Ally Smith

Knapsack Report

1. Implementation

For the exhaustive approach, I implemented a solution outlined by Simon Hessner in order to obtain the powerset of the items. We begin by iterating i over $[0, 2^n)$ to account for the fact that for any set of n elements, the power set contains 2^n elements. The binary representation of i can be interpreted bit by bit to determine each combination of elements in the powerset. For example, if $i = 1001_2$, then this would mean that the 0th and 3rd elements would be in this subset of the powerset. In order to evaluate this, we iterate k from $[0, n)$ and use k as the shift amount for the bitwise shift in the expression $i \ \& \ 1 << k$. If this expression evaluates to true, that means that there is a one bit in the kth index, implying that index belongs in the subset. From here, we can add items to our subset, and when our k loop finishes, we simply add this subset to the powerset. Because this algorithm relies primarily on bitwise operations, which are very efficient to calculate, we are able to fairly quickly obtain the powerset of any set of items.

For the greedy heuristic approach, I relied on the built-in *sorted* function in Python. This function uses the Timsort algorithm, which has an average and worst case performance of $O(n\ log(n))$.

2. Asymptotic Analysis

In the exhaustive algorithm, we begin by generating the powerset of our list. This is achieved by looping from 0 to 2^n, and within each iteration we also loop from 0 to n. Therefore, the powerset is obtained in $n \times 2^n = O(2^n)$. From here, we iterate through the powerset, and within each iteration we evaluate the conditions for each item, of which there are no more than n. As a result, our total runtime for the exhaustive approach is as follows.

$$n \times 2^n + n \times 2^n = 2n\ 2^n = O(2^n)$$

In the greedy heuristic, we start by sorting our list of items first by the ratio of value to weight and then using value in the case of a tie. This is done using a built-in function in Python that utilizes the Timsort algorithm, which in the average and worse cases is $O(n\ \log n)$. From there, we simply iterate through the list of items n times. Therefore, the full execution time for the greedy heuristic approach is as follows. heuristic approach is as follows. heuristic approach is as follows.

$$n\ \log(n) + n = O(n\ \log\ n).$$

3. **Runtime results**

   (a) Exhaustive Approach

   For the exhaustive algorithm, I chose relatively small values of n, as the exponential nature of this algorithm can easily be seen just by increasing n a slight amount. I kept the change in n relatively consistent between each input.

n	**Avg. Runtime (ms)**
12	6.8
17	294.0
21	6,478.2
24	57,977.0

   (b) Heuristic Approach

   For the greedy heuristic, I had to choose much larger values of n in order to be able to measure the performance. I started at $n \approx 1500$ and increased by about 500 each time. This allows us to see a significant change across these four inputs.

n	**Avg. Runtime (ms)**
1,000	0.672
2,000	1.469
4,000	3.067
8,000	6.920

4. **Demonstration**

   From the results in the above table, we can experimentally verify the complexities of these equations.

   In the exhaustive case, we can easily see that with each increase in n our runtime increases dramatically. Based on my analysis, the algorithm is $O(2^n)$, so each value that n changes, the runtime should double. From $n = 12$ to $n = 17$, the difference in n is 5, so we would expect our runtime to increase by a factor of $2^5 = 32$. In reality, it increased by a factor of approx. 43. From $n = 17$ to $n = 21$, we would expect an increase by a factor of 16, when in reality it is about a factor of 22. And from $n = 21$ to $n = 24$, we would expect an increase by a factor of 8. In reality, it increased by a factor of approx. 9. From this experimental data, we can see that this function's runtime is exponential with relation to n, verifying the analysis from before.

   In the greedy heuristic, we expect the algorithm to run in $O(n \log n)$. From $n = 1000$ to $n = 2000$, we expect the runtime to increase by a factor of $\frac{2000 \log 2000}{1000 \log 1000} \approx 2.20$, which it did. From $n = 2000$ to $n = 4000$, we expect an increase of $\frac{4000 \log 4000}{2000 \log 2000} \approx 2.18$, when in reality it increased by a factor of approx. 2.08. In the case of $n = 4000$ to $n = 8000$, we expect an increase of ≈ 2.16, and in reality we saw an increase of 2.25. Based on this experimental data, we can see that the algorithm fairly closely follows our analysis of the algorithm.

Full Project Code

```
# Project 1 - Knapsack problem
# Ally Smith
# Sept. 9, 2021
# CSCI 406
import time

# basic item class that stores the weight, value, and their ratio
class Item:
    def __init__(self, w, v):
        self.weight = w
        self.value = v
        self.ratio = v/w

    def __eq__(self, o):
        return (self.value == o.value and self.weight == o.weight)

    def __repr__(self) -> str:
        string = "(weight=" + str(self.weight) + ", value="
            + str(self.value) + ", ratio=" + str(self.ratio) + ")"
        return string

# function to get the powerset of a given set
# source for this elegant solution:
# https://simonhessner.de/calculate-power-set-set-of-all-subsets
# -in-python-without-recursion/
def get_powerset(list):
    n = len(list)
    return [[list[k] for k in range(n) if i&1<<k] for i in range(2**n)]

# exhaustive approach, from provided pseudocode
def exhaustive(W, n, items):
    knapsack = []
    best_value = 0

    powerset = get_powerset(items)
    for subset in powerset:
        subset_value = 0
        subset_weight = 0
        for item in subset:
            subset_value += item.value
            subset_weight += item.weight
        if subset_weight <= W and subset_value > best_value:
            best_value = subset_value
            knapsack = subset
    return knapsack
```

```
# uses a built-in sorting function comparing the weight to value
# ratio primarily and the value as a secondary comparison in the
# event of a tie
def get_sorted_ratios(input_list):
    return sorted(input_list, key=lambda x: (x.ratio, x.value))

# heuristic approach from pseudocode
def heuristic(W, n, items):
    knapsack = []
    currentW = W

    items_list = get_sorted_ratios(items)
    # reverse the list (in place) so the highest ratios
    # are encounteredfirst
    items_list.reverse()

    for item in items_list:
        if item.weight <= currentW:
            knapsack.append(item)
            currentW -= item.weight

    return knapsack

# function to read in a file
# returns a tuple of (weight_capacity, num_items, items_list)
def generate_inputs(file_name):
    file_path = "./Projects/Project 1/" + file_name
    file = open(file=file_path, mode="r")
    weight_capacity = 0
    num_items = 0
    items = []

    for i, line in enumerate(file):
        if i == 0:
            weight_capacity = int(line)
        elif i == 1:
            num_items = int(line)
        else:
            values = line.split()
            weight = int(values[0])
            value = int(values[1])
            new_item = Item(w=weight, v=value)
            items.append(new_item)

    return (weight_capacity, num_items, items)
```