MOTIVATION

Understanding algorithm design is fundamental to understanding the basic operations of computer science and software development. Though algorithms exist outside of digital computing, the age of modern computing has heralded a time wherein algorithms have become an integral part of the technology with which we interact every day. Resultantly, algorithms are now synonymous with their software implementations, and as Cormen et al. introduce in the third edition of *Introduction to Algorithms*, algorithms are seen as "tools for solving well-specified computational problems (5)." Such tools can be harnessed to effectively utilize finite computational resources and thereby solve complex problems, which without efficient algorithms would otherwise require inordinate amounts of labor or whose humanly computation would become irrelevant upon completion.

Aware that algorithms play a fundamental role in computer science and have innumerable practical applications in software development, computer science and software development curriculums consider an understanding of algorithms to be a foundational pillar in academic and professional training.

However, the study of algorithms is broad and unwieldly, as grappling with its many intricacies and applications is not possible within the time constraints of an academic term. Therefore, in an attempt to keep the study of algorithms tangible and germane, the current canonical approach in algorithm pedagogy is to introduce algorithmic concepts through various computational problems.

But learning computational algorithms is not merely a novice exercise in algorithm analysis; rather, an understanding of the function, applications, advantages, and disadvantages of various algorithmic approaches finds many practical applications in computer science and software development.

Combinatorial optimization is something that is universally utilized both in and outside of digital computing; finding an optimal object from a finite set of objects is a test of efficiency and has applications in artificial intelligence, machine learning, mathematics, auction theory, and software engineering.

One combinatorial optimization algorithm is the *0/1 Knapsack Problem*. Because of its tangibility, relatability, and known algorithmic solutions of varying time complexity, an analysis of the 0/1 Knapsack Problem serves as an ample introduction to combinatorial optimization and to algorithmic studies as a whole.

There are at least three existing solutions to the 0/1 Knapsack Problem—a greedy algorithm, a dynamically programed algorithm, and a brute force algorithm.

This paper seeks to understand how each of the aforementioned algorithms works by analyzing each algorithm's pseudocode, correctly implementing and documenting the algorithm in the Java programming language, and analyzing an experiment using the Java implementation.

Graphing the data collected from the experimentation will provide an effective visual representation of the differences in each solution's time complexity and performance.

BACKGROUND

Previously mentioned was that the 0/1 Knapsack Problem has at least three algorithmic solutions. According to Cormen et al. in *Introduction to Algorithms,* the first two algorithms, greedy and dynamic, "exploit optimal substructure" and might in turn be mistaken as similar approaches. An understanding of the subtle differences between the two techniques is illustrated through implementations and data collection. The latter algorithm, brute force, does not share the same similarities, and its implementation and results are wildly different than the former two, observed later in this paper.

First, it is paramount to understand the nature of the 0/1 Knapsack Problem, and how it differs from related combinational optimization problems. According to Cormen et al., the 0/1 Knapsack Problem can be imagined as;

> A thief robbing a store finds n items. The ith item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers. The thief wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack, for some integer W. Which items should he take? (We call this the 0-1 knapsack problem because for each item, the thief must either take it or leave it behind; he cannot take a fractional amount of an item or take an item more than once.)

The 0/1 Knapsack Problem differs from the *Fractional Knapsack Problem* in that the thief must make binary choices of whether or not to take an item, as opposed to taking fractions of items.

The primary difference between greedy and dynamic algorithms comes in correctness. Interestingly, of the two knapsack problems, the greedy solution is correct only for the Fractional Knapsack Problem, whereas it provides correct solutions for the 0/1 Knapsack only a portion of the time.

Despite not always providing correct solutions, the greedy algorithm has use in the 0/1 Knapsack Problem—the greedy algorithm finds a solution very quickly, regardless of its correctness. Should solution speed ever be favored over solution correctness when solving a combinational optimization problem, then the greedy algorithm is to be preferred. However, correctness is often important, and so correct algorithms may be implemented over the greedy solution.

When finding the solution to a 0/1 Knapsack Problem with a large number of elements is required, it is important to examine how a certain algorithm will perform when needing to find a solution. Despite guaranteeing correct solutions, the brute force algorithm's time complexity performance on large 0/1 Knapsack problems is greatly superseded by other algorithms. The dynamic programming algorithm provides an alternative to the two extremes, offering correct solutions with a reasonable time complexity.

Pseudocode analysis and implementation experimentation allow for the precise calculation of exactly how long finding a 0/1 Knapsack solution for a collection of n items will take with each respective solution algorithm. *Time complexity* expresses the quantifiable amount of time taken by an algorithm to run as a function of the number of elements in a collection. *Big-O notation* is used to describe the worst case running time of an algorithm. To do this, only the n term of the largest order is considered, because as n grows large, the largest order is all that matters. In *Big-O* notation, the time complexity for the greedy algorithm is the quadratic $O(n \log_2 n)$; for n elements in a collection, the sorting time will be $n \log_2 n$. Comparatively, the time complexities for the dynamic solution and the brute force solutions are $O(nW)$, and $O(2^n)$, respectively, where W is the maximum capacity of the knapsack.

However, it is important to recognize that Big-O describes an algorithm's time complexity in the worst case scenarios, and real world results may differ. This paper observes the solution times for greedy, dynamic, and brute force solutions.

PROCEDURE

Greedy Algorithm

The greedy algorithm follows the steps presented in the pseudocode below.

```
GREEDY-ALGORITHM(Values[], Weights[], Capacity)
1      if Values[].length = 0
2            return 0
3      sort KnapsackItems[]
4      for KnapsackItems[].length to 0
5            if KnapsackItem.weight <= remainingCapacity
6                  valueTaken = valueTaken + KnapsackItem.value
7                  remainingCapacity = remainingCapacity - KnapsackItem.weight
8      return valueTaken
```

The algorithm makes use of a **for loop** and therefore a loop invariant. The loop invariant reads as follows;

> At the start of each iteration of the **for loop** on lines four through seven in GREEDY-ALGORITHM, each element in the array *KnapsackItems* is comparable, and the density of the taken items exceeds the density of the items that are not taken, if the items that are not taken have weights below the remaining capacity of the knapsack, and at least one item has been taken.

Initialization: At the start of the first iteration of the **for loop** on lines four through seven, each element in *KnapsackItems* is comparable. No items have been taken, so the remaining capacity is equivalent to the total capacity, and the nonexistent taken items cannot be compared by price density, so the loop invariant remains true.

Maintenance: At each iteration of the **for loop**, the next most price dense item is taken. The remaining capacity is at least equal to or greater than zero, and the price density of the taken item is less than all items proceeding it and greater than all items that follow it, unless a proceeding item has a weight greater than the remaining capacity.

Termination: The **for loop** terminates when each element in the array has been compared or when the remaining capacity of the knapsack is equal to zero.

Dynamic Programming Algorithm

The dynamic programming algorithm follows the steps in the two methods presented in the pseudocode below. The HELPER method is recursive. This pseudocode is adapted from Sahni.

```
DYNAMIC-ALGORITHM(Values[], Weights[], Capacity)
1      if Values.length = 0
2            return 0
3      return HELPER(0, Capacity)

HELPER(i, Capacity)
1      if i = numberOfObjects
2            if Capacity < Weights[numberOfObjects]
3                  return 0
4            return Values[numberOfObjects]
5      if Capacity < Weights[i]
6            return HELPER(i+1, Capacity)
7      return MAX(HELPER(i+1, Capacity), HELPER(i+1, Capacity-Weights[i]) + Values[i])
```

The algorithm makes use of an **if** statement and therefore an invariant. The invariant reads as follows;

> With each evaluation of the **if** statement's guard, a recursive call occurs. The called recursion makes an optimal choice to fill the knapsack.

Initialization: At the start of the first recurs, an optimal choice has not been made. The invariant holds trivially as the recursion will return an optimal choice at its completion.

Maintenance: The invariant is preserved with each recurs because the optimal choice—adding an item to the knapsack or ignoring it and continuing onto the next item—is decided. Because the 0/1 Knapsack Problem can be divided into smaller sub problems, the optimal choice in the recurs' sub-problem results in an optimal solution for the recurs.

Termination: After the last recurs, the recursive process to find optimal choices in sub-problems is complete; the taken items and highest possible profit within the 0/1 Knapsack Problem's constraints is determined.

Brute Force Algorithm

The brute force algorithm follows the steps among the three methods presented in the pseudocode below. This pseudocode is adapted from Hristakeva and Shrestha.

```
BRUTE-FORCE(Weights[], Values[], A[])
1     for i to 2^n do
2           j = n
3           tempWeight = 0
4           tempValue = 0
5           while A[j] != and j > 0
6                 A[j] = 0
7                 j--
8           A[j] = 1
9           for k = 1 to n do
10                if A[k] = 1 then
11                      tempValue = tempValue + Values[k]
12                      tempWeight = tempWeight + Weights[k]
13          if tempValue > bestValue AND tempWeight <= Capacity then
14                bestValue = tempValue
15                best Weight = tempWeight
16          bestChoice = A
17    return bestChoice
```

The algorithm makes use of a **for loop** and therefore a loop invariant. The loop invariant reads as follows;

> As each iteration of the **for** loop begins, the temporary value *tempValue* is checked against the maximum, ensuring that *tempValue* does not exceed the maximum.

Initialization: At the start of the **for loop**'s first iteration, both the temporary value *tempValue* and the maximum values are set equal to zero, so the loop invariant trivially holds.

Maintenance: The **for loop**'s invariant is maintained with each iteration, as the temporary value *tempValue* is not made greater than the set maximum profit.

Termination: At the close of the loop, the entire array of Values[] has been analyzed and the taken value is equal to or greater than all other permutations of the array Values[].

SIMULATION TESTING

To ensure that experimental results can be trusted, it is necessary to test the mechanisms that will complete the full testing.

An empty array and arrays of four and one-million items were used as test input for each algorithm. The actual results matched the expected results, as seen in **Table 1** below.

Input	*Expected Results*	*Actual Results*
Empty Array	Nothing returned	Nothing returned
Array of four items	Correctly determined solution	Correctly determined solution
Array of one-million items	Slow processing	Java Exception

Table 1: Test Results

Additionally, `assert` statements are included in the Java implementation to prove that each loop invariant holds true.

Problems Encountered

The dynamic programming algorithm implemented in this paper was somewhat challenging to understand.

First tests revealed that outliers were quite common in the data. This was resolved by requiring a number of trials on each *n* so that various combinations of values and weights could be tested.

The brute force algorithm is not very time efficient after *n* exceeds twenty-five. This was not realized until partway through a test. The issue was later corrected by limiting brute force to run for only $1/10^{th}$ of the end size of *n*.

EXPERIMENTAL ANALYSIS: METHODOLOGY AND RESULTS

Each solution algorithm was implemented in the object oriented Java programming language. Using a series of Java classes to facilitate input generation, testing, and data collection, an experiment was conducted to confirm if the implementation in Java would yield the same results as the theoretical models. The complete Java classes can be found in the Appendix. The table below provides brief summaries of each classes' function.

Class	*Purpose*
KnapsackMain	Holds the main() method, which runs algorithm analyses, as well as the testing conditions. The main() method increments a dataset size and runs sorts on each dataset.
DoublesDataGenerator	Creates a new array and fills it with random data of type `double`.
AlgorithmTester	Holds the main testing method, as well as some utility methods.
TestResult	Encapsulates data from testing as a single (x,y) point.
KnapsackItem	Encapsulates value, weight, and price density data for individual items.
Sorter	An interface for the sorter class.
QuickSort	The Java implementation of the Quick Sort algorithm.
Solution	An interface for the solution classes.
GreedySolution	The Java implementation of the 0/1 Knapsack's greedy algorithm.
DynamicSolution	The Java implementation of the 0/1 Knapsack's dynamically programmed algorithm.
BruteforceSolution	The Java implementation of the 0/1 Knapsack's brute force algorithm.
Stopwatch	Serves as a timer to calculate the millisecond duration of an action.
MultiFileWriter	Serves as wrapper for multiple FileWriters, directing input to the correct FileWriter, which in turn, writes experiment data to a comma separated value (CSV) file.

Table 3: Classes developed for this experiment

Methodology

The experimentation was conducted under the following conditions;

initial n = 0 *maximum n = 300* *max. value = 120* *n increment = 1* *trials = 5* *repetitions = 5*

Under the above conditions, each experiment was conducted on arrays of size `n = 0` to `n = 300`. For each experiment, *n* was incremented by one, underwent five trials and five repetitions, and the mean sort time was calculated from the five trials and printed to a `.csv` file.

For the brute force algorithm, the experiment was conducted only to a size of `n = 30`, as calculating the solution beyond thirty proved excessively time consuming and restricted further data collection on the greedy and dynamic algorithms.

Specifically, `KnapsackMain` creates a `DoublesDataGenerator` object. Within a **for loop**, the `DoublesDataGenerator` object repeatedly creates random arrays that are fed into an `AlgorithmTester` object. The `AlgorithmTester` object takes a solution as a parameter—the

passed sorter is the sorter used for a particular trial. Trial conditions are passed as parameters to a `testAlgorithm()` function, which returns a mean sort time to a `TestResult` object, eventually printed by the `MultiFileWriter` to a `.csv` file. For each value `n`, this is done five times, so as to remove the chance of outliers skewing the data.

Within the `AlgorithmTester` object, the `Solution` object completes sorts *r* times, where *r* is the number of repetitions—in this experiment, five. For each algorithm, a mean time is calculated via the `Stopwatch` object instantiated in the `AlgorithmTester` object. The mean of the times is returned as the `TestResult` object.

Results

The hypotheses and actual results can be found in the table below;

Solution Algorithm	*Hypothesis*	*Actual Result*
Greedy Algorithm	As *n* grows large, the time to compute a solution would remain nearly the same.	As *n* grew large, the time to compute a solution remained nearly the same.
	The computed solution will not always be accurate or consistent with the other two algorithms.	The greedy algorithm was slower than the dynamic algorithm for n < 50.
		The computed solutions were not consistent with the other two algorithms when *n* was small, but as *n* grew large, all three algorithms more frequently yielded the same computed solution.
Dynamically Programmed Algorithm	As *n* grows large, the mean time to compute a solution would grow.	As *n* grew large, the mean time to compute solutions grew.
	The computed solution would match the brute force algorithm's solution.	The dynamic algorithm was faster than the greedy algorithm for n < 50.
		The solutions found by the dynamic algorithm were always consistent with the brute force algorithm's solutions.
Brute Force Algorithm	As *n* grows large, the mean time to compute a solution would grow greatly.	As *n* grew large, the mean time to compute a solution grew greatly.
	The computed solution would match the dynamic algorithm's solution.	The computed solutions were always consistent with the dynamic algorithm's solutions.

Table 2: Table of expected and actual results

As observed in **Table 2**, the experimental results mostly confirm the hypotheses.

Figure 1 displays the results of the twelve experiments, with $100n$ and $n \log_2 n$ graphed for comparison.

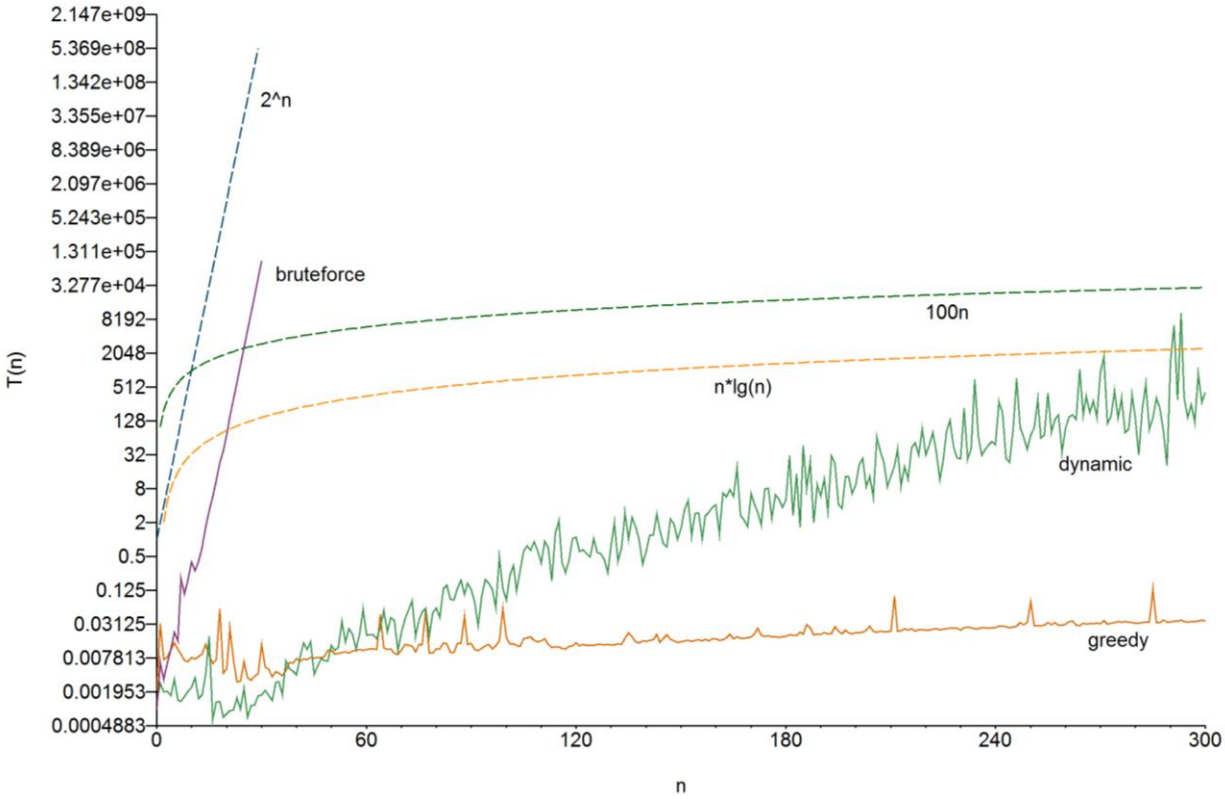

Figure 1: Experiment times for greedy, dynamic, and brute force solution algorithms for the 0/1 Knapsack Problem.

Analysis

The hypothesized results are reflected in the actual results, as graphed in Figure 1.

Figure 1 shows that there are clear differences in the time complexity performance among the three 0/1 Knapsack Problem solution algorithms. The brute force algorithm closely follows its theoretical time complexity of $O(2^n)$. Data collection was stopped at $n = 30$, as it impeded data collection for the other two algorithms.

The greedy algorithm also closely follows its theoretical time complexity of $O(n \log_2 n)$, and for all values of n after $n = 50$, drastically outperforms the other two 0/1 Knapsack Problem solution algorithms. Interestingly, the greedy algorithm is mostly consistent in the amount of time taken for all n.

As expected, Figure 1 demonstrates that the dynamic algorithm's time complexity performance sits between the brute force algorithm and the greedy algorithm. However, the testing performance does not closely match the theoretical time complexity of O(nW).

It is also worth noting that despite taking precautions to mitigate great variation and outliers in the collected data, the dynamic algorithm seems to frequently jump between ever increasing upper and lower bounds of T(n) for certain n values; that is to say that the collected data isn't as consistent as the data from the greedy algorithm.

CONCLUSION

Through the testing procedure, the hypotheses—that the Java implementation of each 0/1 Knapsack Problem solution algorithm would fit the theoretical time complexities—were mostly confirmed. Although the results from the dynamic experimentation exhibited an apparent time complexity that did not match O(nW), the overall 2^n or n $\log_2 n$ trends are clearly visible. Should this experiment be repeated, it would be best to increase the maximum size of n and to limit the number of programs that have access to the test environment's CPU in order to limit erratic cycles and mitigate the occurrence of outliers. Further experimentation should also test for each size of *n* a greater number of times in an effort to reduce variance.

References

Cormen et al.:

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