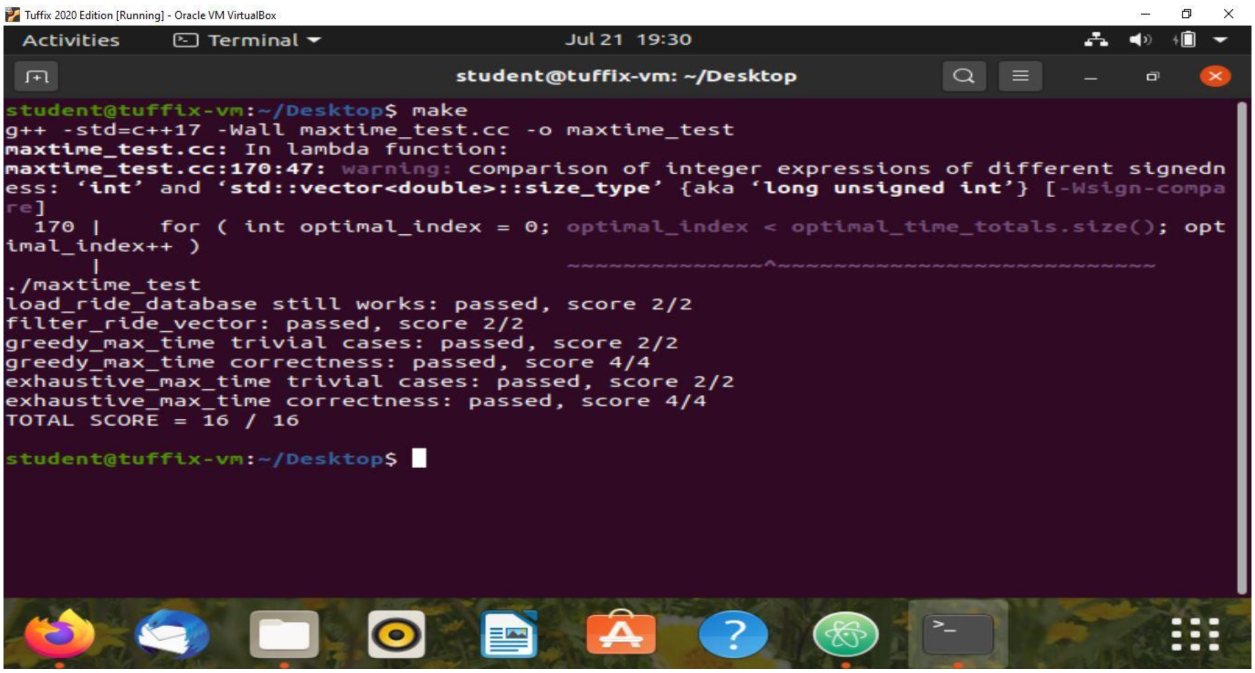
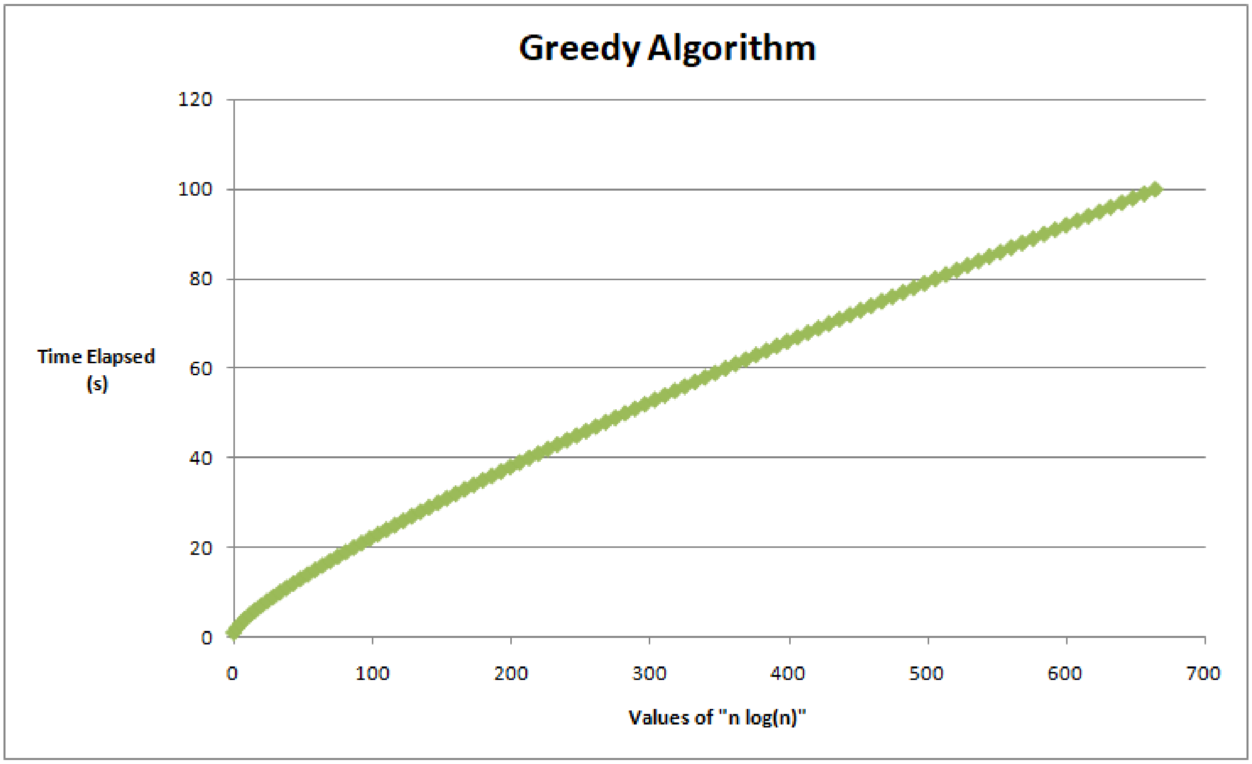
Project 2: Greedy versus Exhaustive

Mike Sim msim@csu.fulleton.edu
Steven Tran transteven@csu.fullerton.edu
CPSC 335-02 Algorithm Engineering
Date 23 July, 2021

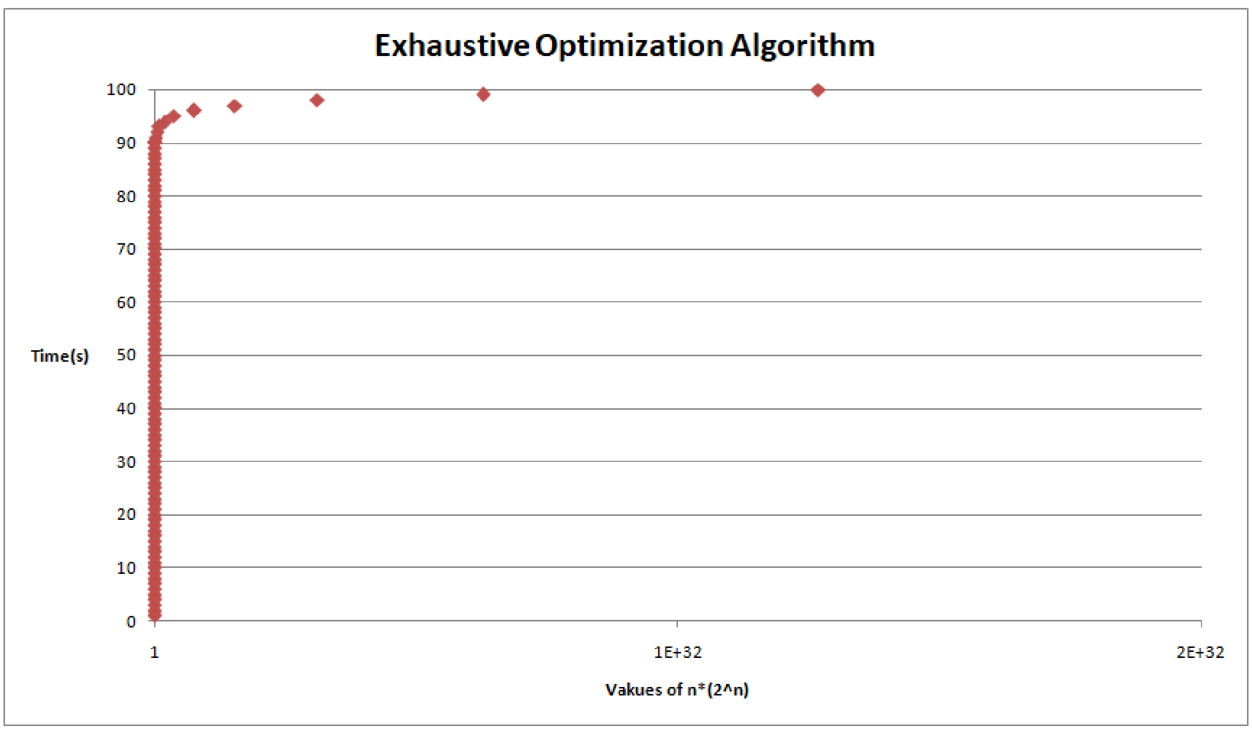
Screen Capture of programming running in Tuffix Environment



Scatter plot of the Greedy Algorithm is shown below. (Note: The base of the log used is 2)



Scatter plot of the Exhaustive Optimization is below.



As seen on the next page, the graphs do show correlation to the mathematical analysis of the pseudo code for each algorithm.

Greedy Algorithm

```
std::unique_ptr<RideVector> greedy_max_time
(
	const RideVector &rides,
	double total_cost
)
{
	RideVector todo_rides = rides;
	std::unique_ptr<RideVector> result(new RideVector());
	double result_cost = 0;
	while (!todo_rides.empty()) {                          --While
		double max_time_per_cost = 0;                      --1
		int max_idx = 0;                                   --1
		int current_idx = 0;                               --1


		for (auto &ride : todo_rides) {
			double time_per_cost = ride->rideTime() / ride->cost();
			if (time_per_cost > max_time_per_cost) {
				max_time_per_cost = time_per_cost;
				max_idx = current_idx;
			}
			current_idx++;
		}

		if ((result_cost + todo_rides[max_idx]->cost()) <= total_cost) {
			result->push_back(todo_rides[max_idx]);
			result_cost += todo_rides[max_idx]->cost();
		}
		// removes the ride from todo_rides at index = max_idx
		todo_rides.erase(todo_rides.begin() + max_idx);
	}
	return result;
}
```

Greedy Algorithm

```
std::unique_ptr<RideVector> greedy_max_time
(
        const RideVector &rides,
        double total_cost
)
{
        RideVector todo_rides = rides;
        std::unique_ptr<RideVector> result(new RideVector());
        double result_cost = 0;
        while (!todo_rides.empty()) {                                   --While
                double max_time_per_cost = 0;                           --1
                int max_idx = 0;                                        --1
                int current_idx = 0;                                    --1

                for (auto &ride : todo_rides) {
                        double time_per_cost = ride->rideTime() / ride->cost();
                        if (time_per_cost > max_time_per_cost) {
                                max_time_per_cost = time_per_cost;
                                max_idx = current_idx;
                        }
                        current_idx++;
                }

                if ((result_cost + todo_rides[max_idx]->cost()) <= total_cost) {
                        result->push_back(todo_rides[max_idx]);
                        result_cost += todo_rides[max_idx]->cost();
                }
                // removes the ride from todo_rides at index = max_idx
                todo_rides.erase(todo_rides.begin() + max_idx);
        }
        return result;
}
```

let todo: n

n

6

$n*6$

3

4

$6 + while(6n+4)$

$= 6 + n\log 6n + n\log 4$

$\in O(n\log 6n + n\log 4 + 6)$ trivial

$= O(n\log 6n)$ dominated

$= O(n\log n)$ drop constant

$\therefore O(n\log n)$

The Greedy Algorithm takes **O(nlogn)** time

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;
}
```

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

-3

-1, -1, -2, -1: 5

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

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

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

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

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

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

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

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

The Exhaustive 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 greedy algorithm (O(nlogn)) 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 greedy 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.