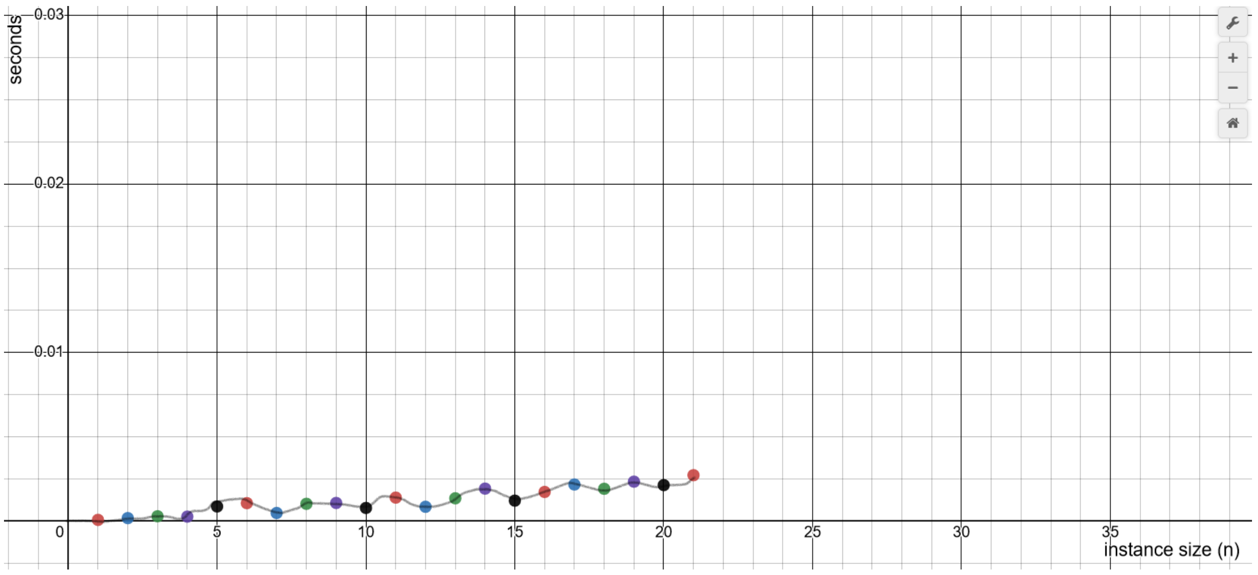
Group members:

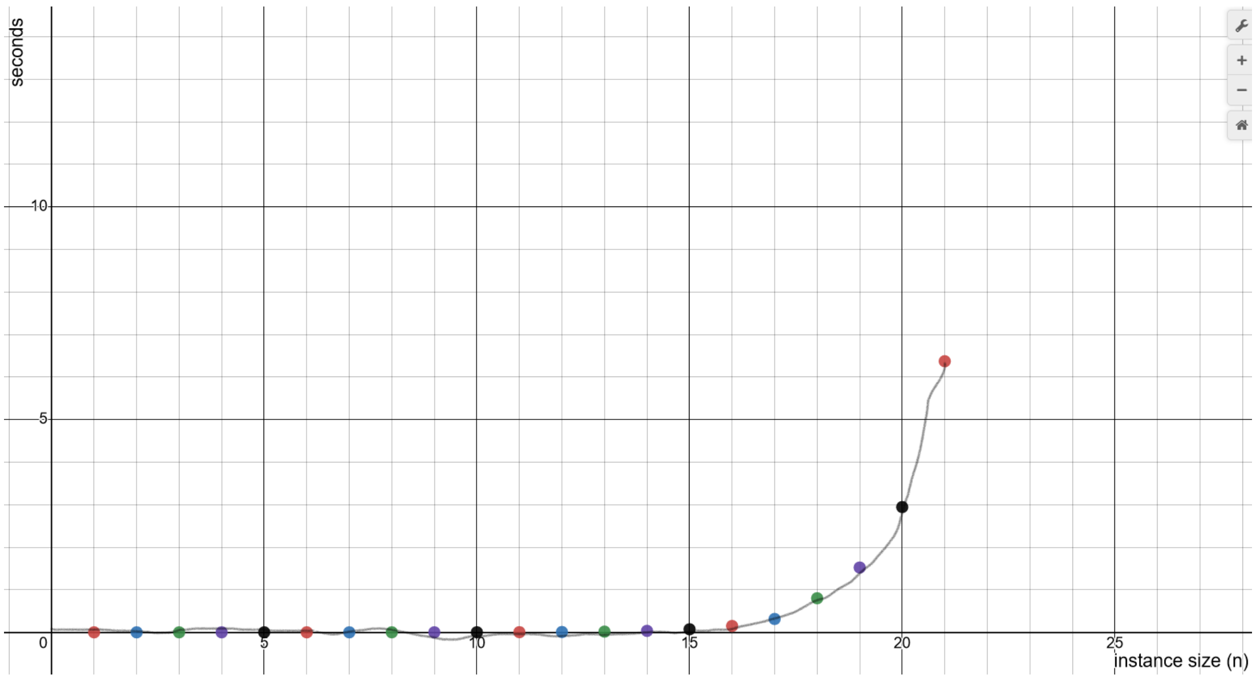
Evan Heidebrink heidebrinkevan@csu.fullerton.edu

This submission is for project 4.

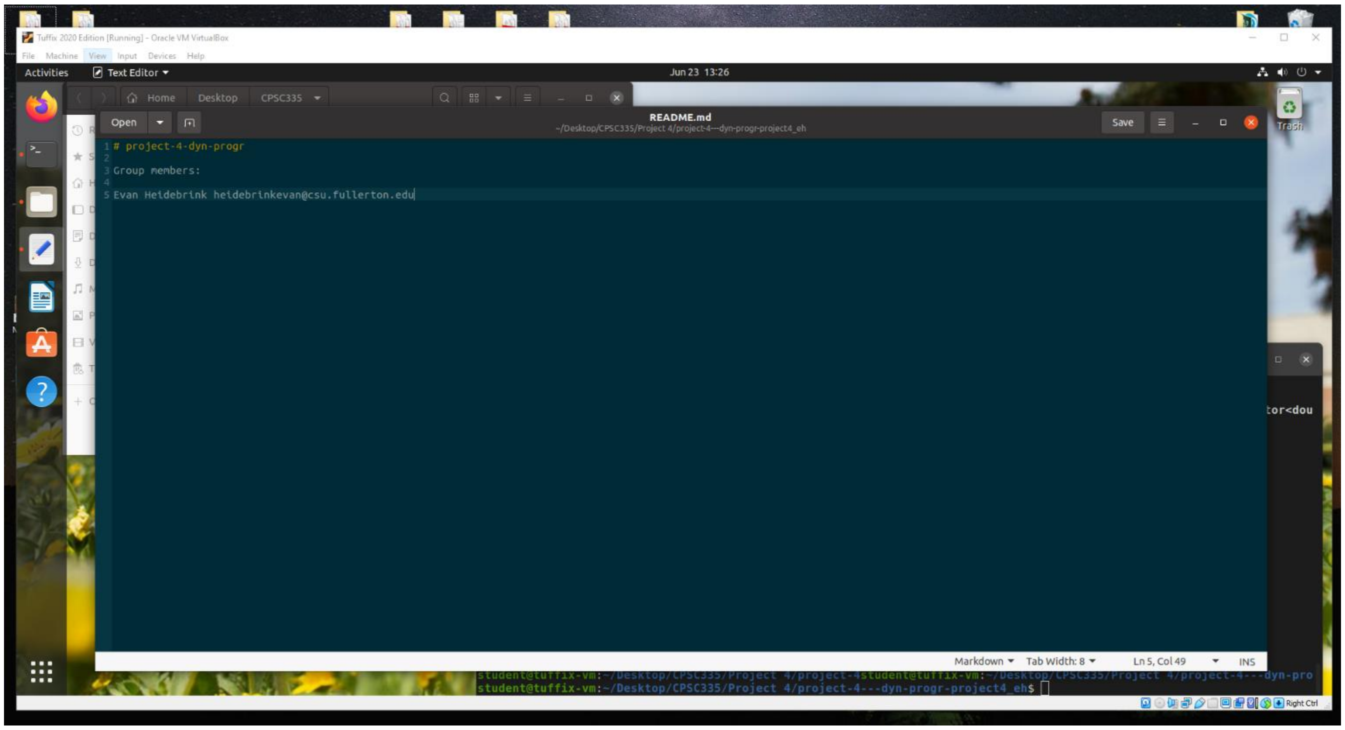
DYNAMIC SCATTER PLOT



EXHAUSTIVE SCATTER PLOT



TUFFIX README



```
# project-4-dyn-progr

Group members:

Evan Heidebrink heidebrinkevan@csu.fullerton.edu
```

```
student@tuffix-vm:~/Desktop/CPSC335/Project 4/project-4student@tuffix-vm:~/Desktop/CPSC335/Project 4/project-4---dyn-pro
student@tuffix-vm:~/Desktop/CPSC335/Project 4/project-4---dyn-progr-project4_eh$
```

<u>Mathematical Analysis</u>:

<u>dynamic_max_weight</u>

std::vector<std::vector<double>> cache; // 0

std::unique_ptr<FoodVector> items(new FoodVector(foods)); // 1

std::unique_ptr<FoodVector> best(new FoodVector); // 1

int maxItemIndex = items->size(); // 3

int maxCalories = int(total_calories); // 2

std::vector<double> weights = { 0 }; // 1

std::vector<int> calories = { 0 }; // 1

for (const auto& element : foods) // n times

{

```
        weights.push_back(element->weight()); // 2
        calories.push_back(element->calorie()); // 2
}

cache.resize(maxItemIndex + 1); // 2
for (auto& element : cache) // n times
        element.assign(maxCalories + 1, 0); // 2



for (int i = 1; i <= maxItemIndex; i++) // n times
        for (int j = 0; j <= maxCalories; j++) // n times
        {
                if (j - calories[i] < 0) // 2
                        cache[i][j] = cache[i - 1][j]; // 2
                else
                        if (cache[i - 1][j - calories[i]] + weights[i] > cache[i - 1][j]) // 5
                                cache[i][j] = cache[i - 1][j - calories[i]] + weights[i]; // 4
                        else
                                cache[i][j] = cache[i - 1][j]; // 2
        }

int item = maxItemIndex; // 1
int calorieLimit = maxCalories; // 1
while (item != 0 && calorieLimit != 0) // n times
{
        if (cache[item][calorieLimit] == cache[item - 1][calorieLimit]) // 2
                item--; // 1
        else
```

```
    {
        best->push_back(foods[item - 1]); // 3
        calorieLimit -= calories[item]; // 1
        item--; // 1
    }
}
return best; // 0
```

$$\begin{aligned}\text{Step count} &= 9 + 4n + 2 + 2n + n(n[2 + \max(2, [5 + \max(4, 2)])]) + 2 + n(2 + \max(1, 5)) \\ &= 13 + 6n + n(n[2 + \max(2, [9])]) + n(7) \\ &= 13 + 6n + n(n[11]) + 7n \\ &= 11n^2 + 13n + 13 \\ &= T(n)\end{aligned}$$

$T(n) \in O(n^2)$

Proof:

$$\begin{aligned}\lim_{n \to \infty} T(n)/n^2 &= \lim_{n \to \infty} [11n^2 + 13n + 13]/n^2 \\ &= \lim_{n \to \infty} [11 + 13/n + 13/n^2] \\ &= 11 >= 0 \text{ and a constant, so } T(n) \in O(n^2). \ \square\end{aligned}$$

exhaustive_max_weight

```
int n = foods.size(); // 1
double bestWeight = 0.0; // 1
std::unique_ptr<FoodVector> result(new FoodVector); // 1
for (uint64_t bits = 0; bits <= std::pow(2, n) - 1; bits++) // 2^n times
{
    std::unique_ptr<FoodVector> candidate(new FoodVector); // 1
    for (int j = 0; j <= n - 1; j++) // n times
    {
```

```
                if (((bits >> j) & 1) == 1) // 3
                {
                        candidate->push_back(foods[j]); // 1
                }
        }
        double candidateCalories = 0.0; // 1
        double candidateWeight = 0.0; // 1
        for (auto iter = candidate->begin(); iter != candidate->end(); iter++) // n times
        {
                candidateCalories += (*iter)->calorie(); // 1
                candidateWeight += (*iter)->weight(); // 1
        }
        if (candidateCalories <= total_calorie) // 1
                if (result->empty() || candidateWeight > bestWeight) // 2
                {
                        result->clear(); // 1
                        for (auto iter = candidate->begin(); iter != candidate->end(); iter++) // n
times
                        {
                                result->push_back(*iter); // 1
                        }

                        bestWeight = 0.0; // 1
                        for (auto iter = result->begin(); iter != result->end(); iter++) // n times
                        {
                                bestWeight += (*iter)->weight(); // 1
                        }
                }
}
```

```
return result; // 0
```

Step count $= 3 + (2^n) * (1 + (n) * (3 + \max(1, 0)) + 2 + (n) * (2) + [1 + \max([2 + \max(1 + (n) * (1) + 1 + (n) * (1)), 0)], 0)])$

$= 3 + (2^n) * (3 + (n) * (4) + (n) * (2) + [1 + 4 + (n) * (1) + (n) * (1)])$

$= 3 + (2^n) * (3 + 6n + [5 + 2n])$

$= 3 + (2^n) * (8 + 8n)$

$= (2^n) * (8n) + (2^n) * (8) + 3$

$= T(n)$

$T(n) \in O(2^n * n)$

Proof:

$\lim_{n \to \infty} T(n)/2^n * n = \lim_{n \to \infty} [(2^n) * (8n) + (2^n) * (8) + 3]/2^n * n$

$= \lim_{n \to \infty} [(2^n) * (8n) + (2^n) * (8)]/2^n * n + \lim_{n \to \infty} 3/2^n * n$

$= \lim_{n \to \infty} [(2^n) * (8n + 8)]/2^n * n$

$= \lim_{n \to \infty} (8n + 8)/n$

$= \lim_{n \to \infty} (8)/1$

$= 8 >= 0$ and a constant, so $T(n) \in O(2^n * n)$. □

Questions

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, which does not surprise me because the dynamic algorithm is $O(n^2)$ while the exhaustive algorithm is $O(2^n * n)$. The dynamic algorithm is much faster by a wide margin. By n = 21, the dynamic takes only ~0.0027 seconds to execute, while the exhaustive takes ~6.3729 seconds.

b. Are your empirical analyses consistent with your mathematical analyses? Justify your answer.
My empirical analysis is consistent with my mathematical analysis because I concluded mathematically that the dynamic algorithm is $O(n^2)$ while the exhaustive algorithm is slower at $O(2^n * n)$. My empirical analysis showed that the exhaustive algorithm is much slower by recording the execution time for various instance sizes.

c. Is this evidence consistent or inconsistent with hypothesis 1?

This evidence is consistent with hypothesis one, which states that exhaustive algorithms can be implemented and produce correct results. The pseudocode and implementation do just that, although inefficiently.

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

This evidence is consistent with hypothesis two, which states that exponential algorithms are extremely slow and impractical. The empirical analysis justifies this statement, since by just n = 21 instance size, the exhaustive exponential algorithm already takes ~6.3729 seconds, and will grow significantly from there.