ALGORITHM FullKnapsack(S)

INPUT: a set S containing n items (input.txt file containing first row Knapsack object data in format "<weight_cap> <size_cap>" and subsequent row Item object data in format "<name> <weight> <size> <value>").

OUTPUT: a set T containing the most optimal packing of set S items (update the console with process progress and results). The output should be consistent with the Liao-Knapsack Problem (two-dimensional, 0-1, full-Knapsack).

Step 1: Generate all subsets of possible Item packings with the SubMySet algorithm.

Step 2: Check each subset to see if its valid (matches weight and size caps exactly)

Step 3: Check the remaining valid entries for the one with the highest value entry (if multiple optimal packings are found, choose the first that appears with that value entry in your set of subsets).

```
ALGORITHM SubMySet(S)


INPUT: a set S containing n items.

OUTPUT: a set T containing all subsets of S (not including the
null set)


For each i in S:

     T’ -> set of saved sets.

          T’ <- T //copy all previously found subsets to T’

     for each s in T:

          s.add(i) // add the next layer to all discovered sets.

     T’.add(i) //add the set of just i.

     T = T= T’ // doubles arr size during each iteration.
```

ANALYSIS

The output of the FullKnapSack Algorithm as implemented by this study’s implementation (FullKnapSack.java) follows the requirements very closely as seen by the following unix-CLI “diff” command output on the Project Required Output vs. the Real Implementation Output when run on the provided input file (input.txt).

```
--- sampleOutput.txt	2026-01-19 15:04:02
+++ realOutput.txt	2026-01-19 15:03:05
@@ -3,18 +3,18 @@
 Knapsack weight: 15
 Knapsack size: 20
 Number of items: 4
-Item A (weight, size, value): 5 10 2.5
-Item B (weight, size, value): 10 10 10
-Item C (weight, size, value): 2 3 1
-Item D (weight, size, value): 3 7 2
+Item A (weight, size, value): 5.0 10.0 2.5
+Item B (weight, size, value): 10.0 10.0 10.0
+Item C (weight, size, value): 2.0 3.0 1.0
+Item D (weight, size, value): 3.0 7.0 2.0
 Finding optimal packing ....
 Found a packing!
-Total weight: 15
-Total size: 20
-Total value: [value]
-Packing: Item A, Item B, ..., etc.
-Total running time: [second] seconds
-or
+Total weight: 15.00
+Total size: 20.00
+Total value: 13.00
+Packing: Item B, Item C, Item D
+Total running time: 0.055963 seconds
+or
 Finding optimal packing ....
 No packing is found!
-Total running time: [second] seconds
+Total running time: 0.038951 seconds
```

The FullKnapSack Algorithm used in this study – which utilizes a brute force algorithm – follows a time complexity of O(n^2) (i.e. exponential time complexity) due to the process of generating all possible subsets from a set. When generating the set of all possible subsets, each item in the input set must be iterated over all previously discovered subsets. This process effectively causes the number of possible subsets to be doubled with each subsequent Item object.

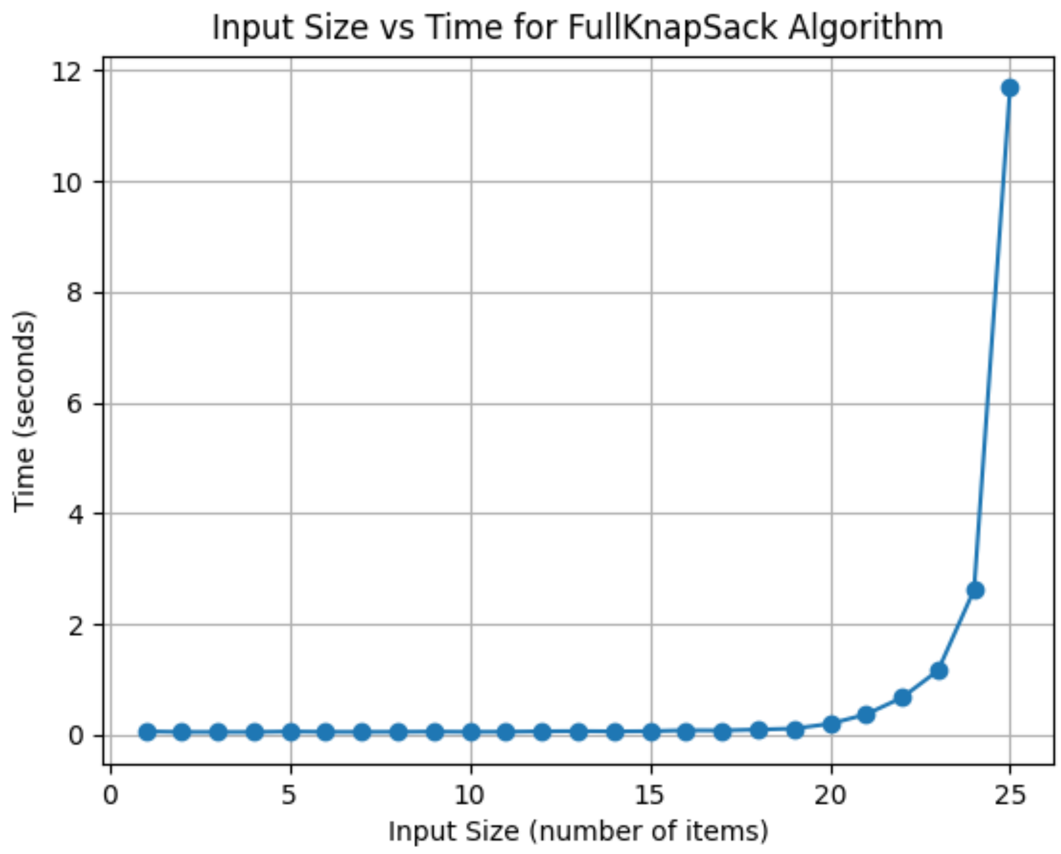

Figure 1.1: FullKnapSack Algorithm Follows an Exponential Time Complexity for Increasing Magnitudes of Input Size. These values were collected using a simple bash script (gen_graph_from_algorithm.bash) to generate a relatively large test set of data – 30 trials sets (for stable means) of input sizes of 1–25 Item objects – which was subsequently pipelined into a modified FullKnapSack (FullKnapSack4Bash.java) script for processing, and graphed with a simple python script (graph.py).

This is not an ideal time complexity as it would not be practical to find optimal packings in real world scenarios and because much faster, so-called “greedy” algorithms exist for solving these problems more efficiently. The actual running time on a set of Item objects begins to become impractical after about 25 items for the machine I used (macOS, Apple’s M2 ARM chip, 16 gigabytes of RAM, ~512 gigabytes of storage) with the implementation regularly exceeding 10 seconds of compute time, with subsequent amounts of Item objects being far too impractical for real world applications of packing (such as in a pack and ship center). Running the same implementation on my system on 26 Item objects (instead of 25) gave the error java.lang.OutOfMemoryError (for java heap space overflow) and crashed the program.

DISCLAIMERS:

LLM output was used extensively throughout code implementations in this assignment in the form of in-line code autocomplete using the Microsoft Co-Pilot extension for VSCode. Such tools were for primarily utilized for simple, straight forward implementations like constructors and helper methods (i.e. any method starting with calc, set, get, toString or like implementations, etc). The driving methods and implementations were primarily constructed by hand with some minor, straight-forward auto-completes.

Many skills utilized in the implementations (python and bash/zsh) were not taught directly in Computer Science courses but were learned through working closely with software that students in the MET department at CMU use (especially in Data and Visualization).