

Project 4 CPSC 335-11



Professor Yang

Submitted by

Ryan Dencker - dencker@csu.fullerton.edu

Hieu Nguyen - hieunguy02@csu.fullerton.edu

Github (different than the one provided)

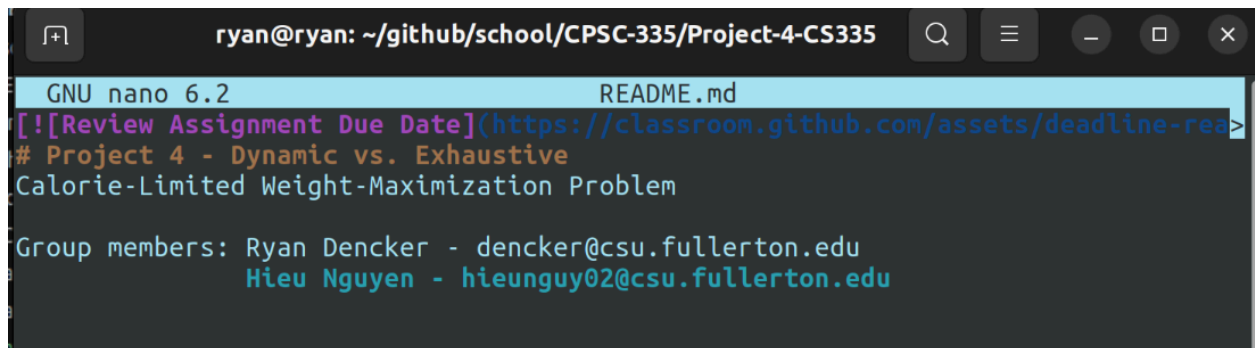
<https://github.com/ryandencker/Project-4-CS335>

Department of Computer Science,
California State University, Fullerton

Table of Contents

Table of Contents.....	2
README.md.....	3
Figure 1.....	3
Introduction.....	3
Scatter Plots.....	4
Figure 2.....	4
Figure 3.....	5
Exhaustive Search Step Count.....	5
Dynamic Search Step Count.....	6
Questions and Answers.....	7
Local Test.....	8
Figure 4.....	8
Conclusion.....	9

README.md



```
ryan@ryan: ~/github/school/CPSC-335/Project-4-CS335
GNU nano 6.2 README.md
[![Review Assignment Due Date](https://classroom.github.com/assets/deadline-read>
# Project 4 - Dynamic vs. Exhaustive
Calorie-Limited Weight-Maximization Problem

Group members: Ryan Dencker - dencker@csu.fullerton.edu
                Hieu Nguyen - hieunguy02@csu.fullerton.edu
```

Figure 1

Introduction

In this project, we will implement and compare two algorithms that solve the same problem. For this problem, we will design two separate algorithms, describe the algorithms using clear pseudocode, analyze them mathematically, implement our algorithms in C++, measure their performance in running time, compare your experimental results with the efficiency class of your algorithms, and draw conclusions. The first is the same exhaustive search algorithm with a slow (i.e. exponential) running time developed in Project 2, while the second is a dynamic programming algorithm with a fast (i.e. polynomial) running time.

Scatter Plots

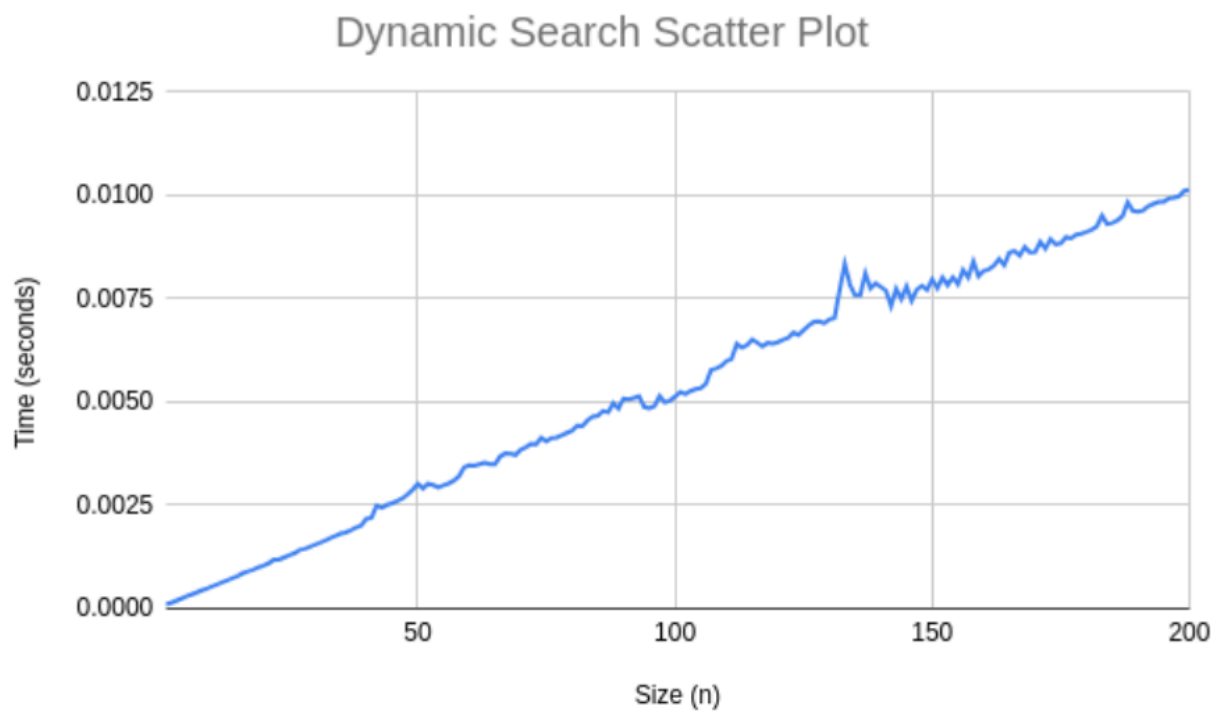


Figure 2

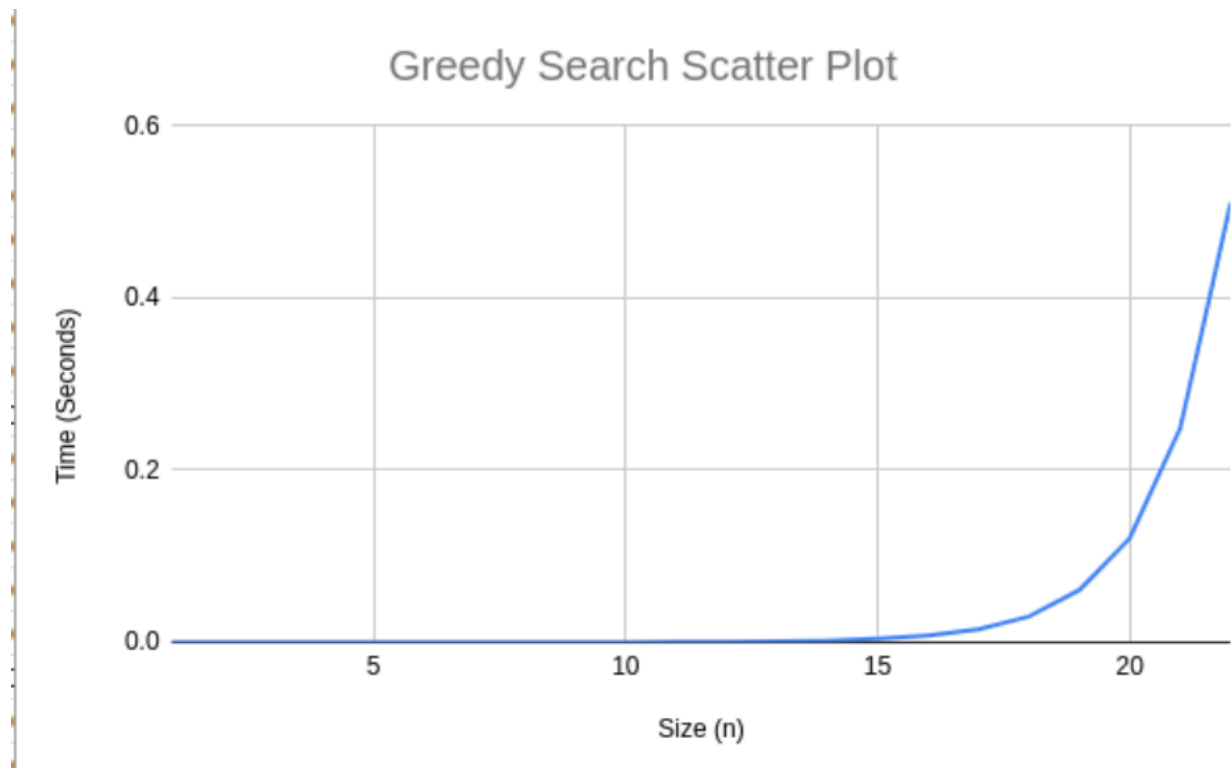


Figure 3

Exhaustive Search Step Count

Exhaustive Search Algorithm

```
std::unique_ptr<FoodVector> exhaustive_max_weight
(
    const FoodVector& foods,
    double total_calorie
)
{
    // Initialize variables
    std::unique_ptr<FoodVector> best(new FoodVector); // 0 step
    double best_weight = 0; // 1 step

    // Get the total number of food items
    size_t n = foods.size(); // 1 step

    // Iterate through all possible subsets using bit manipulation
    for (size_t i = 0; i < (1 << n); ++i) // 2^n steps
```

```

{
    // Create a subset using bit manipulation
    std::unique_ptr<FoodVector> subset(new FoodVector);
    double subset_weight = 0; // 1 step
    double subset_calorie = 0; // 1 step

    for (size_t j = 0; j < n; ++j) // n steps
    {
        if (i & (1 << j)) // 0 step
        {
            subset->push_back(foods[j]); // 1 steps
            subset_weight += foods[j]->weight(); // 2 step
            subset_calorie += foods[j]->calorie(); // 2 steps
        }
    }
    // Check if the subset satisfies the calorie constraint
    if (subset_calorie <= total_calorie && subset_weight > best_weight) // 2 steps
    {
        best = std::move(subset); // 1 step
        best_weight = subset_weight; // 1 step
    }
}
return best; // 0 step
}

```

$$SC = 0 + 1 + 1 + (2^n(1 + 1 + (n(1 + 2 + 2)) + 2 + 1 + 1) = 6 + (2^n)(2 + 5n)$$

Dynamic Search Step Count

```

std::unique_ptr<FoodVector> dynamic_max_weight
(
    const FoodVector& foods,
    double total_calories
)
{
    // Initialize a table to store the maximum weight achievable
    // Table[row][col] represents the maximum weight with the first row items and col calories
    std::vector<std::vector<double>> dp(foods.size() + 1, std::vector<double>(total_calories + 1,
    0.0)); // 0 step

    // Iterate through each food item

```

```

for (size_t i = 1; i <= foods.size(); ++i) // n steps
{
    const FoodItem* item = foods[i - 1].get(); // 1 step

    for (int j = 0; j <= total_calories; ++j) n steps
    {
        if (item->calorie() <= j) // 1 steps
        {
            dp[i][j] = std::max(dp[i - 1][j], dp[i - 1][j - item->calorie()] + item->weight()); //3 steps
        } else {

            dp[i][j] = dp[i - 1][j]; //1 step
        }
    }
}

std::unique_ptr<FoodVector> result(new FoodVector);
int remainingCalories = total_calories; //1 step

for (int i = foods.size(); i > 0 && remainingCalories > 0; --i) n steps
{
    if (dp[i][remainingCalories] != dp[i - 1][remainingCalories]) // 1 steps
    {
        result->push_back(foods[i - 1]); //1 steps
        remainingCalories -= foods[i - 1]->calorie(); // 1 steps
    }
}

return result;
}
SC = (n + 1)(n+1 + max(3,1) + (n + max(1,0))) = (n + 1)(n + 4) + (n+1) = (n+1)(n+5)

```

Questions and Answers

- a. Is there a noticeable difference in the performance of the two algorithms? Which is faster, and by how much? Does this surprise you?

There is a noticeable difference in the performance of the two algorithms. Between the dynamic search and the exhaustive search, dynamic search is considerably faster than the exhaustive search. I am not surprised by this outcome because I understand how exhaustive search works and how inefficient it is.

- b. Are your empirical analyses consistent with your mathematical analyses? Justify your answer.

My empirical analysis is consistent with my mathematical analyses because for the dynamic search, the algorithm is $O(n^2)$ and for exhaustive search, the algorithm is $O(2^n * n)$, and exhaustive search is just a lot larger compared to dynamic search.

- c. Is this evidence consistent or inconsistent with hypothesis 1? Justify your answer.

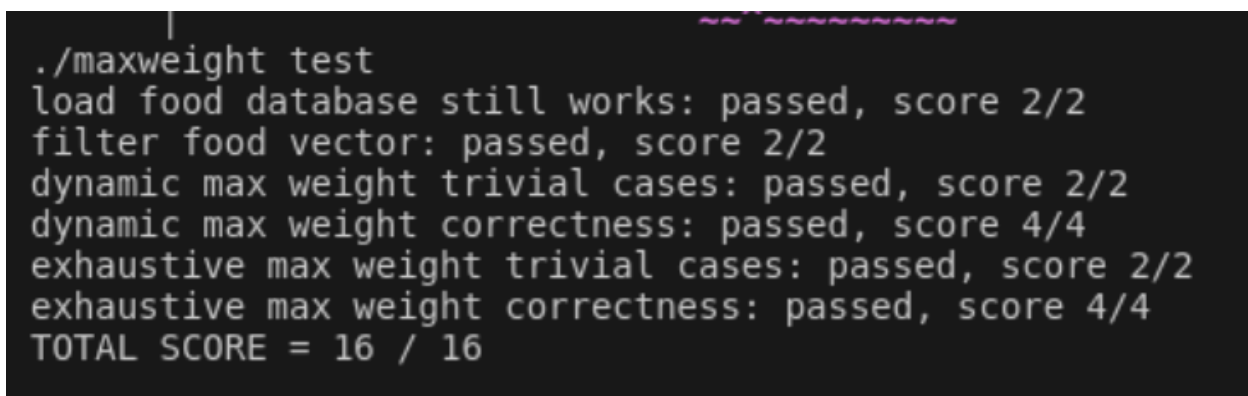
Based on the evidence we have between dynamic search and exhaustive search, it is inconsistent with hypothesis 1, because hypothesis 1 states that: Exhaustive search algorithms are feasible to implement and produce correct outputs.

Although exhaustive search does produce correct outputs, it is not very feasible because there is only a small amount of input, and it takes exponentially longer compared to dynamic search with a lot more input, however, it still takes a lot less time to produce correct outputs.

- d. Is this evidence consistent or inconsistent with hypothesis 2? Justify your answer.

Based on the evidence we have between dynamic search and exhaustive search, it is consistent with hypothesis 2, because hypothesis 2 states that: Algorithms with exponential running times are extremely slow, probably too slow to be of practical use. Exhaustive search does produce correct outputs, its run times required are just way too slow in practice where if one has very large data, it will not be feasible to run the programs with exhaustive search algorithms, and it takes exponentially longer compared to dynamic search.

Local Test

A terminal window with a dark background and light green text. The text shows the execution of a test script and its results. The script is './maxweight test'. The results show several tests passing with scores: 'load food database still works: passed, score 2/2', 'filter food vector: passed, score 2/2', 'dynamic max weight trivial cases: passed, score 2/2', 'dynamic max weight correctness: passed, score 4/4', 'exhaustive max weight trivial cases: passed, score 2/2', and 'exhaustive max weight correctness: passed, score 4/4'. The final line shows 'TOTAL SCORE = 16 / 16'.

```
./maxweight test
load food database still works: passed, score 2/2
filter food vector: passed, score 2/2
dynamic max weight trivial cases: passed, score 2/2
dynamic max weight correctness: passed, score 4/4
exhaustive max weight trivial cases: passed, score 2/2
exhaustive max weight correctness: passed, score 4/4
TOTAL SCORE = 16 / 16
```

Figure 4

Conclusion

From this project, we learned about the uses of dynamic programming by breaking our problems into smaller overlapping problems to solve them optimally. We also see that dynamic programming is way more efficient than exhaustive problem solving.