Andrew Lau - andrewlau2019@csu.fullerton.edu
Tommy Le - tommyle@csu.fullerton.edu

Project 2: Greedy vs. Exhaustive

Greedy Algorithm Code:

```
std::unique_ptr<CargoVector> greedy_max_weight
(
        const CargoVector& goods,
        double total_volume
)
{
std::unique_ptr<CargoVector> todo(new CargoVector(goods)); // 1tu
std::unique_ptr<CargoVector> result(new CargoVector); //1 tu
size_t result_volume = 0; //1 tu

while(!todo->empty()) { //n tu
        double ratio = 0; //1 tu
        int index = 0; //1 tu
        int volume = 0; //1 tu
        for(size_t i = 0; i < todo->size(); i++) {  // (n-1) + 1/ 1 -1  + 1= n tu
                if(ratio < todo->at(i)->weight() / todo->at(i)->volume()) { // 2 tu
                        ratio = todo->at(i)->weight() / todo->at(i)->volume(); // 2 tu
                        index = i; // 1 tu
                        volume = todo->at(i)->volume(); // 1 tu
```

```
            }
        }
        if((result_volume + volume) <= total_volume) { //2 tu
            result->push_back(todo->at(index)); //1 tu
            result_volume += volume; //2 tu
        }

        todo->erase(todo->begin() + index); //2 tu
}

return result;
}
```

Step Count of Greedy Algorithm:

S.C. = 3 + (n * (1 + 1 + 1 + n * (2 + 2 + 1 + 1))) + 2 + 1 + 2 + 2
= 3 + (n * (3 + n * 6) +7
= 3 + (n * (6n+3)) + 7
= 3 + (6n^2 +3n) + 7
= 6n^2 +3n + 10

Mathematical Analysis of the Greedy Algorithm:

Proof by definition: Given f(n), show that f(n) ∈ O(g(n)).
f(n) ∈ O(g(n)) if there exists two constants c > 0 and $n_o \geq 0$.
such that f(n) ≤ c * g(n) ∀ n ≥ n_o.

Let's say c = 19 and n_o = 1,
Dominating term: 6n^2
6n^2 +3n + 10 ∈ O(n^2)
f(n) = 6n^2 +3n + 10, g(n) = n^2
6(1)^2 + 3(1) + 10 ≤ 19 * (1)^2 ∀ n ≥ 1
19 ≤ 19

Since the condition holds true, we can safely say that $6n^2 + 3n + 10 \in O(n^2)$.

Exhaustive Optimization Code:

```
std::unique_ptr<CargoVector> exhaustive_max_weight
(
        const CargoVector& goods,
        double total_volume
)
{

std::unique_ptr<CargoVector> best(new CargoVector); // 1 tu
int n = goods.size(); //1 tu
double candidateTotalVolume = 0; //1 tu
double candidateTotalWeight = 0; //1 tu
double bestWeight;
for (int i = 0; i <= (pow(2,n)-1); i++) {  //(2^n-1) - 0/1 + 1 = 2^n tu
        std::unique_ptr<CargoVector> candidate(new CargoVector); // 1 tu
        for (int j = 0; j <= (n-1); j++) { //(n-1)/1 + 1 = n tu
                if(((i >> j) & 1) == 1) { // 3 tu
                        candidate->push_back(goods[j]); //2 tu
                }
        }
        sum_cargo_vector(*candidate, candidateTotalVolume, candidateTotalWeight); //2n + 3 tu
        if(candidateTotalVolume <= total_volume) { // 1 tu
```

```
                if(best == NULL || candidateTotalWeight > bestWeight) { //3 tu

                        *best = *candidate; //3 tu

                        bestWeight = candidateTotalWeight; // 1 tu

                }

        }

}

return best;

}
```

- This is the sum_cargo_vector function called in the exhaustive optimization algorithm. The step count for this function is 2n + 2.

```
void sum_cargo_vector
(
        const CargoVector& goods,
        double& total_volume,
        double& total_weight
)
{
        total_volume = total_weight = 0; // 2 tu
        for (auto& item : goods) // n tu
        {
                total_volume += item->volume(); // 1 tu
                total_weight += item->weight(); // 1 tu
        }
}
```

Step Count of Exhaustive Optimization:

S.C = 1 + 1 +1 + 1 + 2^n * ((1 + n * (3 + 2)) + (2n +3) + 1 + 3 + 3 + 1)
= 4 + 2^n * (1 + 5n + 2n+ 3 + 8)
= 4 + 2^n (7n + 12)
= 4 + 2^n (7n + 12)
= 2^n(7n) + 2^n(12) + 4

Mathematical Analysis of the Exhaustive Search:

Proof by definition: Given f(n), show that f(n) ∈ O(g(n)).

f(n) ∈ O(g(n)) if there exists two constants c > 0 and $n_o \geq 0$.

such that f(n) ≤ c * g(n) ∀ n ≥ n_o.

Let's say c = 21 and n_o = 1,

Dominating term: 2^n(7n)

2^n(7n) + 2^n(12) + 4 ∈ O(2^n(n))

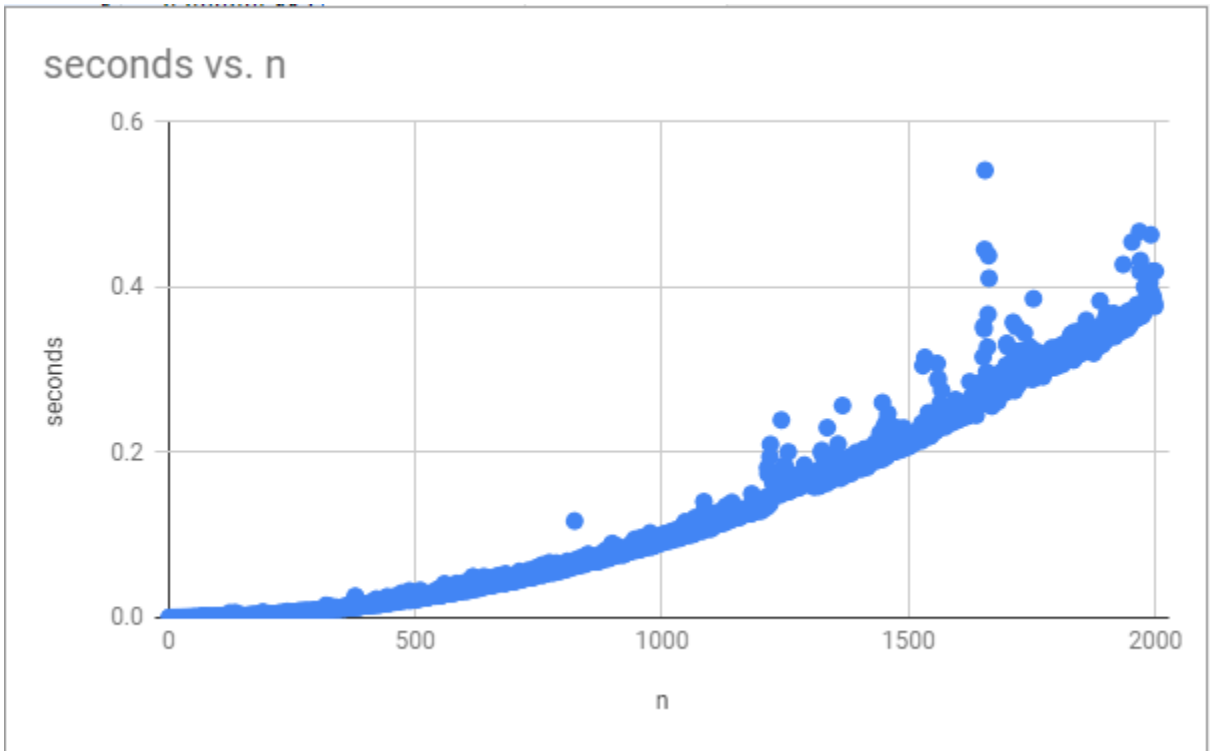
f(n) = 2^n(7n) + 2^n(12) + 4, g(n) = 2^n(n)

2^(1)(7(1)) + 2^(1)(12) + 4 ≤ 21 * 2^(1)(1) ∀ n ≥ 1

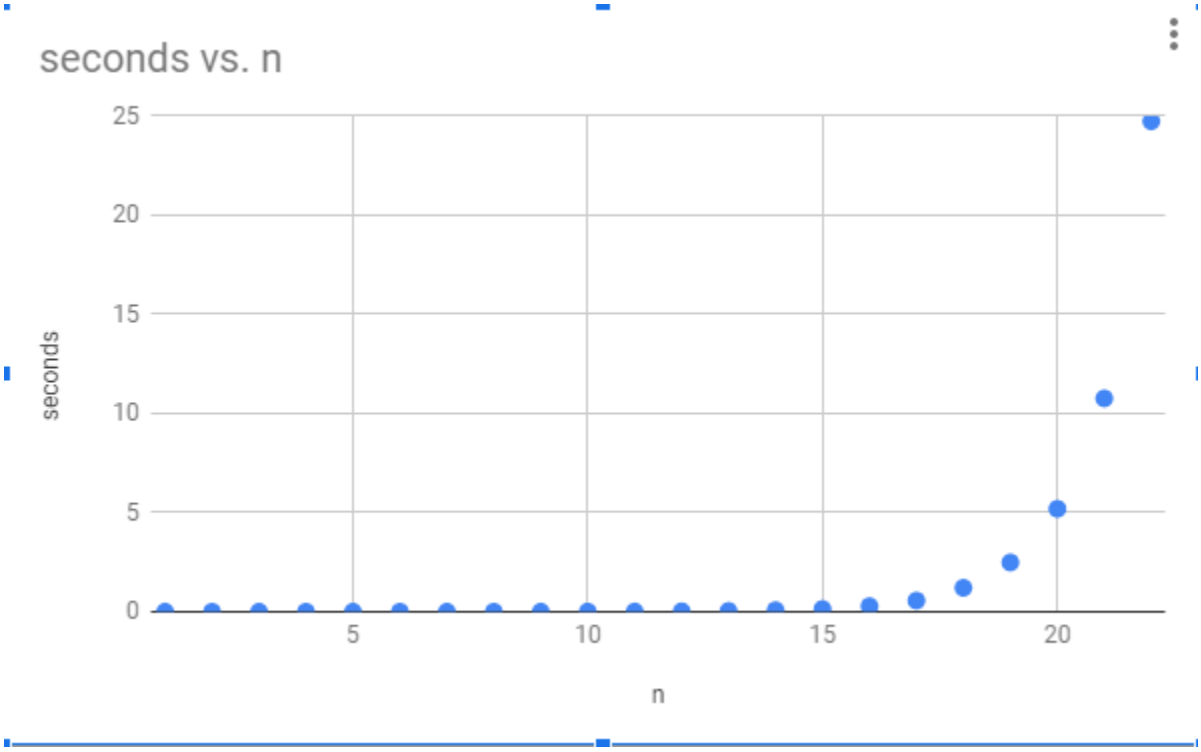
42 ≤ 42

Since the condition holds true, we can safely say that 2^n(7n) + 2^n(12) + 4 ∈ O(2^n(n)).

Greedy.csv



Exhaustive.csv



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?

A: There was a noticeable difference in the performance of the two algorithms. The Greedy algorithm is faster than the exhaustive optimization algorithm. The seconds on the Greedy algorithm are faster than the ones on the exhaustive optimization by a lot. It did surprise me because we expected the exhaustive optimization algorithm.

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

B: Our empirical analyses are consistent with our mathematical analyses. This is because the greedy algorithm is a quadratic function that belongs to the O(n^2) time efficiency class and the exhaustive optimization is an exponential function that belongs to the O(2^n(n)) time efficiency class. This is demonstrated in the scatter plots above (greedy.csv and exhaustive.csv).

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

A: This evidence is consistent with hypothesis 1. We believe this because when the exhaustive.csv file was created, it seems to have been implemented and shows the most optimal path.

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

A: This evidence is consistent with hypothesis 2. This is because when the files were being complied with, it took a while for the CSV files to be created. This is because there was so much data that finding the most optimal way will take some time for the scatterplot to be created.