original price and weight arrays, and the bottom half (the horizontal representation) shows the newly sorted price and weight arrays. As can be seen, originally there was an item worth \$84 with a weight of 54 lbs and a second item, also worth \$84, that weighed 95 lbs. The error can be seen in the display of the newly rearranged graphs, where both items worth \$84 now had an equal weight of 95lbs.



```

Knapsack (1)
item  profit  weight  take
0   24   26   false
1   84   56   false
2   23   60   false
3   70   57   false
4   49   56   false
5   84   95   false
6   68   43   true
7   12   49   false
8   66   28   true
9   15   55   false

time for dynamic programming: 1089418
value of objects taken: 134

-----
[84, 84, 70, 68, 66, 49, 24, 23, 15, 12]
[95, 95, 57, 43, 28, 56, 26, 60, 55, 49]
84
time for greedy algorithm: 1854065
value of objects taken: 84

10; 158; 84; 134

Process finished with exit code 0

```

Figure 2: Console results displaying error

After this error was found, all that was left to do was find the code that was responsible and change it. The newly written and modified code can be seen in figure 3.

```

//used to rearrange weight array to match new, ordered, price array
public static int[] newWeights(int[] newPrices, int[] OGPrices, int[] weight, int numItems){
    for(int j = 0; j < numItems; j++){
        int i = 0;
        boolean found = false;
        while (!found && i < numItems){
            if(newPrices[j] == OGPrices[i]){
                OGPrices[i] = Integer.MAX_VALUE;
                nw[j] = weight[i];
                found = true;
            }
            i++;
        }
    }
    return nw;
}

```

Figure 3: Modified code

EXPERIMENTAL ANALYSIS AND ASYMPTOTIC RUN-TIME COMPARISON

After confirming that the program meets its conditions, and that it produces the required results, we now must compare the run-time complexities of the three solutions with their asymptotic run-times. This is also the moment to compare the results of the three strategies to conclude which solutions yield the most optimal results.

In order to test our solutions, we generated different lists of n items, where the lists contained the randomized weights and prices of each item. Each randomized weight and price was a positive integer between 5-100; this confirmed that our tests aligned with the preconditions required by each strategy. To best compare the time complexity of our results, we plotted the time behavior for each case as a function of n . Similarly, to best compare the results of each case, we plotted their answers. Our results are described in the following sections.

Brute Force

The Brute Force strategy requires that the thief must check every combination possible and then choose the most optimal. The time complexity and results of this strategy are explained in the following sections.

Time Complexity

Notice, in the code for the Brute Force strategy, the for-loop inside the while-loop. Due to these nested loops, it doesn't come to a surprise that the time complexity for Brute Force resembles $O(n^2)$ Asymptotic time.

```

while (!overflow) {
    //go through the array of permutations
    for (int i = 0; i < permutable.length; i++){

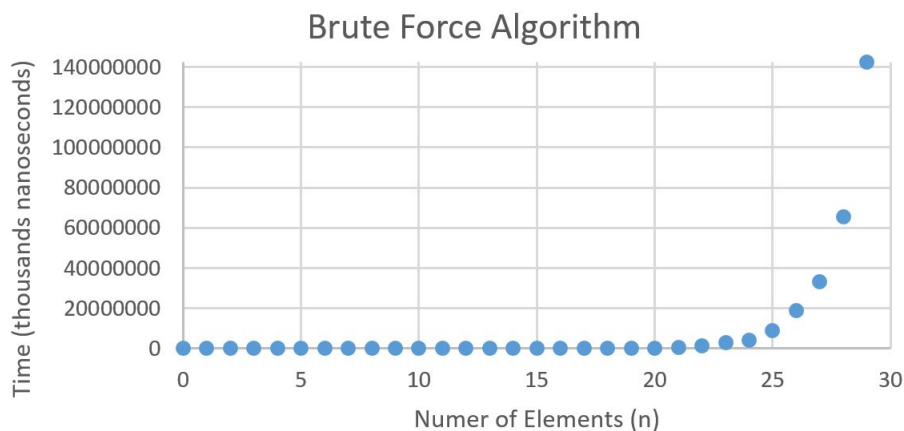
        //if there is an item the thief is taking, add it's weight
        //to the weight the thief is considering taking
        if(permutable[i] == 1) {
            currentWeight += weight[i];

            //if that weight fits in the knapsack, add the
            //values of the prices
            if (currentWeight <= capacity) {
                currentValue += prices[i];
                //find the most optimal value for the thief
                if(currentValue > maxValue)
                    maxValue = currentValue;
            }
            //if the weight was too much for the knapsack's capacity, value is 0
        } else
            currentValue = 0;
    }
}

```

Figure 4: Code for Brute Force displaying nested loops

It is interesting how the distance between each iteration begins to increase substantially after $n > 25$. It is also interesting that the time analysis of this strategy shows no outliers in its outcome. The fact that there aren't any outliers clearly displayed is incentive enough to explore what this graph may look like from a closer perspective. The figure following shows the same graph, but with a more specific scale for the y-axis.

Figure 5: Time Complexity of Brute Force Displaying similarities to $O(n^2)$ complexity.

Note how the close up displays an outlier that had not shown before, at $n = 9$.

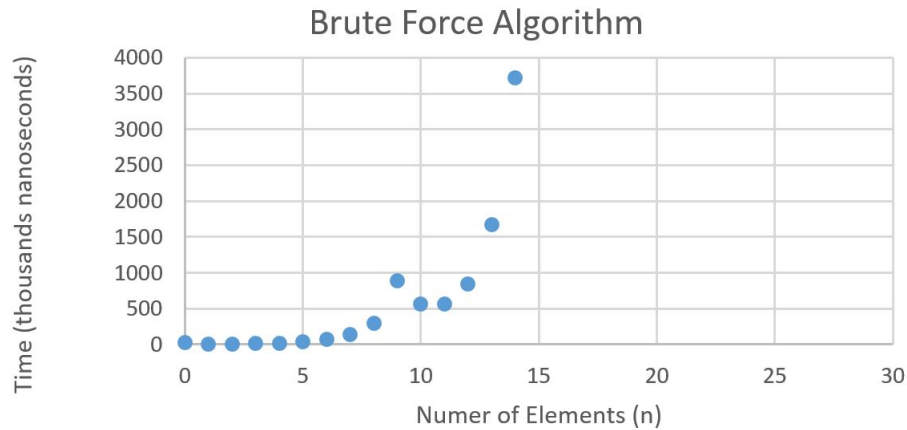


Figure 6: Closer look at the duration of Brute Force

Results

The results of the Brute Force algorithm are shown in the following figure. These results represent the total value of the most optimal combination of items that the thief could have taken.

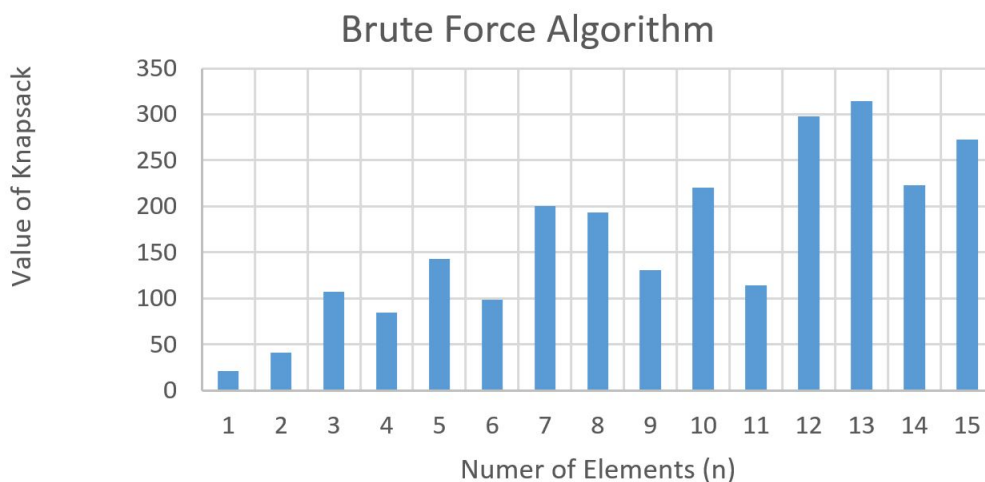


Figure 7: Value of most optimal solution

Dynamic Programming

The dynamic solution to the 0/1 Knapsack problem takes a divide-and-conquer approach towards finding the solution. The following sections will describe what we found when we used this approach to find the most optimal solution to the problem.

Time Complexity

The following figure displays the duration that it took for this approach to find the most optimal solution for a certain amount of items n . Notice how at first, when $n \leq 8$ the times appear to be very erratic: increasing in an unpredictable manner. However, after $n = 9$ the results seem to become less unstable, and begin to display a more linear pattern.

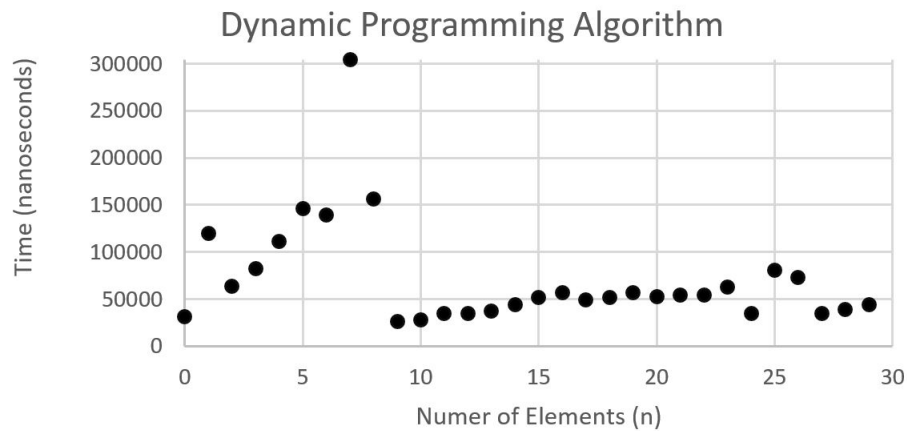


Figure 8: Time Complexity of Dynamic Programming

Results

The results of the Brute Force algorithm are shown in the following figure. These results represent the total value of the most optimal combination of items that the thief could have taken.

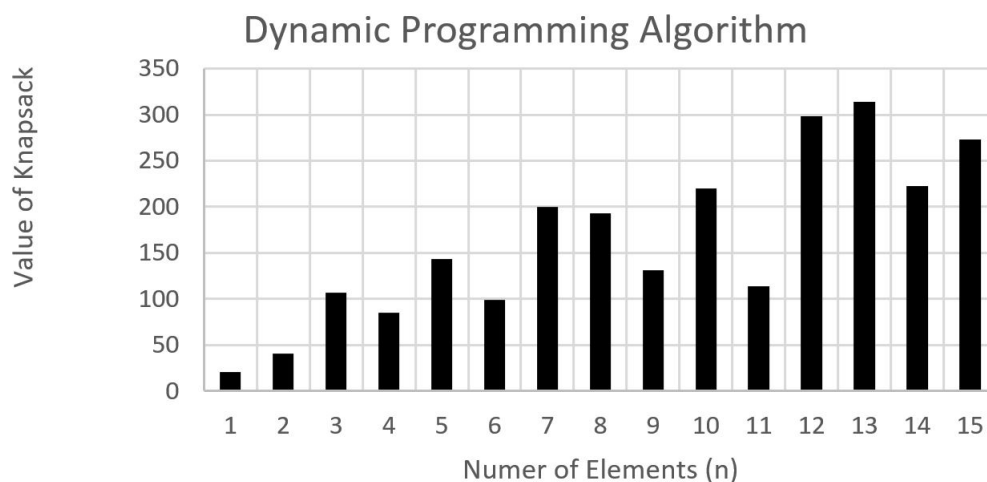


Figure 9: Value of most optimal solution

Greedy Algorithm

The greedy algorithm for the 0/1 Knapsack takes an interesting approach of choosing the most valuable item *at the time* for the thief to take in their knapsack. Of the three strategies, this approach was the most unpredictable, further confirming our suspicions that the greedy algorithm isn't effective for this problem.

Time Complexity

The following figure displays the duration that it took for this approach to find the most optimal solution for a certain amount of items n . Notice how at first, this appears to be extremely linear, with a remarkable exception when $n = 0$.

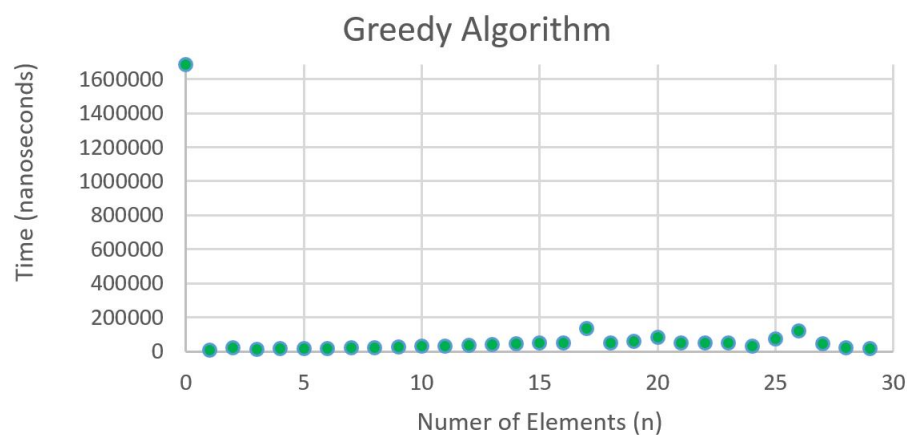


Figure 10: Time Complexity of Greedy Algorithm

This second figure shows what happens when we take a closer look to the duration of the greedy algorithm. Notice how now, the results tend to steadily incline, until $n = 16$ where the results become more unpredictable.

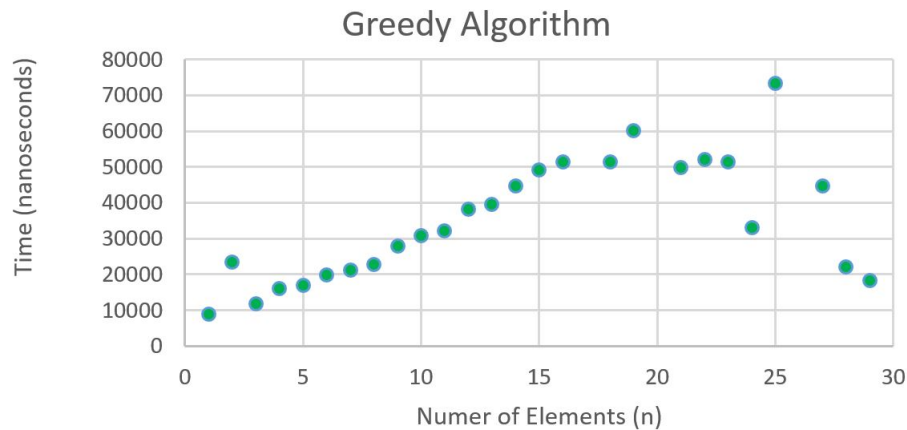


Figure 11: A Closer Look to the Time Complexity of Greedy Algorithm

Results

The results of the Brute Force algorithm are shown in the following figure. These results represent the total value of the most optimal combination of items that the thief could have taken.

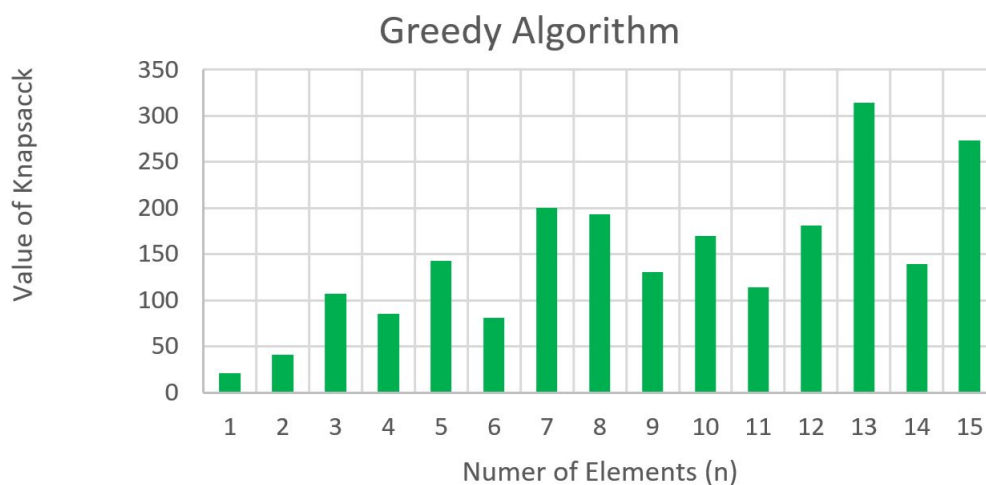


Figure 12: Value of most optimal solution

Comparison of all Strategies

In order to best compare the three strategies, to find out which was the most efficient in terms of time and which gave the desired solutions, it is best to see them side-by-side. The following figures display the same results as above but in unison.

Time Complexity

As can be seen in the following figure, the greedy algorithm stayed consistently below 2000000 nanoseconds (aside for the outlier when $n = 0$). On the other hand the brute force approach can be seen quickly becoming the slowest of the strategies, as it can be seen steadily increasing the graph.

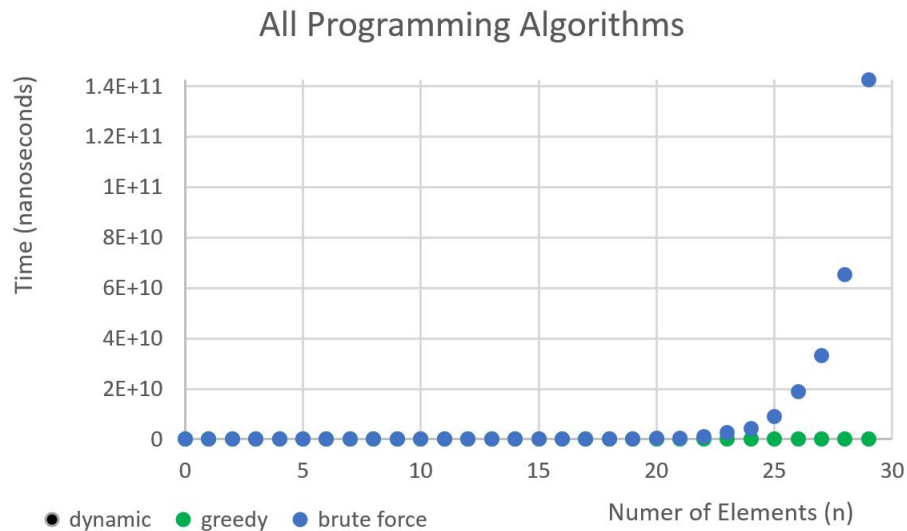


Figure 13: Time Complexities of each approach

Even though the above figure shows the three approaches, greedy and brute force hide the durations of the dynamic approach. In order to see these results, we decided to close in on the graph, and expose the results for dynamic programming.

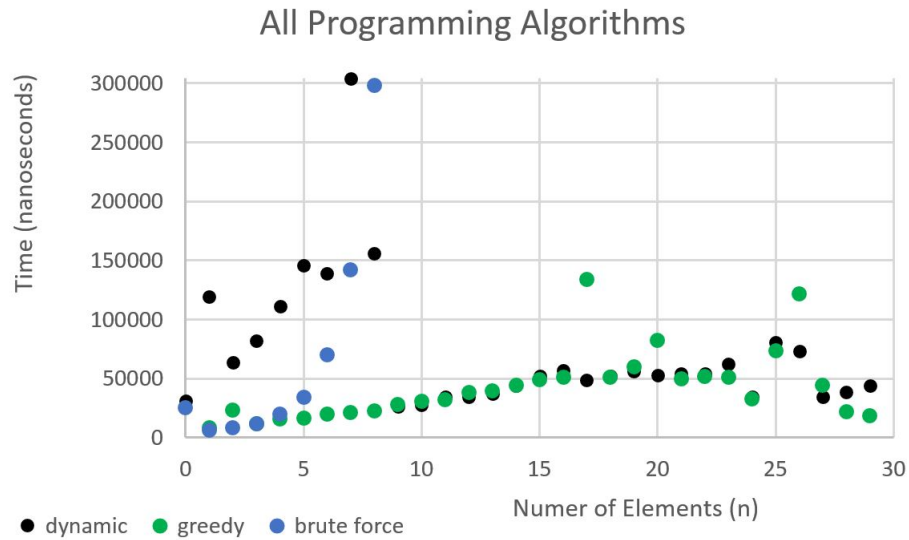
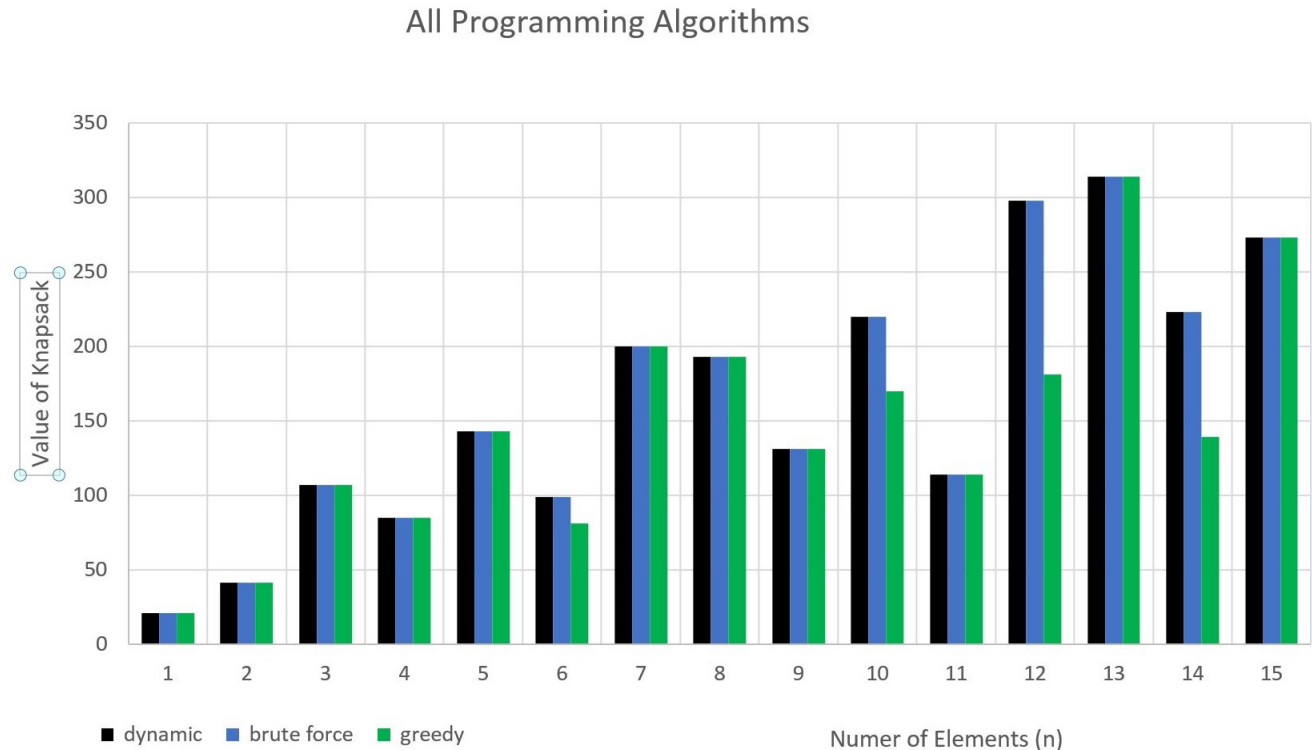


Figure 14: Time Complexities of each approach

As can be seen in the above figure, Dynamic programming *begins* as the slowest of the three, but consistently stays near the time results for the greedy algorithm once the brute force approach surpassed it.

Results

As can be seen in the following figure, the Brute Force and Dynamic approaches gave the same answer to what the total value of the most optimal knapsack would be. On the other hand, and as predicted, the brute force approach occasionally gave solutions that were less than the actual optimal answer.



Results

CONCLUSIONS

After constructing a knapsack program that implemented the brute force, dynamic and greedy algorithm solutions, we were able to analyze and compare the run-time behaviors of each strategy. We have found that The dynamic programming gives the most optimal solution due to its it's speed compared to the brute force option, and the fact that the greedy algorithm didn't yield the most beneficial results. With a better understanding of each solution strategy, we have learned how to generate different complexities, implement and scrutinize a class so that it surpasses an intensive testing phase, and have understood the different algorithms used to solve the 0/1 knapsack problem.

We predicted, *a priori*, that the greedy strategy will not be very successful for the 0/1 Knapsack problem. After testing each solution we found that our predictions were confirmed, and that the greedy algorithm does not yield the desired solution.

APPENDIX A: Main Class

The following is the Java class written for the main class.

```

package MDiaz;

/*
   Class: Knapsack

   Author: Melany Diaz

   0/1 is a classic problem in Computer Science. The idea is to
       load a knapsack of fixed capacity with
       items of known weights and profits such that the profit is
       maximized and the knapsack is not overloaded.

   The 0/1 Knapsack problem is NP-Complete so we do not expect a
       low-cost solution to the problem.

   Modifications:
   Date          Name          reason
   3/10/16       Melany Diaz   Debugging and finalizing

   This project will implement three solutions (brute force ,
       greedy algorithm , and dynamic programming) to 0/1 Knapsack
       and compare the quality of those solutions and the run-time
   .
   This class will create the knapsack, the items, and implement
       the three
       strategies to it to find their different execution times.
   */

import java.util.Arrays;
import java.util.Random;

public class Knapsack {
    public static void main(String[] args) {
        //the number of items for the robber to take
        int numItems = 20;
        //make random generator for the weights and prices
        Random generator = new Random();

        //capacity of the Knapsack (to remain constant)

```

```

int capacity = 100;

//Array of prices
//for brute force
int [] prices = new int [numItems];
//for greedy
int [] pricesg;
//for dynamic (must be one relevant)
int [] pricesd = new int [numItems + 1];

//Array of weights
//for brute force and greedy
int [] weight = new int [numItems];
//for dynamic (must be 1 relevant)
int [] weightd = new int [numItems + 1];

//fill the price and weight arrays
//integers between 5-100
for (int i = 0; i < numItems; i++) {
    prices[i] = generator.nextInt(96) + 5;
    weight[i] = generator.nextInt(96) + 5;
}

//price array for greedy, completed
pricesg = Arrays.copyOf(prices, numItems);

//price array for dynamic, completed
//must be one relevant with the 0 index equal to 0
for (int i = 1; i < numItems + 1; i++) {
    pricesd[0] = 0;
    weightd[0] = 0;
    pricesd[i] = prices[i - 1];
    weightd[i] = weight[i - 1];
}

//          //print scenario details
//          System.out.println(Arrays.toString(prices) + "\n" +
Arrays.toString(weight));
//          System.out.println();
//          System.out.println("numTimes: " + numItems);

////////////////////// BRUTE FORCE

```

```

//////////////////////////
//make the cases and print the time it took to each
BruteForce bf = new BruteForce();
long timebf = bf.TimeToFind(capacity , prices , weight ,
    numItems);

//          //To print the time it took to solve and the full
// value of the knapsack
//          System.out.println(" Duration for brute force: " +
// timebf + "\n");

////////////////////////// DYNAMIC PROGRAMMING
//////////////////////////
Dynamic dp = new Dynamic();
long timedyn = dp.TimeToFind(capacity , pricesd ,
    weightd , numItems);

//          //To print the time it took to solve and the full
// value of the knapsack
//          System.out.println("duration for dynamic programming
// : " + timedyn + "\n");

////////////////////////// GREEDY ALGORITHM
//////////////////////////
Greedy greedy = new Greedy(capacity , pricesg , weight ,
    numItems);
long timegreedy = greedy.TimeToFind(capacity , pricesg ,
    weight , numItems);

//          //To print the time it took to solve and the full
// value of the knapsack
//          System.out.println("duration for greedy: " +
// timegreedy + "\n");

//////////////////////////prints out CSV for results
//////////////////////////

// numtimes; brute force; dynamic; greedy
// for time complexities
System.out.println(numItems + ";" + timebf + ";" +
    timedyn + ";" + timegreedy);

```

```
}  
}
```

Appendix B: BruteForce

The following is the Java class written for Brute Force strategy.

```

package MDiaz; /*
    Class: BruteForce

    Author: Melany Diaz
    with assistance from: Gerry Howser

    Creation date: 3/5/2016

    Modifications:
        Date          Name          reason
        03/09/2016    Melany Diaz    Implemented Binary Counter
*/

import java.util.ArrayList;

/**
    This will implement three solutions to 0/1 Knapsack and compare
    the quality of those solutions
    and the run-time cost of their solutions.

    This class will implement the Brute force solution: Try all
    possible
    combinations of xi and take the maximum value that fits within
    the
    given capacity.
*/
public class BruteForce{

    //instance variables
    private static int capacity;
    private static int [] prices;
    private static Integer [] weight;
    private static int numItems;

    private static int currentValue = 0;
    private static int maxValue = 0;
    private static int currentWeight = 0;

    //the knapsack
    public static Integer [] permutations = new Integer[numItems];

```



```

public static boolean overflow = false;

//Constructors
public BruteForce()
{
}

// Methods

/**
 * finds the solution to the 0/1 knapsack problem
 *
 * Pre-condition: Array has a length > 0, the integers in the
 *               price and weight array are positive, and the
 *               prices array has the same number of
 *               items as the weights array
 * Post condition: The returned array is filled with integer
 *               "0" and
 *
 *               overflow is set to "false".
 * @returns the value of the best combination
 */
public static int bruteForce(int capacity, int[] prices, int[]
    weight, int numItems) {
    //checking preconditions
    assert (numItems > 0);
    assert (prices.length == weight.length);
    for (int i = 0; i < numItems; i++){
        assert (prices[i] >= 0);
        assert (weight[i] >= 0);
    }

    int[] permutable = new int[numItems];
    permutable = Reset(permutable);    //resets permutable to
        all 0's

    while (!overflow) {
        //go through the array of permutations
        for (int i = 0; i < permutable.length; i++){

            //if there is an item the thief is taking, add it's
            //weight
            //to the weight the thief is considering taking

```

```

        if(permutable[i] == 1) {
            currentWeight += weight[i];

            //if that weight fits in the knapsack, add the
            //values of the prices
            if (currentWeight <= capacity) {
                currentValue += prices[i];
                //find the most optimal value for the
                thief
                if(currentValue > maxValue)
                    maxValue = currentValue;
            }
            //if the weight was too much for the knapsack's
            //capacity, value is 0
            else
                currentValue = 0;
        }
    }

    // //to print all of the combinations of the knapsack
    // considered
    // //the last one printed is the most optimal
    // if(currentValue == maxValue)
    // System.out.println("the permutation is now: " +
    // toString(permutable) + " Weight: " + currentWeight + " value:
    // " + currentValue + "\n");

    currentWeight = 0;
    currentValue = 0;

    if (!overflow) {
        permutable = Bump(permutable);
    }
}

return maxValue;
}

public long TimeToFind(int capacity, int[] prices, int[]
weight, int numItems){
    long start = System.nanoTime();
    int value = bruteForce(capacity, prices, weight, numItems)
    ;
    long end = System.nanoTime();
}

```

```

        long duration = end - start;
        System.out.println("value_for_brute_force:" + value);
        return duration;
    }

    /**resets the permutation array to all 0's.
     * Pre-condition: Array has a length > 0
     * Post condition: The returned array is filled with integer
     * "0" and
     * overflow is set to "false".
     */
    public static int[] Reset(int[] x)
    {
        assert (x.length > 0);
        int i = 0;
        overflow = false;
        while (i < x.length)
        {
            x[i] = 0;
            i++;
        }
        return x;
    }

    /**returns a permutation of all possible combinations of "1"s
     and "0"s that an array of
     * size n can have
     *
     * Pre-condition: Array contains only "0" and "1" and length
     > 0
     * Post condition: The returned array is "bumped" by 1 as a
     binary counter
     *
     * If the binary counter overflows, overflow is
     set to
     *
     * "true" otherwise overflow is set to "false"
     */
    public static int[] Bump(int[] x)
    {
        assert (x.length > 0);
        assert (isBinary(x));
        int i = x.length - 1;
        overflow = true;
        while ((i >= 0) && (overflow))
        {

```

```

        if (x[i] == 1)
        {
            x[i] = 0;
        }
        else
        {
            x[i] = 1;
            overflow = false;
        }
        i--;
    }
    return x;
}

//takes the array of permutaitons and transforms it to a
//printable string
public static String toString(int[] x)
{
    String result = "_";
    int i = 0;
    while (i < x.length)
    {
        result = result + x[i];
        i++;
    }
    return result;
}

//Prints out the different permutations of a binary array
public void printPermutations(int[] permutable) {
    permutable = this.Reset(permutable);
    while (!this.overflow) {
        System.out.println("the_permutation_is_now:_ " + this.
            toString(permutable) + "\t");
        if (!this.overflow) {
            permutable = this.Bump(permutable);
        }
    }
}

//assert methods
//confirms that the integers in the permutation array are just
//0s and 1s
public static boolean isBinary(int[] x)
{

```

```
    boolean result = true;
    int i = 0;
    while ((i <= x.length) && (result))
    {
        if ((x[i] != 0) && (x[i] != 1))
        {
            result = false;
        }
        i++;
    }
    return result;
}
}
```

Appendix C: Greedy Class

The following is the Java class written for the Greedy Algorithm.

```
package MDiaz; /*
    Class: Greedy

    Author: Melany Diaz
    with assistance from: Gerry Howser

    Creation date: 3/5/2016

    Modifications:
        Date          Name          reason
        3/10/2016     Melany Diaz  Debugging
*/

import java.lang.reflect.Array;
import java.util.ArrayList;
import java.util.Arrays;

/**
    In this project you will implement three solutions to 0/1
    Knapsack and compare the quality of those solutions
    and the run-time cost of their solutions.

    This class will implement the Greedy solution: Form a price
    density array, sort it in non-ascending order,
    and use a greedy strategy to fit the greatest value into the
    knapsack.

    */
public class Greedy {

    //instance variables
    private static int capacity;
    private static int [] prices;
    private static int [] weight;
    private static int numItems;

    private static int maxValue;
    private static int currentWeight;
```

```
//rearranged weight array
public static int[] orderedWeights;

//Constructors
public Greedy(int capacity, int[] prices, int[] weight, int
    numItems)
{
    this.capacity = capacity;
    this.numItems = numItems;
    this.prices = prices;
    this.weight = weight;
    this.orderedWeights = new int[numItems];
}

// Methods

/**
 * finds the solution to the 0/1 knapsack problem
 *
 * Pre-condition: Array has a length > 0, the integers in the
 *               price and weight array are positive, and the
 *               prices array has the same number of items
 *               as the weights array
 *
 * Post condition: The return value is the max value with the
 *               capacity allotted.
 */
public static int greedy(int capacity, int[] prices, int[]
    weight, int numItems) {
    //checking preconditions
    assert (numItems > 0);
    assert (prices.length == weight.length);
    for (int i = 0; i < numItems; i++){
        assert (prices[i] >= 0);
        assert (weight[i] >= 0);
    }

    //sort price list in non-ascending order
    HeapSort h = new HeapSort();
    int[] OGPrices = Arrays.copyOf(prices, numItems);

    h.heapSort(prices);
    //    System.out.println(Arrays.toString(prices));
}
```

```

        //rearrange weight list accordingly
        orderedWeights = newWeights(prices , OGPrices , weight ,
            numItems);
    //        System.out.println(Arrays.toString(orderedWeights));

    //fill the knapsack using the greedy idea
    currentWeight = 0;
    int w =0;
    while(currentWeight <= capacity && w < numItems){
        if(orderedWeights[w] <= (capacity - currentWeight)) {
    //            System.out.print(prices[w] + " ");
            currentWeight += orderedWeights[w];
            maxValue += prices[w];

        }
        w++;
    }
    return maxValue;
}

//used to rearrange weight array to match new, ordered, price array
public static int [] newWeights(int [] newPrices , int [] OGPrices
    , int [] weight , int numItems){
    for(int j = 0; j < numItems; j++){
        int i = 0;
        boolean found = false;
        while (!found && i < numItems){
            if(newPrices[j] == OGPrices[i]){
                OGPrices[i] = Integer.MAX_VALUE;
                orderedWeights[j] = weight[i];
                found = true;
            }
            i++;
        }
    }
    return orderedWeights;
}

//used to time how long it takes to find the solution using greedy algorithm
//returns the value of the knapsack stolen
public long TimeToFind(int capacity , int [] prices , int []

```



```
weight, int numItems){  
    long start = System.nanoTime();  
    int value = greedy(capacity, prices, weight, numItems);  
    long end = System.nanoTime();  
    long duration = end - start;  
    System.out.println("value_for_greedy_algorithm:_" + value)  
    ;  
    return duration;  
}  
  
}
```

Appendix C: Dynamic Class

The following is the Java class written for the Dynamic Programming strategy.

```
package MDiaz; /*
    Class: Dynamic

    Author: Melany Diaz
    with assistance from cs.princeton.edu and Gerry Howser

    Creation date: 3/5/2016

    Modifications:
        Date      Name      reason
        3/10/16   Melany Diaz enhanced
*/

import java.util.ArrayList;
import java.util.Arrays;

/**
    In this project you will implement three solutions to 0/1
    Knapsack and compare the quality of those solutions
    and the run-time cost of their solutions.

    This class will implement the Dynamic Programming solution.
*/
public class Dynamic {

    //instance variables
    private static int capacity;
    private static int [] prices;
    private static Integer [] weight;
    private static int numItems;

    private static int maxVal;

    //Constructors
    public Dynamic()
    {
    }
}
```

```

// Methods

/**
 * finds the solution to the 0/1 knapsack problem
 *
 * Pre-condition: Array has a length > 0, the integers in the
 *               price and weight array are positive, the
 *               prices array has the same number of items
 *               as the weights array, and the weights are integers
 *
 * Post condition: The return value is the max value with the
 *               capacity allotted.
 */
public static int dynamic(int capacity, int[] prices, int[]
    weight, int numItems) {
    //checking preconditions
    assert (numItems > 0);
    assert (prices.length == weight.length);
    for (int i = 0; i < numItems; i++){
        assert (prices[i] >= 0);
        assert (weight[i] >= 0);
    }

    int [][] f = new int[numItems + 1][capacity + 1];
    boolean [][] knapsack = new boolean[numItems + 1][capacity
        + 1];

    //Build table k[][] in bottom up manner
    for (int i = 1; i <= numItems; i++) {
        for (int w = 1; w <= capacity; w++) {
            //don't take the item
            int f1 = f[i - 1][w];

            //take it
            int f2 = Integer.MIN_VALUE;
            if (weight[i] <= w) {
                f2 = prices[i] + f[i - 1][w - weight[i]];
            }

            //select the better of two options
            f[i][w] = Math.max(f1, f2);
            knapsack[i][w] = (f2 > f1);
        }
    }
}

```

```

    }

    //determine which items to take
    boolean[] take = new boolean[numItems + 1];
    for (int n = numItems, w = capacity; n > 0; n--) {
        if (knapsack[n][w]) {
            take[n] = true;
            w = w - weight[n];
        } else
            take[n] = false;
    }

    //          //print results
    //          System.out.println("item" + "\t" + "profit" + "\t" +
    "weight" + "\t" + "take");
    //          for (int n = 1; n < numItems+1; n++)
    //          System.out.println(n + "\t" + prices[n] + "\t" +
    weight[n] + "\t" + take[n]);

    //finds the max value of the most optimal knapsack the
    thief can fill
    for(int i = 0; i < take.length; i++){
        if(take[i]) {
            maxVal += prices[i];
        }
    }
    return maxVal;
}

//used to time how long it takes to find the solution using
dynamic programming
//returns the value of the knapsack stolen
public long TimeToFind(int capacity, int[] prices, int[]
weight, int numItems){
    long start = System.nanoTime();
    int value = dynamic(capacity, prices, weight, numItems);
    long end = System.nanoTime();
    long duration = end - start;
    System.out.println("Value_for_dynamic_programming:_ " +
        value);
    return duration;
}
}

```

REFERENCES

- [1] Cormen, Thomas H. Introduction to algorithms. MIT press, 2009.
- [2] Rouse, Margaret. "What Is Brute Force Cracking? - Definition from WhatIs.com." SearchSecurity. July 2006. Accessed March 07, 2016. <http://searchsecurity.techtarget.com/definition/brute-force-cracking>.