William Sutanto
wsutanto@csu.fullerton.edu
Mariah Harris
mharris2021@csu.Fullerton.edu

This experiment will test the following hypotheses:

1. Exhaustive search algorithms are feasible to implement, and produce correct outputs. Algorithms with exponential running times are extremely slow, probably too slow to be of practical use.
2. A dynamic programming algorithm has a fast (i.e. polynomial) running time.

1. Analyze your exhaustive optimization algorithm code mathematically to determine its big-O efficiency class, probably $O(2^n \cdot n)$:

```
exhaustive_max_weight(C, food_items):
    n = |food_items|                    //1
    best = None                 //1
    best_weight = 0                     //1
    for subset from 0 to (2^n -1):      //2^n times
        current_subset = empty vector           //1
        subset_weight = 0                       //1
        subset_calorie = 0                      //1
        for j from 0 to n-1:                    //n times
            if (subset & (1 << j):  //2
                subset_weight += food_items[j]->weight      //1
                subset_calorie += food_items[j]->calorie//1
                current_subset.add_back(food_items[j]) //1

        if subset_calorie <= C:         //1
            if best is None or          //1
              subset_weight > best_weight:              //1
                best_weight = subset_weight             //1
                best = current_subset                   //1
    return best
```

Step count = $3 + 2^n(8+n(5)) = 3 + 8(2^n) + (2^n)(5n)$

From the step count, we can conclude that the exhaustive optimization has the time complexity of big-O efficiency class of $O(2^n * n)$. The scatterplot for the exhaustive optimization algorithm

shown below confirms this analysis, as it exhibits exponential growth (2^n) and some linear growth (*n) due to the presence of both terms in the time complexity expression. The exhaustive optimization has the time complexity of big-O efficiency class of O(2^n * n). Therefore, the empirical analysis for the exhaustive optimization algorithm aligns with the mathematical analysis.

2. **Analyze your dynamic programming algorithm code mathematically to determine its big-O efficiency class, probably $O(n^2)$:**

```
function dynamic_max_weight(foods, total_calories):
    n = size of foods //1
    total_calories_int = convert total_calories to an integer // 1
    T = create a 2-dimensional array of size (n+1) x (total_calories_int + 1) and initialize all
    elements to 0 // n*m

    for i = 1 to n: // n
        food_calorie = convert foods[i-1] calorie to an integer //2
        food_weight = get the weight of foods[i-1] // 2
        for j = 0 to total_calories_int:  // m times
            if food_calorie is less than or equal to j: // 1
                T[i][j] = maximum of (T[i-1][j], food_weight + T[i-1][j-food_calorie])//3
            else:
                T[i][j] = T[i-1][j] //2

    return T //0
```

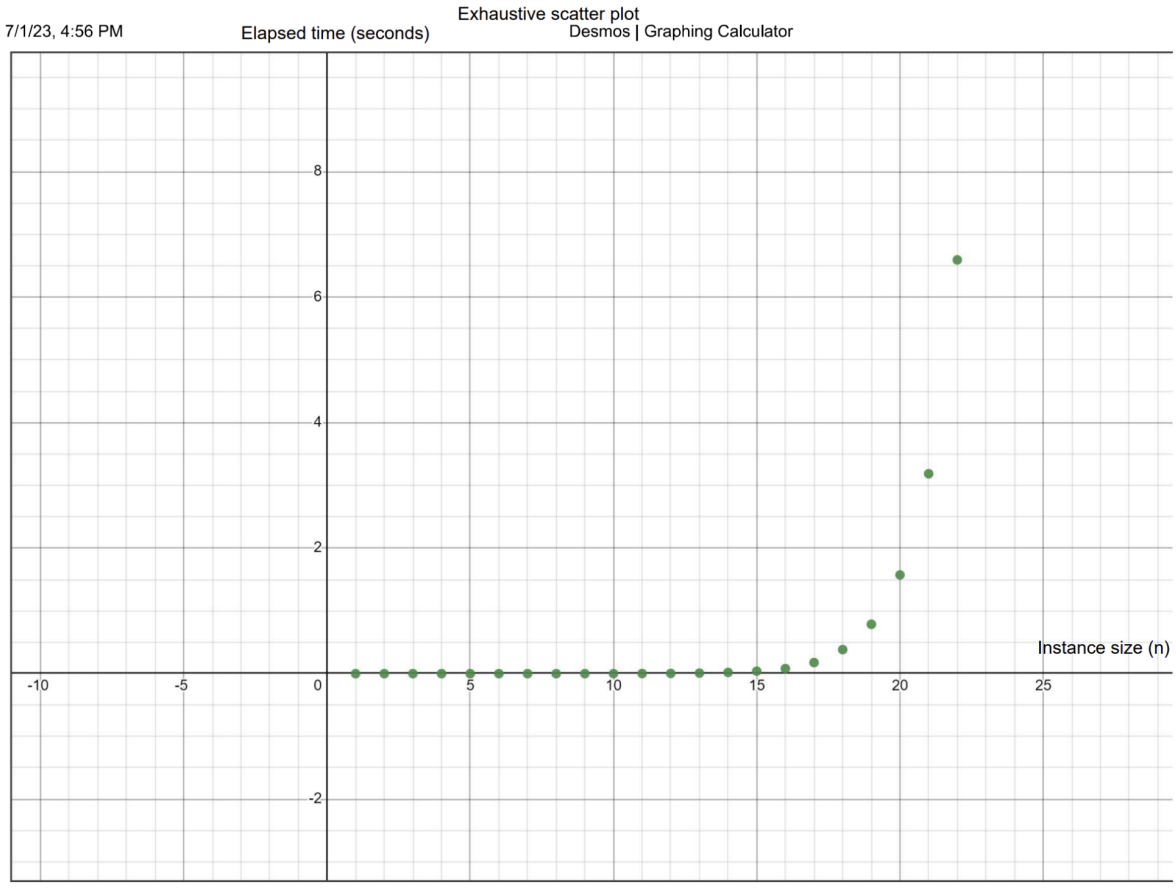
Step count = 2 +(n*m) +(((n*(4))*m(3))) = 2 + nm + 12nm = 2 + 14nm
From the step count, we can conclude that the dynamic programming algorithm has the time complexity of big-O efficiency class of O($n * m$). The scatterplot for the dynamic programming algorithm shown below confirms this analysis, as it exhibits linear growth. Therefore, the empirical analysis for the dynamic programming algorithm aligns with the mathematical analysis.

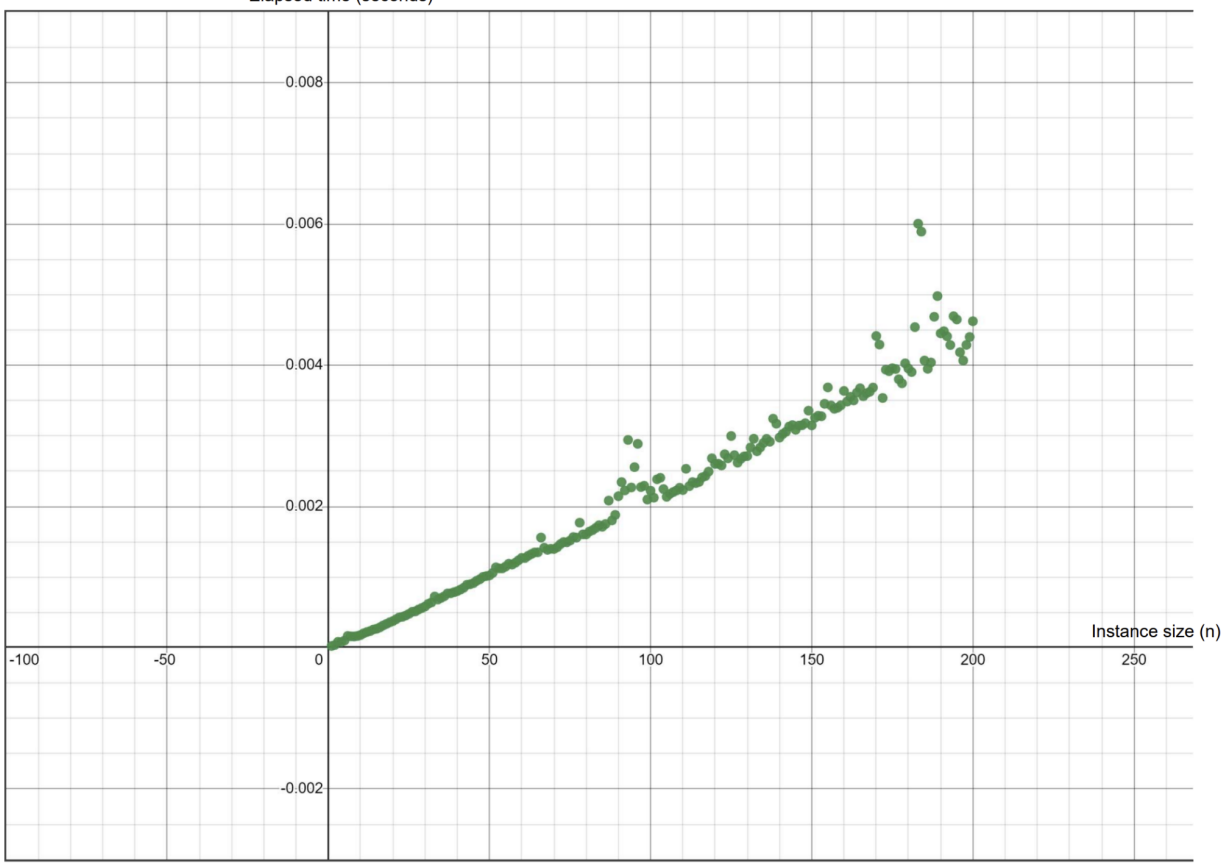
a. Is there a noticeable difference in the performance of the two algorithms? Which is faster, and by how much? Does this surprise you? The dynamic programming algorithm is significantly faster. At 20 instances, the dynamic programming algorithm has reached 0.0005 seconds, whereas the exhaustive optimization algorithm has exceeded 1.75 seconds. This is not surprising. Exhaustive optimization examines all possible solutions, which requires a

great amount of time in comparison with the dynamic programming algorithms which eliminate redundant computations and hence can efficiently solve complex problems.

b. Are your empirical analyses consistent with your mathematical analyses? Justify your answer. The scatterplot for the exhaustive optimization algorithm shown above confirms this analysis, as it exhibits exponential growth (2^n) and some linear growth (*n) due to the presence of both terms in the time complexity expression. The exhaustive optimization has the time complexity of big-O efficiency class of O(2^n * n). Therefore, the empirical analysis for the exhaustive optimization algorithm aligns with the mathematical analysis. From the step count, we can conclude that the dynamic programming algorithm has the time complexity of big-O efficiency class of O(n^2). The scatterplot for the dynamic programming algorithm shown below confirms this analysis, as it exhibits polynomial growth. Therefore, the empirical analysis for the dynamic programming algorithm aligns with the mathematical analysis.
c. Is this evidence consistent or inconsistent with hypothesis 1? Justify your answer. Exhaustive search algorithms systematically explore all possible solutions within a given search space, ensuring that no potential solution is missed. Through thorough examination of each available option, exhaustive search algorithms guarantee correct output. They are feasible for smaller search spaces. However, feasibility diminishes for larger input sizes because the search space will grow exponentially and therefore require a large amount of resources and time. When dealing with large-scale datasets in practical scenarios, exponential algorithms prove to be inefficient and unsuitable. Instead, more efficient algorithms with polynomial or sub-exponential time complexities are favored to handle larger input sizes that also offer reasonable execution times.
d. Is this evidence consistent or inconsistent with hypothesis 2? Justify your answer. Consistent; The dynamic programming algorithm is very fast. The running time of the algorithm grows linearly with the input size. As the input size increases, the algorithm's running time increases proportionally. This is generally considered efficient and faster than algorithms

with higher time complexities, such as O(n^2) or O(2^n).




Elapsed time (seconds)
0.008
0.006
0.004
0.002
-0.002
Instance size (n)
-100
-50
0
50
100
150
200
250