

0/1 Knapsack

Preston Stosur-Bassett

Abstract

The present study investigates two implementations for a 0/1 Knapsack problem, a greedy solution, and a dynamic solution. The dynamic solution will always result in the correct answer but takes $\theta n * m$ time to complete. The greedy solution will generally result in a reasonable answer, and even sometimes the correct answer, and take θn time to complete. Which solution to use will depend on what is most important to the problem at hand, speed, or accuracy.

I. MOTIVATION

A 0/1 Knapsack problem is most often described as how does a thief steal the most valuable items with the amount of space he has available to carry them. However, the applications of a 0/1 Knapsack far beyond just this. When solved, a 0/1 Knapsack problem will render the optimal solution, whether it be how one should construct their schedule so they can get the most done in one day, or how to pack a truck such that the fewest amount of trips are taken.

II. BACKGROUND

Unfortunately, solving a 0/1 Knapsack problem often times proves difficult because in order to find the best solution, all possible solutions must be attempted, a method known as brute force. It is not practical to brute force a problem of this kind because it would take $\theta n!$ to solve, which is unacceptable for large sets of numbers. However, there are other methods beside brute force that allow for the correct or 'good-enough' solution to be found. One of these methods is known as dynamic programming, which trades off speed for space. With dynamic programming the correct solution can be found in $\theta n * m$ time, where n is the amount of items and m is the size of the knapsack. Another method for solving this type of problem is called a greedy algorithm, which works by making the best decision on which item to take locally. A greedy solution to the 0/1 Knapsack problem traditionally runs in θn time. Unfortunately a greedy solution is not always correct and therefore highly unreliable.

III. PROCEDURE

A multidimensional array insertion sort can be implemented in a multitude of languages using the pseudocode provided in Algorithm 1.

Insertion Sort Pre-Condition: A is an unsorted non-empty array of non-empty arrays containing a comparable data type with a natural order such that v is an index value of the inner array.

Insertion Sort Post-Condition: A' is a permutation of A that is in strictly non-increasing order.

Insertion Sort Outer-Loop Invariant: The subarray $A'[1 \dots i - 1]$ contains all the same elements as the subarray $A[1 \dots i - 1]$.

Insertion Sort Outer-Loop Initialization: The outer-loop invariant holds because $A'[1 \dots i - 1]$ and $A[1 \dots i - 1]$ both contain the same one element.

Insertion Sort Outer-Loop Maintenance: The outer-loop invariant holds because $A'[1 \dots i - 1]$ and $A[1 \dots i - 1]$ both contain the same elements, although they may be in different orders.

Insertion Sort Outer-Loop Termination: When the outer-loop terminates, $i = A.length$, which implies that the entire array has been traversed and the guard has been negated. The negation of the guard implies that $A'[1 \dots i - 1]$ contains all the elements in $A[1 \dots i - 1]$.

Insertion Sort Inner-Loop Invariant: $A'[1 \dots j]$ is sorted in strictly non-decreasing order.

Algorithm 1 INSERTION-SORT(A, v)

```

1: procedure INSERTION-SORT( $A, v$ )
2:   if  $A.length < 2$  then
3:     return  $A$ 
4:   end if
5:    $i = 2$ 
6:   while  $i$  upto  $A.length$  do
7:      $key = A[i][v]$ 
8:      $a = A[i]$ 
9:      $j = i - 2$ 
10:    while  $j$  downto 1 and  $key > A[j][v]$  do
11:       $A[j + 1] = A[j]$ 
12:       $j = j - 1$ 
13:    end while
14:     $A[j + 1] = a$ 
15:     $i = i + 1$ 
16:  end while
17:  return  $A$ 
18: end procedure

```

Insertion Sort Inner-Loop Initialization: Before the first iteration of the loop, $j = 1$, meaning the subarray $A'[1 \dots j]$ contains exactly one element, which is already sorted.

Insertion Sort Inner-Loop Maintenance: At the beginning of each iteration of the loop the inner-loop invariant holds because j counts down from i , and $A'[j+1]$ is swapped with $A'[j]$ only if $A'[j+1]$ is less than $A[j]$.

Insertion Sort Inner-Loop Termination: The negation of the guard implies that $j = A.length$ and that $A'[1 \dots j]$ has been entirely traversed and sorted in strictly non-decreasing order, which maintains the inner-loop invariant.

Insertion Sort Conclusion: The termination of both the inner and outer loops implies that the entire array has been traversed, A' is a permutation of A containing all the same elements in strictly non-decreasing order. This satisfies the post condition.

A greedy solution to a 0/1 knapsack problem can be implemented in a variety of languages using the pseudocode in Algorithm 2.

Greedy Knapsack Pre-Condition: *Weights* and *Prices* both have in them n number of elements

Greedy Knapsack Post-Condition: The returned value will be a reasonable solution for the largest value of price combinations such that the aggregate of the corresponding weights does not exceed knapsack capacity c .

Greedy Solution First-Loop Invariant: $ratio[1 \dots v-1]$ has the same number of non-null elements in it as both $Prices[1 \dots v-1]$ and $Weights[1 \dots v-1]$.

Greedy Solution First-Loop Invariant Initialization: The invariant holds true before the first iteration of the loop because $v = 1$, and $ratio[1 \dots v-1]$, $Prices[1 \dots v-1]$ and $Weights[0 \dots v-1]$ all do not exist, and is therefore vacuously true.

Greedy Solution First-Loop Invariant Maintenance: At the beginning of each iteration of the loop the invariant holds true because each time the loop runs, v is incremented by 1, and exactly 1 element is added to $ratio$, therefore $ratio[0 \dots v-1]$ will have the exact same number of elements as

Algorithm 2 GREEDY-KNAPSACK(n , $Weights$, $Prices$, c)

```

1: procedure GREEDY-KNAPSACK( $n$ ,  $Weights$ ,  $Prices$ ,  $c$ )
2:   if  $n == 1$  and  $Weights[1] \leq c$  then
3:     return  $Prices[1]$ 
4:   end if
5:   if  $n \leq 0$  then
6:     return 0
7:   end if
8:    $profit = 0$ 
9:    $ratio = newArray$ 
10:  for  $v = 1$  upto  $n$  do
11:     $a = Prices[v] / Weights[v]$ 
12:     $ratio[v] = a$ 
13:  end for
14:   $ratio = Insertion - Sort(ratio, 1)$ 
15:   $i = 1$ 
16:  while  $c > 0$  and  $i < n$  do
17:    if  $c - ratio[i][3] \geq 0$  then
18:       $profit = profit + ratio[i][2]$ 
19:       $c = c - ratio[i][3]$ 
20:    end if
21:     $i = i + 1$ 
22:  end while
23:  return  $profit$ 
24: end procedure

```

both $Prices[1 \dots v-1]$ and $Weights[1 \dots v-1]$.

Greedy Solution First-Loop Invariant Termination: After the termination of the loop the invariant holds true because v has been incremented by 1 during each iteration of the loop, and $ratio$ has gained 1 element during each iteration of the loop, therefore $ratio[1 \dots v-1]$ contains the exact same number of elements as both $Prices[1 \dots v-1]$ and $Weights[1 \dots v-1]$.

Greedy Solution Second-Loop Invariant: The aggregate of the weights corresponding to the prices included in $profit$ are less than or equal to c .

Greedy Solution Second-Loop Invariant Initialization: Before the first iteration of the loop the invariant holds true vacuously because no prices have been included in $profit$.

Greedy Solution Second-Loop Invariant Maintenance: At the beginning of each iteration of the loop the invariant holds true because only items whose weight is less than or equal to the size of c are added to $profit$.

Greedy Solution Second-Loop Invariant Termination: After the termination of the loop, the invariant holds true because only items whose weight was less than or equal to the value of c have been added to $profit$.

A dynamic solution to 0/1 knapsack can be implemented in a variety of languages using the pseudocode provided in Algorithm 3.

Dynamic Knapsack Pre-Condition: $Weights$ and $Prices$ both have n number of elements

Dynamic Knapsack Post-Condition: The returned value will be the correct solution for the

largest value of price combinations such that the aggregate of the corresponding weights does not exceed knapsack capacity c .

Algorithm 3 DYNAMIC-KNAPSACK(n , $Weights$, $Prices$, c)

```

1: procedure DYNAMIC-KNAPSACK( $n$ ,  $Weights$ ,  $Prices$ ,  $c$ )
2:   if  $n == 1$  and  $Weights[1] \leq c$  then
3:     return  $Prices[1]$ 
4:   end if
5:   if  $n \leq 0$  then
6:     return 0
7:   end if
8:    $tab[n][c] = newNestedArray$ 
9:   for  $x = 1$  upto  $n$  do
10:    for  $y = 1$  upto  $c$  do
11:       $tab[x][y] = 0$ 
12:    end for
13:  end for
14:  for  $i = 1$  upto  $n$  do
15:    for  $j = 0$  upto  $c$  do
16:      if  $Weights[i] \leq j$  and  $Prices[i] + tab[i][j - Weights[i]] > tab[i][j]$  then
17:         $tab[i + 1][j] = Price[i] + tab[i][j - weights[i]]$ 
18:      end if
19:      if  $!(Weights[i] \leq j)$  and  $!(Prices[i] + tab[i][j - Weights[i]] > tab[i][j])$  then
20:         $tab[i + 1][j] = tab[i][j]$ 
21:      end if
22:    end for
23:  end for
24:  return  $tab[n][c]$ 
25: end procedure

```

Dynamic Solution First Outer-Loop Invariant: $tab[1 \dots x-1]$ has the same number of non-null elements in it as both $Weights[1 \dots x-1]$ and $Prices[1 \dots x-1]$.

Dynamic Solution First Outer-Loop Invariant Initialization: Before the first iteration of the loop the invariant holds true vacuously because $tab[0 \dots x-1]$, $weights[0 \dots x-1]$ and $prices[0 \dots x-1]$ all do not exist.

Dynamic Solution First Outer-Loop Invariant Maintenance: At the beginning of each iteration because x is incremented by one at the same rate that one element is added to tab .

Dynamic Solution First Outer-Loop Invariant Termination: At the termination of the loop the invariant holds true because x has been incremented at the same rate that an element has been added to tab .

Dynamic Solution First Inner-Loop Invariant: $tab[x]$ has y number of non-null elements in it.

Dynamic Solution First Inner-Loop Invariant Initialization: Before the first iteration of the loop the invariant holds true because $y = 0$ and $tab[x]$ has 0 elements in it.

Dynamic Solution First Inner-Loop Invariant Maintenance: At the beginning of each iteration of the loop the invariant holds true because because y is incremented at the same rate that elements are added to $tab[x]$.

Dynamic Solution First Inner-Loop Invariant Termination: After the termination of the loop the invariant holds true because elements have been added to $tab[x]$ at the same rate in which y has

been incremented.

Dynamic Solution Second Outer-Loop Invariant: $\text{tab}[i][j]$ is larger than or equal to $\text{tab}[i-1][j]$
Dynamic Solution Second Outer-Loop Invariant Initialization: Before the first iteration of the loop the invariant holds true vacuously because $\text{tab}[-1]$ does not exist.
Dynamic Solution Second Outer-Loop Invariant Maintenance: At the beginning of each iteration of the loop the invariant holds true because during the loop the values are incremented, but they are never decreased.
Dynamic Solution Second Outer-Loop Invariant Termination: After the loop has terminated the invariant holds true because the values in tab have never been decreased, only increased, and therefore $\text{tab}[i][j]$ will be greater than or equal to $\text{tab}[i-1][j]$

Dynamic Solution Second Inner-Loop Invariant: $\text{tab}[i][j]$ is always equal to or greater than $\text{tab}[i][j-1]$
Dynamic Solution Second Inner-Loop Invariant Initialization: Before the first iteration of the loop the invariant holds true vacuously because $\text{tab}[i][j-1]$ does not exist.
Dynamic Solution Second Inner-Loop Invariant Maintenance: At the beginning of each iteration of the loop the invariant holds true because the values in tab have never been decreased, only increased, therefore $\text{tab}[i][j]$ is going to be larger than or equal to $\text{tab}[i][j-1]$.
Dynamic Solution Second Inner-Loop Invariant Termination: After the termination of the loop the invariant holds true because the values in tab have never been decremented, only incremented.

IV. TESTING

A. Testing Plan and Results

All arrays used in testing are Java `ArrayList<Integer>` except for tab in the dynamic solution, which is a primitive Java nested array. All times are recorded in milliseconds using a stopwatch class borrowed from Robert Sedgwick and Kevin Wayne [?] It is important to note that the stopwatch class used takes the elapsed real-time between the start of the call to the knapsack solution and the end of that call as opposed to taking the elapse processor-time because these tests were run on a multi-core computer. The size of elements used in testing are as follows: For $n < 100$, $c = 50$. For $n \geq 100$, $c = 2n$ where n represents the number of items, and c represents the capacity of the knapsack.

Table I
GREEDY SOLUTION REAL-TIME RUN-TIME

Number of Elements	Size of Knapsack	Time
0	50	0.0
1	50	0.0
10	50	0.0006
100	200	0.0043
1000	2000	0.0231
10000	20000	0.252
100000	200000	59.1176

Table II
DYNAMIC SOLUTION REAL-TIME RUN-TIME

Number of Elements	Size of Knapsack	Time
0	50	0.0001
1	50	0.0
10	50	0.0002
100	200	0.0119
1000	2000	0.0356
10000	20000	1.9243

The percent error of greedy solutions is shown in Table 3. Percent error was calculated by the percent error formula: $\frac{|GreedySolution - DynamicSolution|}{DynamicSolution} * 100$. The percentages shown in Table 3 are average percent errors taken from trials of 10.

Table III
DYNAMIC VS GREEDY SOLUTIONS ACCURACY COMPARISON

Number of Elements	Size of Knapsack	Percent Error
0	50	0%
1	50	0%
10	50	5.3%
100	200	0.9%
1000	2000	0.032887%
10000	20000	0.00555725%

B. Problems Encountered

One major issue encountered during the development of the dynamic programming solution was that when the second pair of loops would make calls to any of ArrayLists (including *tab*, which at the time was also an ArrayList implementation) there would be an out of bounds error because of a negative index value. The out of bounds error was emanating from the conditional in the second nested loop. The cause for this error is still unknown. To aid the issue *tab* was changed from a nested ArrayList to a primitive nested Java array.

Another issue encountered during the development of this program was executing Java *assert* statements. When assertions were wrapped in parenthesis, they evaluated to a boolean, and the assert would throw an *AssertionError* with the explanation 'cannot compare to boolean'. Removing the parenthesis wrapping the assertion fixed this issue.

When the dynamic solution was executed with values of anything above $n = 10000$ and $c = 20000$ Java gave a "Out Of Memory Error: Java heap space". This can be attributed the extreme number of elements being stored in *tab*. The computer's system monitor reported RAM usage at 95%. Unfortunately, this issue cannot be aided.

The last issue encountered during the development process of this program was generic programming. The program would not compile because not all Objects have a *compareTo* method. Adding the statement `< T extends Comparable < T >>` fixed this issue. [?]

V. EXPERIMENTAL ANALYSIS

The insertion sort, greedy solution and dynamic solution demonstrated in Algorithm 1, 2 and 3 was implemented in Java and executed on an HP SpectreXT TouchSmart with a 4 core Intel i7 Ivybridge processor clocked at 1.9GHz running Ubuntu Gnome 14.10 64-bit with 8GB of DDR3 RAM. The expected growth of insertion sort as the number of elements (n) grows large can be represented as θn^2 where θ represents the asymptotically tightly bound running time. [?]

The greedy solution also follows a similar θn^2 run time, see Figure 1. This is because an insertion sort was chosen to implement as the sorting algorithm used in the greedy solution. An insertion sort was chosen for the sorting algorithm because, while it is not the fastest, requires less axillary space.

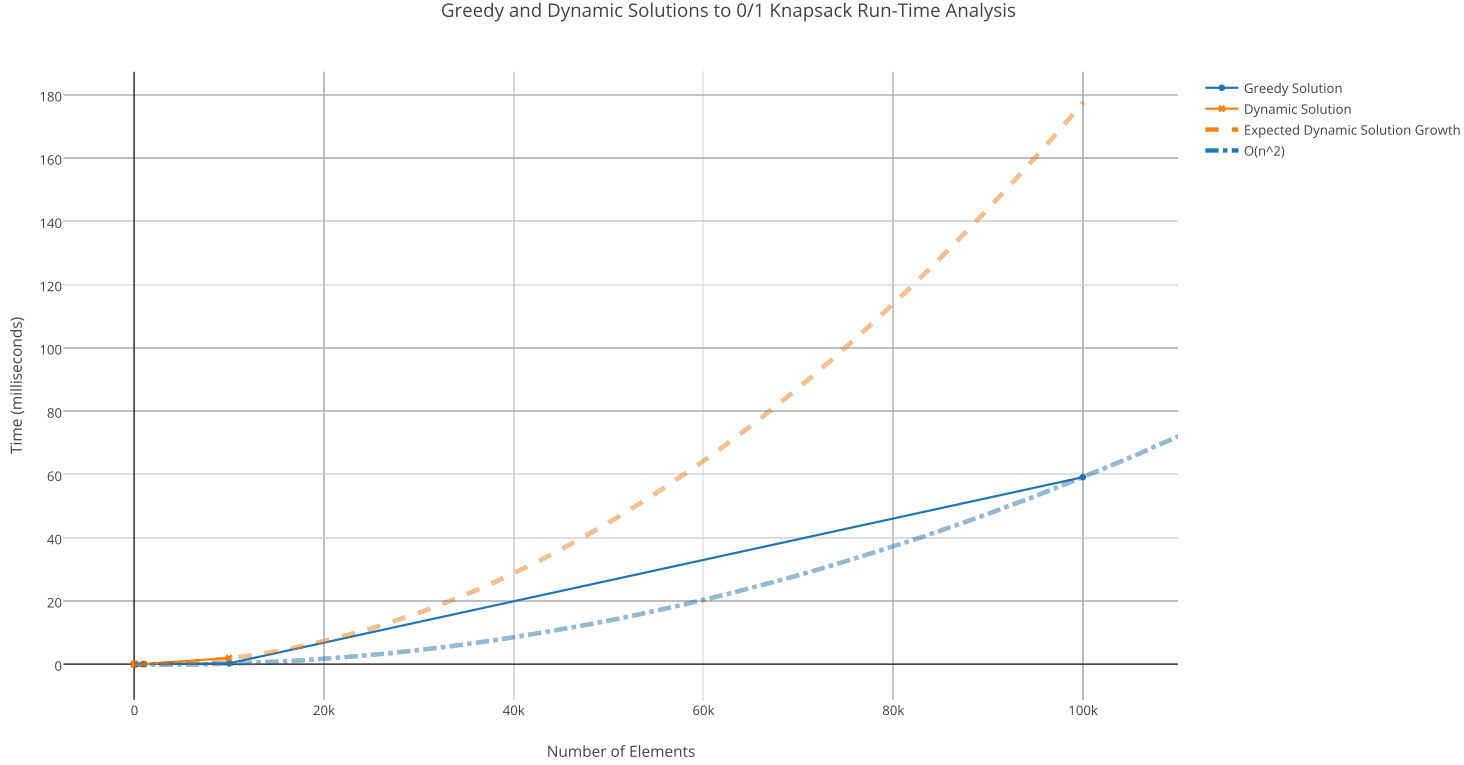


Figure 1. Java Implementation of Greedy and Dynamic Solutions to 0/1 Knapsack Run-Time Analysis

While run time is an important factor, the space required to run the algorithm is also very important, and should not be neglected. As seen in Table 2, the dynamic solution unavoidably required too much axillary space and Java gave an error when values of n became increasingly large. It also important to note that the data graphed in Figure 1 was executed while the machine executed superfluous commands such as checking for system updates. To aid this issue, averages of 10 trails were taken and used in Figure 1.

The run time for the dynamic solution as seen in figure one $\theta n * c$ where n is the number of elements and c is the size of the knapsack. In this implementation for values of n greater than or equal to 100, a value of $2n$ was used for c . That would make the run time of the specific implementation of the dynamic solution $2n^2$ but after dropping constants becomes θn^2 . In Figure 1, the reason that the dynamic solution takes so much more time to complete is because there is a constant of 2 for n^2 which can be attributed to doubling the real time it took to return the results. Because the dynamic solution works by storing answers originally found in axillary space, the values of n cannot be very high without causing a Java error, as seen in Table 2.

In Table 3 there looks to be a negative correlation between the size of n and the percent error. This negative correlation can be attributed to the fact that the percent error formula takes the difference between the two results and divides it by the correct result. For example, if the result for a greedy solution was 2 and the result for the dynamic solution was 3 the formula would read $\frac{2}{3} * 100$ rendering a 66.66% error, however the difference between the two is only 2. If the values were 101 and 103 the percent error formula would read $\frac{2}{103} * 100$ and renders 1.9% error. The second value is substantially lower, even though the difference between the two numbers remains the same.

VI. CONCLUSION

Although in this specific implementation, both the dynamic solution and the greedy solution had a run-time of θn^2 , it can be possible for the dynamic solution to have a run-time of $\theta n * c$ and the greedy solution to have a run-time of θn . The best algorithm to use for a given problem is really a subjective decision and depends on two things, 1. how accurate does the answer to the problem really need to be, and 2. what is the time requirement to find out what the answer is.

REFERENCES

- [1] R. Sedgewick and K. Wayne, “Stopwatch,” Java Class.
- [2] E. Hartig, “Generic programming,” Conversation, January 2015.
- [3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, Massachusetts: The MIT Press, 2009.

APPENDIX

Listing 1
DRIVER

```

/*
 * @Author: Preston Stosur-Bassett
 * @Date: 3, 3, 2015
 * @Class: Driver
 * @Description: This class will serve as a driver function for our
   Knapsack class
 */

import java.util.ArrayList;

public class Driver {
    public static void main(String args[]) {
        Knapsack theif = new Knapsack();
        DummyData testData = new DummyData();
        Stopwatch watchman = new Stopwatch();

        ArrayList<Integer> prices = new ArrayList<Integer>();
        ArrayList<Integer> weights = new ArrayList<Integer>();

        if (args[0].equals("greedy") == true && args[1] != null && args[2] !=
            null) {
            System.out.println("Running greedy algorithm...");

            int numberOfElements = Integer.parseInt(args[1]);
            int knapsackSize = Integer.parseInt(args[2]);

            System.out.println("Max Knapsack Capacity: "+knapsackSize);

            prices = testData.runArrayList(numberOfElements, 1, 1000, prices);
            weights = testData.runArrayList(numberOfElements, 1, 50, weights);

            System.out.println("Set P:" + prices);
            System.out.println("Set W:" + weights);
            System.out.println("");

            watchman.startTime();
            int totalProfit = theif.greedyKnapsack(numberOfElements, weights,
                prices, knapsackSize);
            double elapsedTime = watchman.elapsedTime();

            Integer totalProfitObject = new Integer(totalProfit);
            System.out.println("The total profit according to this greedy
                algorithm is: "+totalProfitObject);
            System.out.println("Time to Complete: "+elapsedTime);
        }
    }
}

```

```

else if (args[0].equals("dynamic") == true && args[1] != null && args
    [2] != null) {
    System.out.println("Running dynamic algorithm...");

    int numberOfElements = Integer.parseInt(args[1]);
    int knapsackSize = Integer.parseInt(args[2]);

    System.out.println("Max Knapsack Capacity: "+knapsackSize);

    prices = testData.runArrayList(numberOfElements, 1, 1000, prices);
    weights = testData.runArrayList(numberOfElements, 1, 50, weights);

    System.out.println("Set P:"+prices);
    System.out.println("Set W:"+weights);
    System.out.println("");

    watchman.startTime();
    int totalProfit = theif.dynamicKnapsack(numberOfElements, weights,
        prices, knapsackSize);
    double elapsedTime = watchman.elapsedTime();

    Integer totalProfitObject = new Integer(totalProfit);
    System.out.println("The total profit according to this dynamic
        algorithm is: "+totalProfitObject);
    System.out.println("Time to Complete: "+elapsedTime);
}
else if (args[0].equals("compare") == true && args[1] != null && args
    [2] != null) {
    System.out.println("Comparing greedy solution to dynamic solution
        ...");

    Stopwatch dynamicWatch = new Stopwatch();
    Stopwatch greedyWatch = new Stopwatch();

    int numberOfElements = Integer.parseInt(args[1]);
    int knapsackSize = Integer.parseInt(args[2]);

    if (numberOfElements <= 0) {
        System.out.println("Cannot have 0 or negative number for number
            of elements. Changing to 1.");
        numberOfElements = 1;
    }

    System.out.println("Max Knapsack Capacity: "+knapsackSize);

    prices = testData.runArrayList(numberOfElements, 1, 1000, prices);
    weights = testData.runArrayList(numberOfElements, 1, 50, weights);

    System.out.println("Set P:"+prices);
    System.out.println("Set W:"+weights);

```

```

        System.out.println("");

        greedyWatch.startTime();
        int greedyProfit = theif.greedyKnapsack(numberOfElements, weights,
            prices, knapsackSize);
        double greedyTime = greedyWatch.elapsedTime();

        System.out.println("Greedy Solution: $" + greedyProfit + ". Completed
            in " + greedyTime + " milliseconds.");

        dynamicWatch.startTime();
        int dynamicProfit = theif.dynamicKnapsack(numberOfElements, weights
            , prices, knapsackSize);
        double dynamicTime = dynamicWatch.elapsedTime();

        System.out.println("Dynamic Solution: $" + dynamicProfit + ". Completed
            in " + dynamicTime + " milliseconds.");

        System.out.println("");

        int difference = greedyProfit - dynamicProfit;
        int top = Math.abs(difference);
        int bottom = Math.abs(dynamicProfit);
        double error = new Integer(top).doubleValue() / new Integer(bottom)
            .doubleValue();
        double percentError = error * 100;

        System.out.println("Percent Error: " + percentError + "%");

    }
    else {
        System.out.println("Invalid argument error!!");
        System.out.println("The correct format is: Driver [method] [number
            of elements] [size of knapsack]");
    }
}
}
}

```

Listing 2
KNAPSACK

```

/*
 * @Author: Preston Stosur-Bassett
 * @Description: This class implements a greedy and dynamic solution for
 *               the 0/1 Knapsack problem. These two solutions will return the max
 *               value that can be obtained only, and not how to obtain them.
 * @Class: Knapsack
 * @Date: 3, 3, 2015
 */

import java.util.ArrayList;

```

```

import java.lang.Math;
import java.util.List;

public class Knapsack<T extends Comparable<T>> {
    /*
    * @Pre-Condition: <code>weights</code> and <code>prices</code> both
    *   have in them <code>elems</code> amount of elements
    * @Post-Condition: The returned value will be a reasonable solution for
    *   the largest value in price where the aggregate of the corresponding
    *   weight does not exceed the <code>backpackSize</code>
    * @Description: greedyKnapsack implements a greedy algorithm to find a
    *   reasonable solution for a 0/1 Knapsack problem
    * @param int elems is the amount of elements in both <code>ArrayList<
    *   Integer> weights</code> and <code>ArrayList<Integer> prices</code>
    * @param ArrayList<Integer> weights is a non-empty ArrayList of Integer
    *   objects with exactly <code>elems</code> amount of elements in it
    *   and contains absolutely no zeros
    * @param ArrayList<Integer> prices is a non-empty ArrayList of Integer
    *   object with exactly <code>elems</code> amount of elements in it
    * @param int backpackSize is the maximum value of weights that the
    *   knapsack can hold
    * @return int profit is a reasonable solution to the given 0/1 Knapsack
    *   problem
    */
    //INVARIANT (First Loop): ratioListings[0 ... v-1] has the same number
    *   of non-null elements in it as both prices[0 ... v-1] and weights[0
    *   ... v-1]
    //INVARIANT (Second Loop): The aggregate of the weights corresponding
    *   to the prices included in the value of profit are less than or equal
    *   to the value of backpackSize
    public int greedyKnapsack(int elems, ArrayList<Integer> weights,
        ArrayList<Integer> prices, int backpackSize) {
        Debug debugger = new Debug();
        int returnValue;
        if (elems == 1 && weights.get(0).intValue() <= backpackSize) {
            returnValue = prices.get(0).intValue();
        }
        else if (elems < 1) {
            returnValue = 0;
        }
        else {
            int profit = 0;

            ArrayList<ArrayList<Integer>> ratioListings = new ArrayList<
                ArrayList<Integer>>(elems);

            int v = 0;
            /*INITIALIZATION (First Loop): Our invariant holds before the first
            *   iteration of the loop because v = 0, and ratioListings[v-1],
            *   prices[v-1] and weights[v-1] all do not exist, and therefore is

```

```

    vacuously true.*/

while(v < elems) {
    /*MAINTANENCE (First-Loop): At the beginning of each iteration of
       the loop our invariant holds true because each time the loop
       runs, v is increased by one and one element is added to
       raioListings, therefore ratioListings[0 ... v-1] will have
       exactly the same number of elements as both prices[0 ... v-1]
       and weights[0 ... v-1].*/
    if(v > 0) {
        debugger.assertEquals(new Integer(weights.subList(0, v-1).size
            ()), new Integer(prices.subList(0, v-1).size()));
        debugger.assertEquals(new Integer(prices.subList(0, v-1).size())
            , new Integer(ratioListings.subList(0, v-1).size()));
    }

    int ratio = prices.get(v).intValue() / weights.get(v).intValue();
    ArrayList<Integer> innerRatioListing = new ArrayList<Integer>(3);
    innerRatioListing.add(new Integer(ratio));
    innerRatioListing.add(new Integer(prices.get(v).intValue()));
    innerRatioListing.add(new Integer(weights.get(v).intValue()));
    ratioListings.add(innerRatioListing);

    v++;
}
/*TERMINATION (First Loop): After the loop terminates, our
   invariant holds true because v has been increased by one during
   each iteration of the loop, and ratioListings has gained one
   element during each iteration of the loop, so therefore
   ratioListings[0 ... v-1] contains exactly the same number of
   elements as both prices[0 ... v-1] and weights[0 ... v-1].*/
debugger.assertEquals(new Integer(weights.subList(0, v-1).size()),
    new Integer(prices.subList(0, v-1).size()));
debugger.assertEquals(new Integer(prices.subList(0, v-1).size()),
    new Integer(ratioListings.subList(0, v-1).size()));

Sort sorter = new Sort();

ratioListings = sorter.insertionSortNestedArray(ratioListings, 0);

int i = 0;
int totalWeight = 0;
/*INITIALIZATION (Second Loop): Before the first iteration of the
   loop our invariant holds true because no prices are included in
   the value of profit, and therefore, vacuously true.*/
debugger.assertLessEquals(totalWeight, backpackSize);

while(backpackSize > 0 && i < elems) {
    /*MAINTENANCE (Second Loop): At the beginning of each iteration
       of the loop our invariant holds true because only items whose

```

```

        weight is less than or equal to the remaining space in the
        knapsack (backpackSize) are added to the profit.*/
        debugger.assertLessEquals(totalWeight, backpackSize);

        if (backpackSize - ratioListings.get(i).get(2).intValue() >= 0) {
            profit = profit + ratioListings.get(i).get(1).intValue();
            backpackSize = backpackSize - ratioListings.get(i).get(2).
                intValue();
            totalWeight = totalWeight + ratioListings.get(i).get(2).
                intValue();
        }

        i++;
    }
    /*TERMINATION (Second Loop): After the loop terminates, our
       invariant holds true because only items whose weight was less
       than or equal to the remaining size in the knapsack (
       backpackSize) were added to the profit.*/
    debugger.assertLessEquals(totalWeight, backpackSize);

    returnValue = profit;
}

return returnValue;
}

/*
 * @Pre-Condition: <code>weights</code> and <code>prices</code> both
   have in them <code>elems</code> amount of elements
 * @Post-Condition: The returned value will be the correct solution for
   the largest value in price where the aggregate of the corresponding
   weights do not excede the <code>backpackSize</code>
 * @Description: dynamicKnapsack implements a dynamic programming
   solution to find the correct answer for a 0/1 Knapsack problem
 * @param int elems is the amount of elemnts in both <code>ArrayList<
   Integer> weights</code> and <code>ArrayList<Integer> prices</code>
 * @param ArrayList<Integer> weights is a non-empty ArrayList of Integer
   objects with esactly <code>elems</code> amount of elements in it
   and contains absolutely no zeros
 * @param ArrayList<Integer> prices is a non-empty ArrayList of Integer
   objects with exactly <code>elemes</code> amount of elemnts in it
 * @param int backpackSize is the maximum value of the weights that the
   knapsack can hold
 * @return int returnValue is the correct value of the maximum amount of
   values you can get from prices where their correpsponding weights
   do not excede the backpackSize
 */
//INVARIANT (First Outer-Loop): tab[0 ... x-1] has the same number of
non-null elements in it as both weights[0 ... x-1] and prices[0 ...
x-1]

```

```

//INVARIANT (First Inner-Loop): tab[x] has y number of non-null
    elements in it.
//INVARIANT (Second Outer-Loop): tab[i][] is larger than or equal to
    tab[i-1][].
//INVARIANT (Second Inner-Loop): tab[i][j] is always equal to or
    greater than tab[i][j-1]
public int dynamicKnapsack(int elems, ArrayList<Integer> weights,
    ArrayList<Integer> prices, int backpackSize) {
    Debug debugger = new Debug();
    int returnValue = 0;
    backpackSize = backpackSize + 1;
    if (elems == 1 && weights.get(0).intValue() <= backpackSize) {
        returnValue = prices.get(0).intValue();
    }
    else if (elems <= 0) {
        returnValue = 0;
    }
    else {
        int [][] tab = new int [elems][backpackSize];

        int x = 0;
        int y = 0;
        int nonNullElements = 0;
        /*INITIALIZATION (First Outer-Loop): Before the first iteration of
            the loop the invariant holds vacuously because tab[0 ... x-1],
            weights[0 ... x-1] and prices[0 ... x-1] all do not exist.*/
        while (x < elems) {
            /*MAINTENANCE (First Outer-Loop): At the beginning of each
                iteration of the loop the invariant holds because x is
                incremented by one at the same rate that one element is added
                to tab [].*/
            if (x > 0) {
                int [][] subTabM = Array.copyOfRange(tab, 0, x-1);
                debugger.assertEquals(new Integer(weights.subList(0, x-1).
                    intValue()), new Integer(prices.subList(0, x-1).intValue()))
                    ;
                debugger.assertEquals(new Integer(prices.subList(0, x-1).
                    intValue()), new Integer(subTabM.length));
            }

            /*INITIALIZATION (First Inner-Loop): Before the first iteration
                of the loop the invariant holds true because y = 0 and tab[x]
                has 0 elements in it. */
            nonNullElements = 0;
            for (int i = 0; i < tab[x].length; i++) {
                if (tab[x][y] != null) {
                    nonNullElements++;
                }
            }
            debugger.assertEquals(nonNullElements, y);

```



```

while(y < backpackSize) {
    /*MAINTENANCE (First Inner-Loop): At the beginning of each
       iteration of the loop the invariant holds true because at
       the same rate y is incremented an element is added to tab[x]
    */
    nonNullElements = 0;
    for(int i = 0; i < tab[x].length; i++) {
        if(tab[x][y] != null) {
            nonNullElements++;
        }
    }
    debugger.assertEquals(nonNullElements, y);

    tab[x][y] = 0;
    y++;
}
/*TERMINATION (First Outer-Loop): At the termination of the loop
   the invariant holds true because y has been incremented at the
   same rate elements have been added to tab[x]*/
nonNullElements = 0;
for(int z = 0; z < tab[x].length; z++) {
    if(tab[h][x] != null) {
        nonNullElements++;
    }
}
debugger.assertEquals(nonNullElements, z);

x++;
}
/*TERMINATION (First Inner-Loop): At the termination of the loop
   the invariant holds true because x has been incremented at the
   same rate that elements have been added to tab[]. */
debugger.assertEquals(new Integer(weights.subList(0, x-1).intValue(
    )), new Integer(prices.subList(0, x-1).intValue()));
debugger.assertEquals(new Integer(prices.subList(0, x-1).intValue(
    )), new Integer(subTabM.length));

int i = 0;
int j = 0;
/*INITIALIZATION Second Outer-Loop: Before the first iteration of
   the loop our invariant holds vacuously true because tab[i-1]
   does not exist.*/
while(i < elems - 1) {
    /*MAINTENANCE Second Outer-Loop: At the beginning of each
       iteration of the for loop the invariant holds true because
       during the loop tab can only be increased, never decreased.*/
    if(i > 0) {
        debugger.assertGreatEquals(tab[i][j], tab[i-1][j]);
    }
}

```

```

/*INITIALIZATION Second Inner-Loop: Before the first iteration of
the for loop the invariant holds vacuously because tab[i][j
-1] does not exist.*/
while(j < backpackSize) {
/*MAINTENANCE Second Inner-Loop: AT the beginning of each
iteration of hte for loop the invariant holds true because
during the loop tab can only be increased , never decreased
.*/
if(j > 0){
    debugger.assertGreatEquals(tab[i][j], tab[i][j-1]);
}
if(weights.get(i).intValue() <= j && prices.get(i).intValue() +
    tab[i][j - weights.get(i).intValue()] > tab[i][j]) {
    tab[i+1][j] = prices.get(i).intValue() + tab[i][j - weights.
        get(i).intValue()];
}
else {
    tab[i+1][j] = tab[i][j];
}

    j++;
}
/*TERMINATION Second Inner-Loop: After the termination of the
loop the invariant holds true because the values in tab have
only ever been increased and never decreased.*/
if(j > 0) {
    debugger.assertGreatEquals(tab[i][j], tab[i][j-1]);
}

    i++;
}
/*TERMINATION Second Outer-Loop: At the termination of the loop the
invariant holds true because the values of tab have only been
increased , but never decreased.*/
if(i > 0) {
    debugger.assertGreatEquals(tab[i][j], tab[i-1][j]);
}

returnValue = tab[elems-1][backpackSize-1];
}

return returnValue;
}
}

```

Listing 3
SORT

```

/*
 * @Author: Preston Stosur-Bassett
 * @Date: Jan 24, 2015

```

```

*      @Class: Sort
*      @Description: This class will contain many methods that will sort
*                    generic data types using common sorting algorithms.
*/

import java.util.ArrayList;
import java.util.List;

public class Sort<T extends Comparable<T>> {
    /*
    *      @Pre-Condition: ArrayList<T> is a non-empty set of data
    *                    where T is a comparable data type with a natural order
    *      @Post-Condition: Each parent node is more extreme than
    *                    its child node.
    *      @Description: heapify is a helper method for heapSort
    *                    that keeps the heap in order so that the root node is the most
    *                    extreme element in the heap.
    *      @param ArrayList<T> unsorted is a non-empty set of data
    *                    where T is a comparable data type with a natural order
    *      @param int i
    *      @param int total
    *      @return ArrayList<T> unsorted
    */
    private ArrayList<T> heapify(ArrayList<T> unsorted, int i, int
        total) {
        int left = i * 2;
        int right = left + 1;
        int originalI = i;

        if(left <= total && unsorted.get(left).compareTo(unsorted
            .get(i)) > 0) {
            i = left;
        }
        if(right <= total && unsorted.get(right).compareTo(
            unsorted.get(i)) > 0) {
            i = right;
        }
        if(i != originalI) {
            T tmp = unsorted.get(originalI);
            unsorted.set(originalI, unsorted.get(i));
            unsorted.set(i, tmp);
            unsorted = heapify(unsorted, i, total);
        }

        return unsorted;
    }

    /*
    *      @Pre-Condition: ArrayList<T> unsorted is a non-empty
    *                    ArrayList<T> where T is a comparable data type with a natural

```

```

    order.
*      @Post-Condition: ArrayList<T> sorted is a permutation of
    unsorted (it contains all the same elements) in strictly non-
    decreasing order
*      @Description: heapSort will sort a given set of data in
    an ArrayList<T> in strictly non-decreasing order using the
    heap sort method.
*      @param ArrayList<T> unsorted is a non-empty ArrayList<T>
    where T is a comparable data type with a natural order
*      @return ArrayList<T> sorted is a permutation of unsorted
    in strictly non-decreasing order
*/
//Invariant for First While Loop: unsorted[i] is the parent
    element in a heap
//Invariant for Second While Loop: All elements in unsorted
    greater than the index value of y are in strictly non-
    decreasing order
public ArrayList<T> heapSort(ArrayList<T> unsorted) {
    //Debug
    Debug debugger = new Debug();

    int arrSize = unsorted.size() - 1;
    int i = arrSize / 2;
    //Initialization: Our invariant holds true before the
        first iteration of the loop because unsorted[i] must
        have child elements
    debugger.assertChildren(unsorted, i);
    while(i >= 0) {
        //Maintenance: Our invariant holds true at the
            beginning of each iteration of the loop
            because unsorted[i] must have children
            elements
        debugger.assertChildren(unsorted, i);

        unsorted = heapify(unsorted, i, arrSize);

        i--;
    }
    //Termination: Our invariant holds true at the
        termination of the loop because i will be the smallest
        index value of the loop and must have children
        elements
    debugger.assertChildren(unsorted, i);

    int y = arrSize;
    //Initialization: Our invariant holds vacuously true
        before the first iteration of the loop because there
        are no elements in unsorted that are at an index value
        greater than y.
    debugger.assertStrictLess(arrSize, y+1);

```

```

        while(y > 0) {
            //Maintenance: Our invariant holds true at the
                beginning of each iteration of the loop
                because all elements greater than y are in
                strictly non-decreasing order

            T tmp = unsorted.get(0);
            unsorted.set(0, unsorted.get(y));
            unsorted.set(y, tmp);
            arrSize--;
            unsorted = heapify(unsorted, 0, arrSize);

            y--;
        }
        //Termination: Our invariant holds true at the
            termination of the loop because y decreases as each
            largest element is moved to the end of the list until
            the entire array has been traversed, so that all
            elements greater than y are in strictly non-decreasing
            order
        debugger.assertOrder(unsorted);

        ArrayList<T> sorted = unsorted;
        return sorted;
    }

    /*
    *      @Pre-Condition: ArrayList<T> left is a non-empty sorted
        array in strictly non-decreasing order where T is a comparable
        data type with a natural order
    *      @Post-Condition: ArrayList<T> right is a non-empty sorted
        array in strictly non-decreasing order where T is a
        comparable data type with a natural order
    *      @Description: mergeTogether is used by the mergeSort
        method to recombine the left and right sections of the
        ArrayList<T> that is being sorted by merge sort. Note that
        this is a helper method for the mergeSort method, and should
        not be called externally of this class.
    *      @param ArrayList<T> left a non-empty ArrayList<T> where T
        is a comparable data type with a natural order.
    *      @param ArrayList<T> right a non-empty ArrayList<T> where
        T is a comparable data type with a natural order.
    *      @return ArrayList<T> combined should contain all the
        elements of left and right in strictly non-decreasing order
    */
    //Invariant for First While Loop: combined contains x number of
        elements where x is the sum of i and y and those elements are
        contained in left[0 ... i] or right[0 ... y] in strictly non-
        decreasing order

```

```

//Invariant for Second While Loop: combined contains x number of
    elements where x is greater than or equal to i and those
    elements are contained in left[0 ... i] in strictly non-
    decreasing order
//Invariant for Third While Loop: combined contains x number of
    elements where x is greater than or equal to y and those
    elements are contained in right[0 ... y] in strictly non-
    decreasing order
private ArrayList<T> mergeTogether(ArrayList<T> left , ArrayList<T
    > right) {
    ArrayList<T> combined = new ArrayList<T>();
    int i = 0;
    int y = 0;
    int x = 0;

    //Debug
    Debug debugger = new Debug();

    //Initialization: Our invariant holds true vacuously
        before the first execution of the loop because x, i,
        and y are all equal to zero, combined is empty and
        therefore in order
    debugger.assertEquals(0, i);
    debugger.assertEquals(0, y);
    debugger.assertEquals(0, x);
    debugger.assertEquals(i, combined.size());

    while(left.size() != i && right.size() != y) {
        //Maintenance: Our invariant holds true at the
            beginning of each iteration of the loop
            because x is incremented whenever i or y is
            incremented and elements are added to combined
            from left and right in order
        debugger.assertEquals(i+y, x);
        debugger.assertEquals(x, combined.size());
        debugger.assertOrder(combined);
        debugger.assertContains(right, left, combined);

        if(left.get(i).compareTo(right.get(y)) < 0) {
            combined.add(x, left.get(i));
            i++;
            x++;
        }
        else {
            combined.add(x, right.get(y));
            y++;
            x++;
        }
    }
}

```

```
//Termination: Our invariant holds true at the termination
    of the loop because x has been incremented whenever i
    or y has been incremented, and elements are added to
    combined from left and right in order
debugger.assertEquals(i+y, x);
debugger.assertEquals(x, combined.size());
debugger.assertOrder(combined);
debugger.assertContains(right, left, combined);

//Initialization: Our invariant holds true before the
    first execution of the loop because x has been
    incremented whenever i has been incremented and
    elements have been added to combined from left in
    order
debugger.assertGreatEquals(x, i);
debugger.assertEquals(x, combined.size());
debugger.assertOrder(combined);
debugger.assertContains(left, combined);

while(left.size() != i) {
    //Maintenance: Our invariant holds true at the
        beginning of each iteration of the loop
        because x has been incremented whenever i is
        incremented and elements have been added to
        combined from left in order
    debugger.assertGreatEquals(x, i);
    debugger.assertEquals(x, combined.size());
    debugger.assertOrder(combined);
    debugger.assertContains(left, combined);

    combined.add(x, left.get(i));
    i++;
    x++;
}
//Termination: Our invariant holds true at the
    termination of the loop because x has been incremented
    whenever i was incremented and elements have been
    added to combined from left in order
debugger.assertGreatEquals(x, i);
debugger.assertEquals(x, combined.size());
debugger.assertOrder(combined);
debugger.assertContains(left, combined);

//Initialization: Our invariant holds true before the
    first execution of the loop because x has been
    incremented whenever y has been incremented and
    elements have been added to combined from right in
    order
debugger.assertGreatEquals(x, y);
debugger.assertEquals(x, combined.size());
```

```

        debugger.assertOrder(combined);
        debugger.assertContains(right, combined);

        while(right.size() != y) {
            //Maintanance: Our invariant holds tur at the
            //beinning of each iteration of the loop because
            //x has been incremented whenever y is
            //incremented and elements have been added to
            //combined from right in order
            debugger.assertGreatEquals(x, y);
            debugger.assertEquals(x, combined.size());
            debugger.assertOrder(combined);
            debugger.assertContains(right, combined);

            combined.add(x, right.get(y));
            y++;
            x++;
        }
        //Termination: Our invariant holds true at the
        //terminatino of the loop because x has been incremented
        //whenever y was incremented and elements have been
        //added to combined from right in order.
        debugger.assertGreatEquals(x, y);
        debugger.assertEquals(x, combined.size());
        debugger.assertOrder(combined);
        debugger.assertContains(right, combined);

        return combined;
    }

    /*
     *      @Pre-Condition: ArrayList<T> unsorted is a set of data
     *      type T, where T is a Comparable data type with a natural order
     *
     *      @Post-Condition: ArrayList<T> returnValue is a
     *      permutation of unsorted in strictly non-decreasing order.
     *      @Description: mergeSort will sort a given set of data in
     *      ArrayList<T> using the merge sort method
     *      @param a non-empty ArrayList<T> unsorted where T is a
     *      Comparable data type with a natural order
     *      @return ArrayList<T> returnValue which is a permutation
     *      of unsorted, in strictly non-decreasing order,
     */
    //Invariant for First While Loop: left contains i elements, all
    //of which can be found in sorted
    //Invariant for Second While Loop: right contains y elements, all
    //of which can be found in sorted
    public ArrayList<T> mergeSort(ArrayList<T> unsorted) {
        ArrayList<T> sorted = unsorted;
        ArrayList<T> left = new ArrayList<T>();
    }

```



```
ArrayList<T> right = new ArrayList<T>();
ArrayList<T> returnValue;

//Debug
Debug debugger = new Debug();
debugger.turnOn();

if(sorted.size() <= 1) {
    returnValue = sorted;
}
else {
    int mid = (sorted.size() / 2);
    int i = 0;
    //Initialization: Our invariant holds true because
        i is zero and left contains 0 elements before
        the first iteration of the loop.
    debugger.assertEquals(i, left.size());

    while(i < mid) {
        //Maintanance: Our invariant holds true
            because i is increased at the same
            rate elements are added to left from
            the same i index in sorted
        debugger.assertEquals(i, left.size());
        debugger.assertContains(sorted, left);

        T temp = sorted.get(i);
        left.add(temp);

        i++;
    }
    //Termination: Our invariant holds true because i
        has been incremented at the same rate
        elements are added to left from the same index
        i in sorted
    debugger.assertEquals(i, left.size());
    debugger.assertContains(sorted, left);

    int y = mid;
    //Initialization: Our invariant holds true
        because i is zero and right contains 0
        elements before the first iteration of the
        loop.
    debugger.assertEquals(y, right.size());

    while(y < sorted.size()) {
        //Maintanance: Our invariant holds true
            because y is increased at the same
            rate elements are added to right from
            the same y index in sorted.
```

```

        debugger.assertEquals(y, right.size());
        debugger.assertContains(sorted, right);

        T temp = sorted.get(y);
        right.add(temp);

        y++;
    }
    //Termination: Our invariant holds true because y
        has been incremented at the same rate
        elements are added to left from the same index
        y in sorted
    debugger.assertEquals(y, right.size());
    debugger.assertContains(sorted, right);

    left = mergeSort(left);
    right = mergeSort(right);
    returnValue = mergeTogether(left, right);
}
return returnValue;
}

/*
 *      @Pre-Condition: ArrayList<T> unsorted is an unsorted
 *      ArrayList of a comparable data type that is non-empty
 *      @Post-Condition: ArrayList<T> will return a permutation
 *      of <code>unsorted</code> that will be in increasing order
 *      @Description: insertionSort will sort an ArrayList of
 *      generic type T in increasing order using an insertion sort
 *      @param ArrayList<T> unsorted is a non-empty unsorted
 *      array list of T, where T is a comparable type
 *      @return sorted is a permutation of <code>unsorted</code>
 *      where all the elements are sorted in increasing order
 */
// INVARIANT (Outer-Loop): The pre condition implies that sorted
    [0 ... i - 1] will contain all the same data as unsorted[0 ...
        i - 1].
// INVARIANT (Inner-Loop): sorted[0 ... j] is sorted in strictly
    non-decreasing order.
public ArrayList<T> insertionSort(ArrayList<T> unsorted) {
    Debug debugger = new Debug<List<T>>();
    debugger.turnOn();
    ArrayList<T> sorted = unsorted;
    if(sorted.size() > 1) {
        int i = 1;
        /* INITIALIZATION (Outer-Loop): The invariant
            holds because i = 1, and there is one element
            in the subarray of sorted[0 ... i - 1] and
            unsorted[0 ... i - 1], */
        List<T> subSortedOI = sorted.subList(0, i - 1);

```

```

List<T> subUnsortedOI = unsorted.subList(0, i -
    1);
debugger.assertEquals(subUnsortedOI, subSortedOI)
    ;

while(i < sorted.size()) {
    /* MAINTENANCE (Outer-Loop): At the
       beginning of each iteration of the
       loop, the loop invariant is maintained
       because the subarray of sorted[0 ...
       i - 1] contains all the same elements
       as
           unsorted[0 ... i - 1] */
    List<T> subSortedOM = sorted.subList(0, i
        - 1);
    List<T> subUnsortedOM = unsorted.subList
        (0, i - 1);
    debugger.assertEquals(subUnsortedOM,
        subSortedOM);

    T value = sorted.get(i);
    int j = i - 1;
    // INITIALIZATION (Inner-Loop): Before
       the first iteration of the loop, j =
       0, the subarray of sorted[0 ... 0]
       contains one elements and therefore
       the invariants holds vacuously.
    List subSortedII = sorted.subList(0, j);
    debugger.assertOrder(subSortedII);

    while(j >= 0 && (value.compareTo(sorted.
        get(j)) < 0)) {
        // MAINTENANCE: (Inner-Loop): At
           the beginning of each
           iteration sorted[0 ... j] is
           sorted in stricly non-
           decreasing order
        List subSortedIM = sorted.subList
            (0, j);
        debugger.assertOrder(subSortedIM)
            ;

        sorted.set(j+1, sorted.get(j));
        j--;
    }
    sorted.set(j+1, value);
    // TERMINATION (Inner-Loop): The negation
       of the guard implies that the sorted
       [0 ... j] has been traversed and is
       stricly non-decreasing order.

```

```

        List subSortedIT = sorted.subList(0, j+1)
        ;
        debugger.assertOrder(subSortedIT);

        //Count up on the iterator
        i++;
    }
    /* TERMINATION (Outer-Loop): When the loop
       terminates, i is equal to sorted.size()
       meaning the entire array has been traversed
       and that the guard has been negated.
       The negation of the guard implies that
       sorted[0 ... i - 1] contains all the
       elements of unsorted[0 ... i - 1] */
    List subSortedOT = sorted.subList(0, i - 1);
    debugger.assertOrder(subSortedOT);
    Integer integerI = new Integer(i);
    Integer sortedSizeO = new Integer(sorted.size());
    debugger.assertEquals(sortedSizeO, integerI);
    debugger.assertEquals(unsorted, sorted);
}
return sorted;
}

/*
 *      @Pre-Condition: ArrayList<ArrayList<T>> unsorted is an
 *      unsorted a nested non-empty ArrayList of a non-empty ArrayList
 *      (in tabular format) of a comparable data type with a natural
 *      order where sortingIndex is an index value of the nest
 *      ArrayList.
 *      @Post-Condition: ArrayList<ArrayList<T>> will return a
 *      permutation of <code>unsorted</code> that will be in stricly
 *      non-increasing order.
 *      @Description: insertionSortNestedArray will sort a nested
 *      ArrayList in tabular format of a comparable data type and
 *      given a specific index value of the inner ArrayList will sort
 *      the inner ArrayLists into stricly non-increasing order within
 *      the outer ArrayList
 *      @param ArrayList<ArrayList<T>> list is a non-empty unsorted
 *      nested ArrayList of ArrayList of data type T, where T is a
 *      comparable data type with a natural order.
 *      @param int sortingIndex is an index value of the inner
 *      ArrayList to use for sorting comparisons
 *      @return ArrayList<ArrayList<T>> list is a permutation of
 *      <code>unsorted</code> where all the elements in the outer
 *      ArrayList are sorted in stricly non-increasing order by inner
 *      ArrayLists index value of sortingIndex
 */

```

```

//INVARIANT (Outer-Loop): The Pre-Condition implies that A[0 ...
    i - 1] will contain all the same data as A'[0 ... i - 1]
//INVARIANT (Inner-Loop): A[0 ... j] is sorted in stricly non-
    increasing order
public ArrayList<ArrayList<T>> insertionSortNestedArray(ArrayList
    <ArrayList<T>> unsorted, int sortingIndex) {
    Debug debugger = new Debug<List<T>>>();
    ArrayList<ArrayList<T>> list = unsorted;

    if(list.size() > 1) {
        int i = 1;

        /*INITIALIZATION (Outer-Loop): Before the first
            iteration of the loop the invariant holds
            beause i = 1, and there is one elment in the
            subarray of A[0 ... i - 1] and A'[0 ... i - 1]
            */
        List<ArrayList<T>> subSortedOI = list.subList(0,
            i - 1);
        List<ArrayList<T>> subUnsortedOI = unsorted.
            subList(0, i - 1);
        debugger.assertEquals(subUnsortedOI, subSortedOI)
            ;

        while(i < list.size()) {
            /*MAINTENANCE (Outer-Loop): At the
                beginning of each iteration of the
                loop, the loop invariant is maintained
                because the subarray of A'[0 ... i -
                1] contains all the same elements as A
                [0 ... i - 1] */
            List<ArrayList<T>> subSortedOM = list.
                subList(0, i - 1);
            List<ArrayList<T>> subUnsortedOM =
                unsorted.subList(0, i - 1);
            debugger.assertEquals(subUnsortedOM,
                subSortedOM);

            ArrayList<T> currentElement = list.get(i)
                ;
            T value = list.get(i).get(sortingIndex);
            int j = i - 1;

            /*INITIALIZATION (Inner-Loop): Before the
                first iteration of the loop, j = 0,
                the subarray of sorted[0 ... 0]
                contains one element and therefore the
                invariants hold vacuously. */
            List subSortedII = list.subList(0, j);

```

```

        debugger.assertOrder(subSortedII,
                               sortingIndex);
        while(j >= 0 && (value.compareTo(list.get
            (j).get(sortingIndex)) > 0)) {
            /*MAINTENANCE (Inner-Loop): At
              the beginning of each
              iteration A[0 ... j] is sorted
              in stricly non-increasing
              order */
            List subSortedIM = list.subList
                (0, j);
            debugger.assertOrder(subSortedIM,
                                   sortingIndex);

            list.set(j+1, list.get(j));
            j--;
        }
        /*TERMINATION (Inner-Loop): The negation
          of the gaurd implies that A'[0 ... j]
          has been entirely traversed and is in
          stricly non-increasing order. */
        List subSortedIT = list.subList(0, j+1);
        debugger.assertOrder(subSortedIT,
                               sortingIndex);

        list.set(j+1, currentElement);

        i++;
    }
    /*TERMINATION (Outer-Loop): When the loop
      terminates, i is equal to A'.length meaning
      the entire array has been traversed and that
      the guard has been negated.

```

```

        List subSortedOT = list.subList(0, i - 1);
        debugger.assertOrder(subSortedOT, sortingIndex);
        debugger.assertEquals(new Integer(list.size()),
                               new Integer(i));
        debugger.assertEquals(unsorted, list);
    }

    return list;
}

```

The class Stopwatch has been altered from its original form.

Listing 4
STOPWATCH

```

/*****
 *
 * Compilation:  javac Stopwatch.java
 *
 *****/

```

```

*
*****/

/**
 * The <tt>Stopwatch</tt> data type is for measuring
 * the time that elapses between the start and end of a
 * programming task (wall-clock time).
 *
 * See {@link StopwatchCPU} for a version that measures CPU time.
 *
 * @author Robert Sedgewick
 * @author Kevin Wayne
 * @update @ 5.3.15 by Preston Stosur-Bassett, added start method.
 */

public class Stopwatch {

    private long start;

    /**
     * Starts the stopwatch timer
     */
    public void startTime() {
        start = System.currentTimeMillis();
    }

    /**
     * Returns the elapsed time (in seconds) since this object was
     * created.
     */
    public double elapsedTime() {
        long now = System.currentTimeMillis();
        return (now - start) / 1000.0;
    }
}

}

Listing 5
DEBUG

/*
 * @Author Preston Stosur-Bassett
 * @Date Jan 21, 2015
 * @Class Debug
 * @Description This class will help debugging by being able to turn
 * on and turn off debug messages easily
 */

import java.util.List;

```



```
import java.util.ArrayList;

public class Debug<T> {
    boolean debugOn; //Variable to keep track of whether or not debug
                     is on

    /*
     *      @Description constructor method that sets the default
     *      value of debugOn to false so that debug statements will not
     *      automatically print
     */
    public void Debug() {
        debugOn = false;
    }

    /*
     *      @Description turn on debugging print statements
     */
    public void turnOn() {
        debugOn = true;
    }

    /*
     *      @Description turn off debugging print statements
     */
    public void turnOff() {
        debugOn = false;
    }

    /*
     *      @Description will print messages only when debugOn
     *      boolean is set to true
     *      @param String message the string to print when debugging
     *      is turned on
     */
    public void print(T message) {
        if(debugOn == true) {
            System.out.println(message);
        }
    }

    /*
     *      @Pre-Condition <code>T expected</code> and <code>T actual
     *      </code> are both of the same type T
     *      @Post-Condition If <code>T expected</code> and <code>T
     *      actual</code> are found to be equal, the program moves on,
     *      otherwise the program halts with <code>AssertionError</code>
     *      is thrown
     *      @Description runs an assert statement against an expected
     *      value and the actual value that are passed as parameters only
    
```

```

        when <code>debugOn == true</code>
    *      @param T expected the expected value to assert against
        the actual value
    *      @param T actualt he actual value to assert against the
        expected value
    */
    public void assertEquals(T expected, T actual) {
        if(debugOn == true) {
            assert actual.equals(expected);
        }
    }

    /*
    * @Pre-Condition: <code>List<Integer> actual</code> is a iterable
        list of Integer objects
    *      @Post-Conditions: If the List of Integer objects is in
        stricly non-decreasing order, the program moves on normally,
        if not, the program halts with an <code>AssertionError</code>
    * @Description: runs an assertion statement against a list of
        Integer objects to ensure that for <code>k = actual.size(); A[
        k - 2] <= A[k - 1];</code>
    *      @param List<Integer> actual the list to assert is in
        stricly non-decreasing order
    */
    public void assertOrder(List<Integer> actual) {
        if(debugOn == true) {
            int i = actual.size();
            while(i > 1) {
                assert actual.get(i - 1).compareTo(actual
                    .get(i - 2)) >= 0;

                i--;
            }
        }
    }

    /*
    *      @Pre-Condition: <code>List<ArrayList<Integer>> actual</
    code> is an iterable list of ArrayList of Integer objects
        where sortingIndex is an index value of the ArrayList
    *      @Post-Condition: If the Llist of ArrayList of Integer
        objects is in stricly non-increasing order, the program moves
        on normally, if not, the program halts with an <code>
        AssertionError</code>
    *      @Description: runs an assertion statement against a list
        of ArrayList of Integer objects to ensure that for <code>k =
        actual.size(); A[k-2] >= A[k-1];</code>
    *      @param List<ArrayList<Integer>> actual the list to assert
        is in stricly non-decreasing order
    */

```

```

*      @param int sortingIndex the index value to make sorting
*      comparisons from
*/
public void assertOrder(List<ArrayList<Integer>> actual, int
    sortingIndex) {
    if(debugOn == true) {
        int i = actual.size();
        while(i > 1) {
            assert actual.get(i - 1).get(sortingIndex)
                .compareTo(actual.get(i - 2).get(
                    sortingIndex)) <= 0;

            i--;
        }
    }
}

/*
*      @Pre-Condition: <code>ArrayList<Integer> actual</code> is
*      an ArrayList of Integer Objects
*      @Post-Condition: If the ArrayList of Integer Objects is
*      in stricly non-decreasing order, the program moves on normally
*      , if not, the pgoram halts with an <code>AssertionError</code>
*      @Description: runs an assertion statement against an
*      ArrayList of Integer Objects to ensure that for <code>k =
*      actual.size(); A[k-2] <= A[k-1];</code>
*      @param ArrayList<Integer> actual the ArrayList to assert
*      is in stricly non-decreasing order
*/
public void assertOrder(ArrayList<Integer> actual) {
    if(debugOn == true) {
        int i = actual.size();
        while(i > 1) {
            assert actual.get(i - 1).compareTo(actual
                .get(i - 2)) >= 0;

            i--;
        }
    }
}

/*
*      @Pre-Condition: ArrayList is an ArrayList of Integers and
*      i is less than or equal to half of the size of actual
*      @Post-Condition: If elements exist past i the assertion
*      holds
*      @Description: runs an assertion statement against an
*      ArrayList of Integer Objects to ensure that there are children
*      nodes of actual[i].

```

```

*      @param ArrayList<Integer> actual the array to test
*      against
*      @param int i the index value to check has children nodes.
*/
public void assertChildren(ArrayList<Integer> actual, int i) {
    if(debugOn == true) {
        assert actual.size() > i;
    }
}

/*
*      @Pre-Condition: actual and expected both contain Integer
*      Objects
*      @Post-Condition: If all the elements inside of the actual
*      arraylist are also contained in the expected arraylist, then
*      the assertion holds true
*      @Description: Tests to ensure a given ArrayList of
*      Integer Objects contains all the elements of another given
*      ArrayList of Integer Objects
*      @param ArrayList<Integer> expected the list to check
*      contains against
*      @param ArrayList<Integer> actual the list to check to
*      make sure all its elements are contained in the other
*      arraylist
*/
public void assertContains(ArrayList<Integer> expected, ArrayList
<Integer> actual) {
    if(debugOn == true) {
        for(int i = 0; i < actual.size(); i++) {
            assert expected.contains(actual.get(i));
        }
    }
}

/*
*      @Pre-Condition: expectedOne, expectedTwo, and actual all
*      contain Integer Objects
*      @Post-Condition: If all the elems inside of the actual
*      ArrayList are also contined in either the expectedOne
*      ArrayList or the expectedTwo ArrayList, then the assertion
*      holds true
*      @Description: Tests to ensure a given ArrayList of
*      Integer Objects contains all the elements of another given
*      ArrayList of Integer Objects
*      @param: ArrayList<Integer> expectedOne one of the lists
*      to check to see if the given ArrayList actual's elements are
*      contained in
*      @param: ArrayList<Integer> expectedTwo one of the lists
*      to check to see if the given ArrayList actual's elements are
*      contained in

```

```

    * @param: ArrayList<Integer> actual the list to check ot make
      sure all its elements are contained in either expectedOne or
      expectedTwo
    */
    public void assertContains(ArrayList<Integer> expectedOne,
        ArrayList<Integer> expectedTwo, ArrayList<Integer> actual) {
        if(debugOn == true) {
            for(int i = 0; i < actual.size(); i++) {
                assert expectedOne.contains(actual.get(i))
                    || expectedTwo.contains(actual.get(i))
            }
        }
    }

    /*
    * @Description: asserts that the first arguement is stricly
      greator than the second arguement
    * @param int large an integer primative value to assert is
      strictly greator than the second arguement
    * @param int small an integer primative value to assert the
      first arguement is strictly greator than.
    */
    public void assertStrictGreat(int large, int small) {
        if(debugOn == true) {
            assert large > small;
        }
    }

    /*
    * @Description: asserts that the first arguement is
      strictly less than the second arguement
    * @param int small an integer primative value to assert is
      stricly less than the second arguement
    * @param int large an integer primative value to assert the
      first arguement is strictly less than.
    */
    public void assertStrictLess(int small, int large) {
        if(debugOn == true) {
            assert small < large;
        }
    }

    /*
    * @Description: asserts that the first arguement is greator
      than or equal to the second arguement
    * @param int large an integer primative value to assert is
      greator than or equal to the second arguement
    * @param int small an integer primative value to assert the
      first arguement is greator than or equal to.

```

```

    */
    public void assertGreatEquals(int large, int small) {
        if(debugOn == true) {
            assert large >= small;
        }
    }

    /**
     *      @Description: asserts that the first argument is less
     *      than or equal to the second argument
     *      @param int small an integer primitive value to assert is
     *      less than or equal to the second argument
     *      @param int large an integer primitive value to assert the
     *      first argument is less than or equal to.
     */
    public void assertLessEquals(int small, int large) {
        if(debugOn == true) {
            assert small <= large;
        }
    }
}

```

Listing 6
DUMMYDATA

```

/*
 *      @Author Preston Stosur-Bassett
 *      @Date Jan 25, 2015
 *      @Class DummyData
 *      @Description This class contains methods to generate dummy data
 *      given a set of parameters.
 */

import java.util.ArrayList;
import java.util.Random;

public class DummyData {

    /**
     *      @Description runArrayList<Integer> will take an ArrayList
     *      of Integer Objects and add a given amount of values to it
     *      @param int end the ending value to denote when to stop
     *      adding to the array list
     *      @param int min the minimum value of the randomly
     *      generated data.
     *      @param int max the maximum value of the randomly
     *      generated data.
     *      @param ArrayList<Integer> list the list to add value to
     *      and return
     *      @return ArrayList<Integer> the list after it has been
     *      updated with the randomly generated data
     */
}

```

```

*/
public static ArrayList<Integer> runArrayList(int end, int min,
    int max, ArrayList<Integer> list) {
    Random random = new Random();
    Debug debugger = new Debug();
    int start = 0;
    // INVARIANT: A.length >= start
    // INITIALIZATION: start = 0, A.length can be longer than
    // 0 when initially passed, but not smaller, so our
    // invariant holds
    debugger.assertGreatEquals(list.size(), start);
    while(start < end) {
        // MAINTANANCE: At the beginning of each
        // iteration, one element was added to A and
        // start was increased by one, therefore, our
        // invariant holds true.
        debugger.assertGreatEquals(list.size(), start);
        Integer intToAdd = new Integer(random.nextInt((
            max - min + 1) + min));
        if(intToAdd != 0) {
            list.add(intToAdd);

            start++;
        }
    }
    /*TERMINATION: The negation of the guard implies that (
    end - start) number of elements have been added to A,
    since start is initialized as 0 at the beginning of
    the method and is
        incremented by 1 each iteration of the loop,
        which means that start amount of elements have
        been added to A, and so our invariant holds
        true.    */
    debugger.assertGreatEquals(list.size(), start);

    return list;
}

/*
 *      @Description: runArrayList<String> will take an ArrayList
 *      of String Objects and add a given amount of String numerical
 *      values to it
 *      @param int end the ending value to denote when to stop
 *      adding to the array list
 *      @param ArrayList<String> list the list to add String
 *      values to and return
 *      @return ArrayList<String> the list after it has been
 *      updated with the randomly generated numerical String values
 */

```

```

public static ArrayList<String> runArrayList(int end, ArrayList<
    String> list) {
    Random random = new Random();
    Debug debugger = new Debug();
    int start = 0;
    // INVARIANT: A.length >= start
    // INITIALIZATION: Before the first iteration of the loop
    , start = 0 and A.length cannot be less than 0, so our
    invariant holds true
    debugger.assertGreatEquals(list.size(), start);
    while(start < end) {
        // MAINTENANCE: At the beginning of each
        iteration of the loop our invariant holds
        because for each iteration of the loop one
        element is added to A and start is incremented
        by 1
        debugger.assertGreatEquals(list.size(), start);
        Integer intToString = new Integer(random.nextInt
            ((1000000 - 1) + 1));
        String intString = String.valueOf(intToString);
        list.add(intString);

        //Count up on the iterator
        start++;
    }
    /* TERMINATION: The negation of the guard implies that (
    end - start) number of elements have been added to A,
    since start is initialized as 0 at the beginning of
    the method and is
        incremented by 1 each iteration of the
        loop, which means that start amount of
        elements have been added to A, and so
        our invariant holds true. */
    debugger.assertGreatEquals(list.size(), start);

    return list;
}

/*
 *      @Description: identicalElement will take an element and
 *      add it to the ArrayList<Integer> for a given amount of times
 *      @param int end the ending value to denote when to stop
 *      adding elements to the array
 *      @param int element the element to add over and over again
 *      to the array
 *      @param ArrayList<Integer> list the list to add elements
 *      to
 *      @return ArrayList<Integer> the list after it has been
 *      updated with the given data
 */

```



```

public static ArrayList<Integer> identicalElement(int end, int
    element, ArrayList<Integer> list) {
    // INVARIANT: A.length >= start
    int start = 0;
    Debug debugger = new Debug();
    //The element to add over and over again
    Integer iden = new Integer(element);
    // INITIALIZATION: Before the first iteration of the loop
    , start = 0 and A.length cannot equal anything less
    than 0, so our invariant holds true
    debugger.assertGreatEquals(list.size(), start);
    while(start < end) {
        // MAINTENANCE: At the beginning of each
        iteration of the loop our invariant holds
        because for each iteration of the loop one
        element is added to A and start is incremented
        by 1
        debugger.assertGreatEquals(list.size(), start);
        list.add(iden);

        //Count up on the iterator
        start++;
    }
    /* TERMINATION: The negation of the guard implies that (
    end - start) number of elements have been added to A,
    since start is initialized as 0 at the beginning of the
    method and is
        incremented by 1 each iteration of the
        loop, which means that start amount of
        elements have been added to A, and so
        our invariant holds true */
    debugger.assertGreatEquals(list.size(), start);

    return list;
}
}

```