Ally Smith

Knapsack Report

1. 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 within the loops, 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 $O(n\ log(n))$. which has an average and worst case performance of $O(n\ log(n))$. which has an average and worst case performance of $O(n\ log(n))$.

2. 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.

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

3. Runtime results

   (a) Exhaustive Approach

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

Appendix

Full Project Code

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

# 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 = w/v

    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)

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

    return knapsack
```