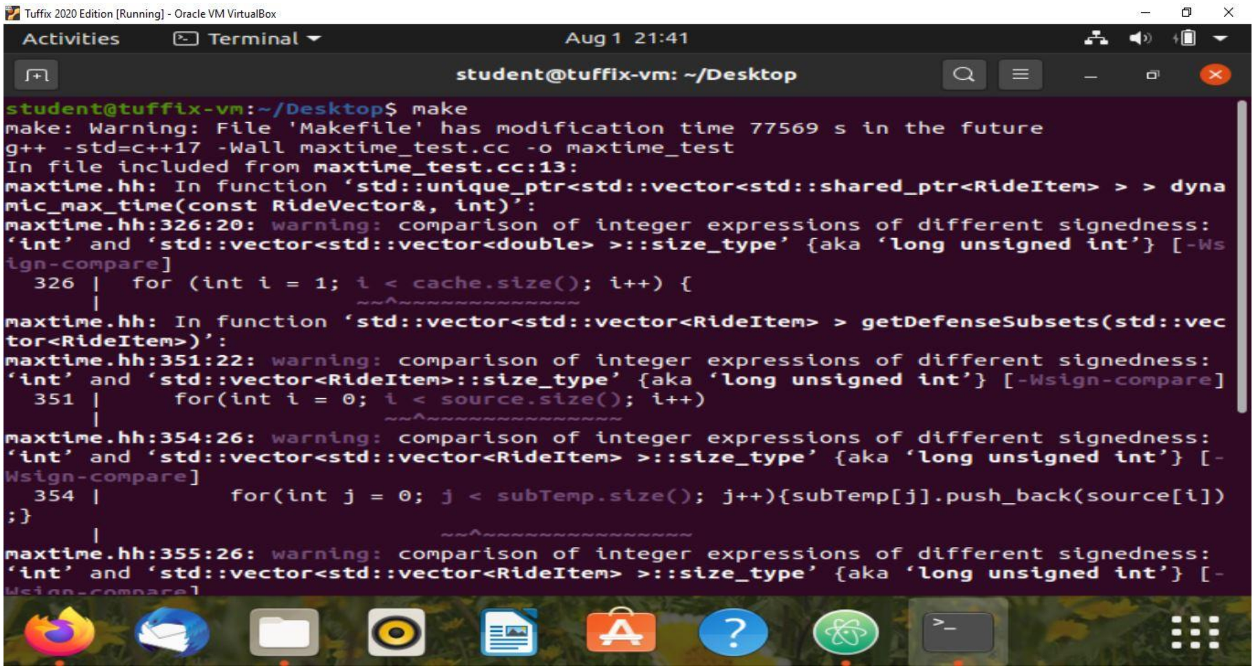
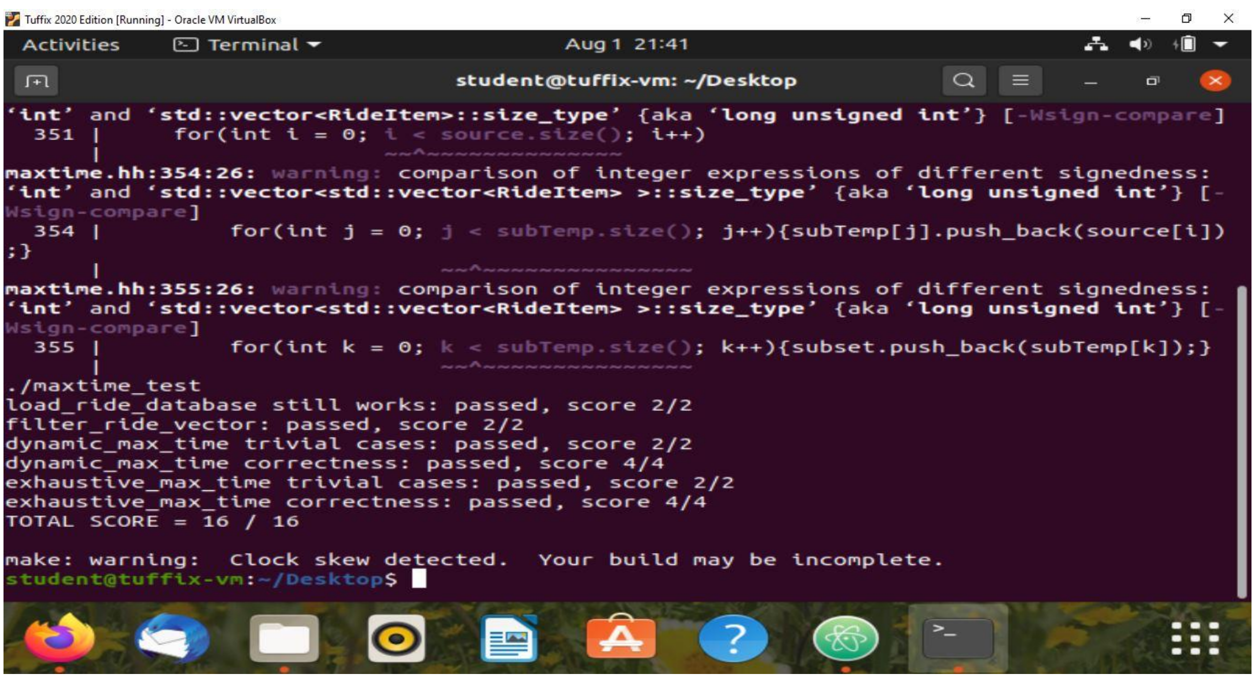
Project 4: Dynamic versus Exhaustive

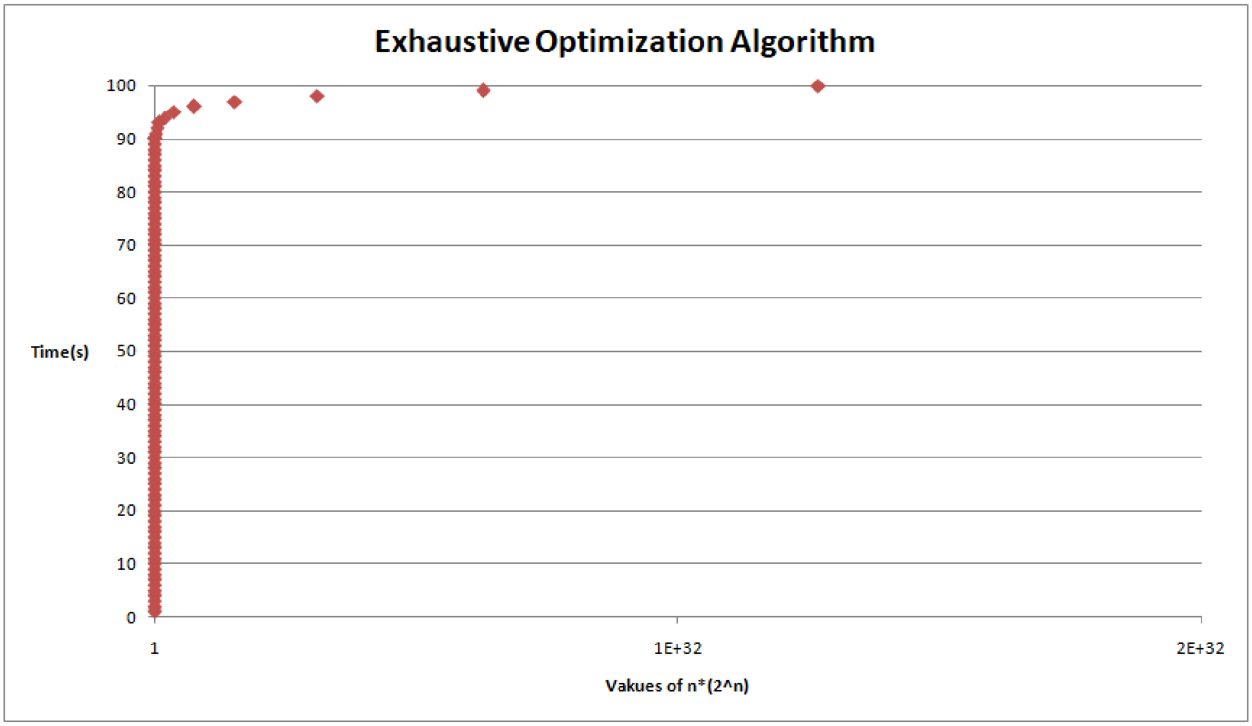
Mike Sim msim@csu.fulleton.edu
Steven Tran transteven@csu.fullerton.edu
CPSC 335-02 Algorithm Engineering
Date 7 August, 2021

Screenshots of code running “Makefile” on Tuffix is shown below

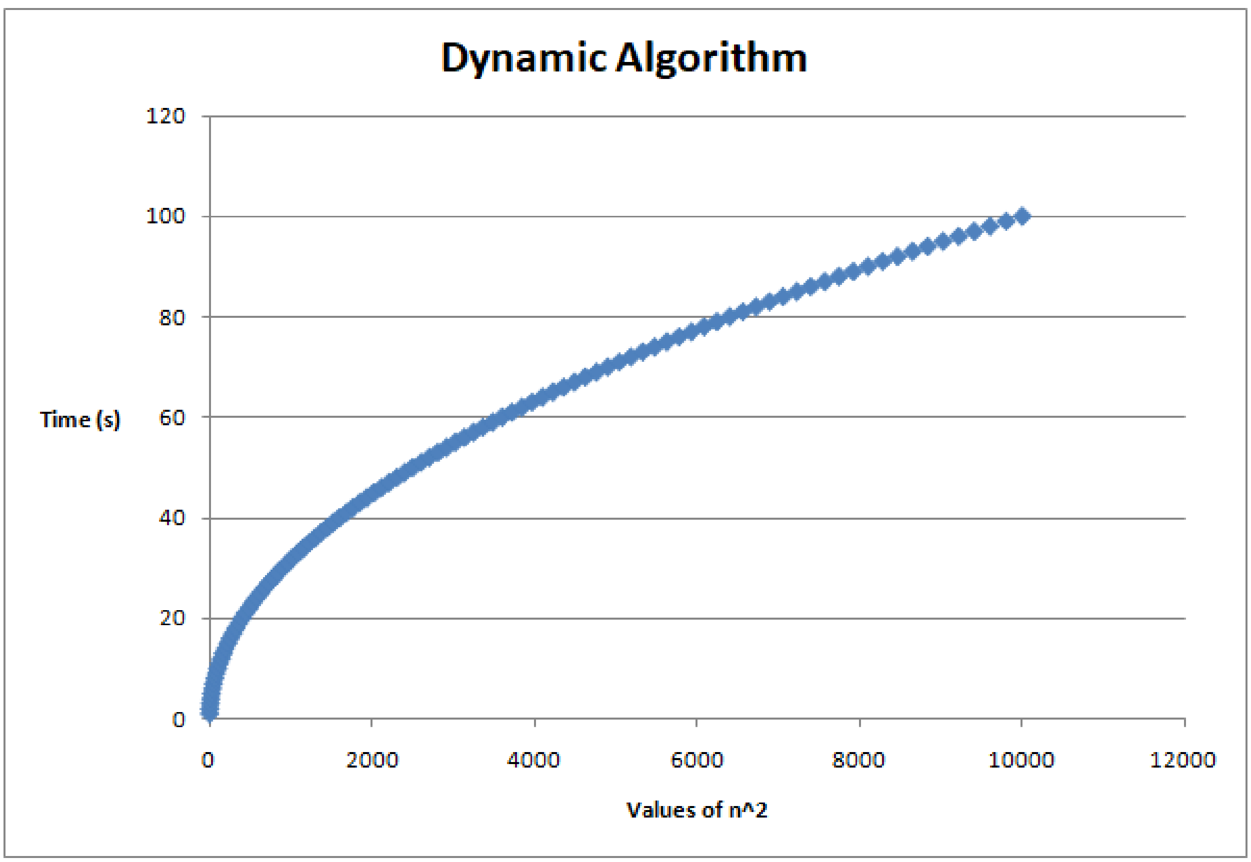




Scatterplot of Exhaustive Optimization is below



Scatter plot of Dynamic Algorithm is below



As seen below, the empirical data is consistent with the mathematical analysis.

Dynamic Programming Algorithm

```
std::unique_ptr<RideVector> dynamic_max_time
(
	const RideVector &rides,
	int total_cost
)
{
	size_t total_rides = rides.size();

	// Creates and initializing the cache with zeros
	std::vector<std::vector<double>> cache;
	for (int i = 0, n = total_rides + 1; i < n; i++) {
		std::vector<double> subVector;
		for (int j = 0; j < total_cost + 1; j++) {
			subVector.push_back(0);
		}
		cache.push_back(subVector);
	}


	// Filling the cache
	for (int i = 1; i < cache.size(); i++) {
		std::shared_ptr<RideItem> currItem = rides[i - 1];
		for (int j = 0; j < total_cost + 1; j++) {
			double value = 0;

			if (j >= currItem->cost()) {
				value = cache[i - 1][j - currItem->cost()] + currItem->defense();
			}
			cache[i][j] = std::max(value, cache[i - 1][j]);
		}
	}

	// Solving the for best RideItems with the help of cache
	RideVector resultSet;
	recursive_solver(rides, cache, resultSet, total_cost, total_rides);

	return std::unique_ptr<RideVector>(new RideVector(resultSet));
}

std::vector<std::vector<RideItem>> getDefenseSubsets(std::vector<RideItem> source)
{
```

```
        std::vector<std::vector<RideItem>> subset, subTemp;

        std::vector<RideItem> temp;
        subset.push_back(temp);

        for (int i = 0; i < source.size(); i++)
        {
                subTemp = subset;
                for (int j = 0; j < subTemp.size(); j++) { subTemp[j].push_back(source[i]); }
                for (int k = 0; k < subTemp.size(); k++) { subset.push_back(subTemp[k]); }
        }
        return subset;
}
```

```
std::unique_ptr<RideVector> dynamic_max_time          — 1
(
    const RideVector &rides,                          — 1
    int total_cost                                    — 1
)
{

    size_t total_rides = rides.size();                — 1

    // Creates and initializing the cache with zeros
    std::vector<std::vector<double>> cache;
    for (int i = 0, n = total_rides + 1; i < n; i++) {          n
        std::vector<double> subVector;
        for (int j = 0; j < total_cost + 1; j++) {          n
            subVector.push_back(0);
        }
        cache.push_back(subVector);
    }


    // Filling the cache
    for (int i = 1; i < cache.size(); i++) {                                  n
        std::shared_ptr<RideItem> currItem = rides[i - 1];
        for (int j = 0; j < total_cost + 1; j++) {                            n
            double value = 0;

            if (j >= currItem->cost()) {                                      — 2
                value = cache[i - 1][j - currItem->cost()] + currItem->defense();   — 6
            }
            cache[i][j] = std::max(value, cache[i - 1][j]);                   — 2
        }
    }
```

$\Rightarrow 4 + n^2 + 10n^2$

$\in O(11n^2 + 4)$ trivial

$= O(11n^2)$ dominated

$= O(n^2)$ drop constant

$\therefore O(n^2)$

The Dynamic Programming Algorithm takes **O(n^2)** time

Exhaustive Optimization Algorithm

```
std::unique_ptr<RideVector> exhaustive_max_time
(
	const RideVector &rides,
	double total_cost
)
{
	size_t n = rides.size();
	// Returns nullptr if the rides' count >= 64
	if (n >= 64) { return nullptr; }

	std::unique_ptr<RideVector> best(nullptr);

	for (uint64_t bits = 0; bits <= (pow(2, n) - 1); bits++) {
		std::unique_ptr<RideVector> candidate(new RideVector());
		for (uint64_t j = 0; j <= (n - 1); j++) {
			if (((bits >> j) & 1) == 1) {
				candidate->push_back(rides[j]);
			}
		}

		// tc -> total_count, and tt -> total_time
		double candidate_tc, candidate_tt, best_tc, best_tt;
		sum_ride_vector(*candidate, candidate_tc, candidate_tt);

		if (best != nullptr) {
			sum_ride_vector(*best, best_tc, best_tt);
		}

		if (candidate_tc <= total_cost) {
			if (best == nullptr || (candidate_tt > best_tt)) {
				best = std::move(candidate);
			}
		}
	}
	return best;
}
```

Exhaustive Algorithm

```
std::unique_ptr<RideVector> exhaustive_max_time
(
	const RideVector &rides,
	double total_cost
)
{
	size_t n = rides.size();
	// Returns nullptr if the rides' count >= 64
	if (n >= 64) { return nullptr; }

	std::unique_ptr<RideVector> best(nullptr);

	for (uint64_t bits = 0; bits <= (pow(2, n) - 1); bits++) {
		std::unique_ptr<RideVector> candidate(new RideVector());
		for (uint64_t j = 0; j <= (n - 1); j++) {
			if (((bits >> j) & 1) == 1) {
				candidate->push_back(rides[j]);
			}
		}

		// tc -> total_count, and tt -> total_time
		double candidate_tc, candidate_tt, best_tc, best_tt;
		sum_ride_vector(*candidate, candidate_tc, candidate_tt);

		if (best != nullptr) {
			sum_ride_vector(*best, best_tc, best_tt);
		}

		if (candidate_tc <= total_cost) {
			if (best == nullptr || (candidate_tt > best_tt)) {
				best = std::move(candidate);
			}
		}
	}
	return best;
}
```

1, 1, 1, 1, 1 → 5

$\Rightarrow (\underbrace{2^n}_{\text{final val}} - \underbrace{0}_{\text{initial val}} + 1)$

3 → $3(\underbrace{n-\cancel{1}}_{\text{f.V}} - \underbrace{0}_{\text{I.V}} \cancel{+1}) = 3n$

1, 1, 2, 1 → 5

$\Rightarrow 5 + (2^n+1)(3n+5)$

$= 5 + 2^n 3n + 3n + 5\cdot 2^n + 5$

$= 2^n 3n + 2^n\cdot 5 + 3n + 10$

$\Rightarrow \in O(2^n 3n)$ trivial

$= O(2^n \cdot n)$ drop constant

$\boxed{\therefore O(2^n \cdot n)}$

The Exhaustive Optimization Algorithm takes **O(n*2^n)** time

Questions & Answer

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

Yes. The Dynamic algorithm (O(n^2)) are faster than The exhaustive algorithm (O(n*2^n).
So through the mathematical step count, it proves from our code.

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

Yes. At the lecture, we learned that the dynamic algorithm has a relatively faster runtime but it doesn't always guarantee the optimal solution.
The exhaustive algorithm on the other hand is relatively slower but it always returns the optimal solution.
With our mathematical analyses, we prove that the Greedy algorithm is faster than the Exhaustive algorithm.

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

The evidence does conclude that the exhaustive algorithm is feasible to implement and does produce correct outputs as we expected. This can be seen with the code along with proven results.

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

The evidence is consistent with hypothesis 2 because the graph and the mathematical analysis proves that the exhaustive algorithm does produce correct results, with the caveat of the run time being substantially longer than the greedy algorithm.