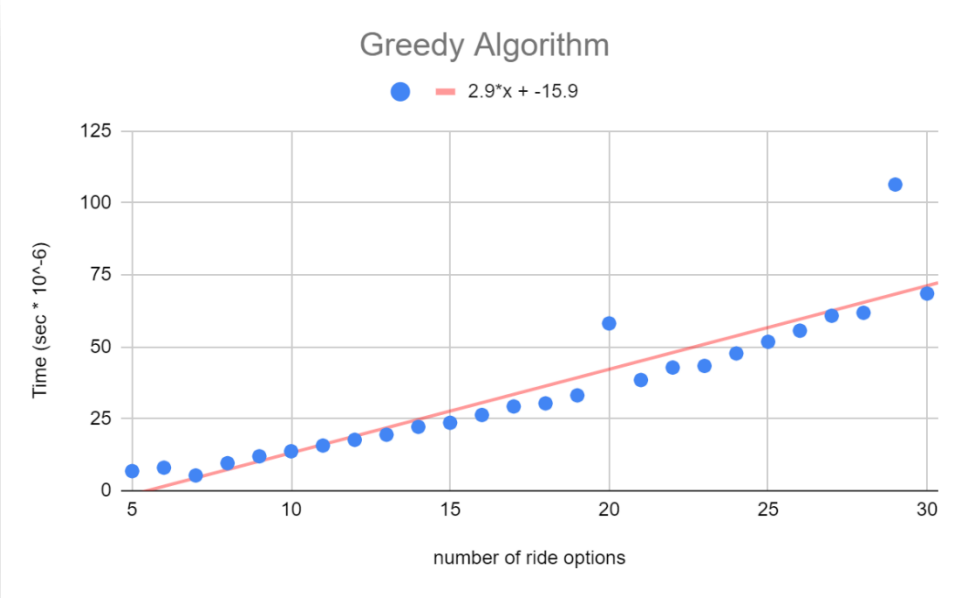
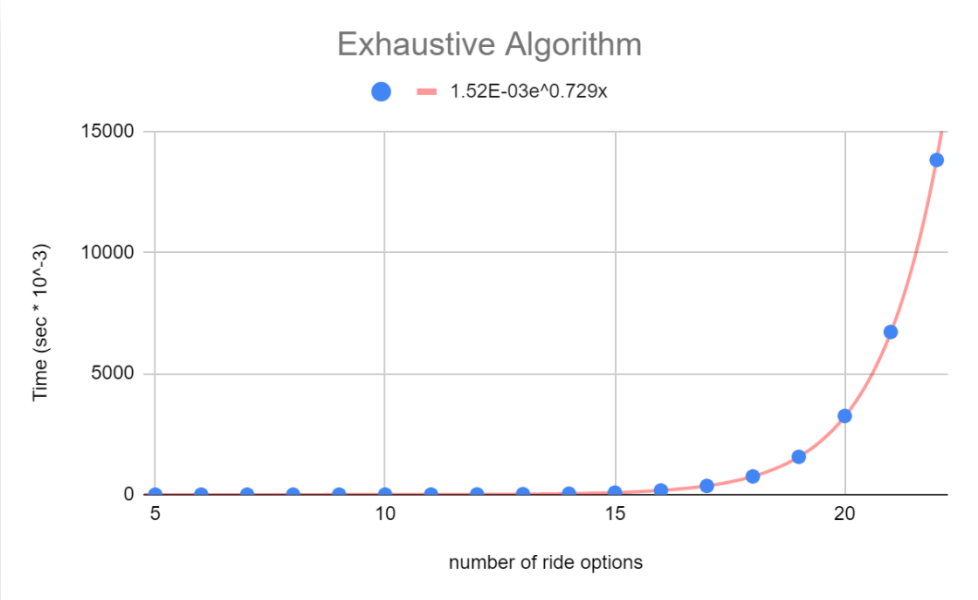
Project 2 - County Fair

Group members:

Preston Fawcett: ptfawcett@csu.fullerton.edu

Gavin Gray: graygavin11@csu.fullerton.edu





Empirical Data

	A	B	C
1		Time (sec*10^-6)	Time (sec*10^-3)
2	n	Greedy Algorithm	Exhaustive Algorithm
3	5	6.819	0.054392
4	6	8.003	0.087032
5	7	5.306	0.224394
6	8	9.573	0.462136
7	9	11.968	0.948889
8	10	13.7	2.02358
9	11	15.692	4.05769
10	12	17.686	8.68481
11	13	19.468	18.7506
12	14	22.195	38.6396
13	15	23.614	82.1454
14	16	26.321	173.146
15	17	29.286	360.556
16	18	30.355	752.914
17	19	33.124	1561.18
18	20	58.138	3248.07
19	21	38.47	6716.04
20	22	42.804	13815.4
21	23	43.369	
22	24	47.71	
23	25	51.758	
24	26	55.614	
25	27	60.797	
26	28	61.833	
27	29	106.393	
28	30	68.513	

Analysis

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 significant difference between the performance of the two algorithms. The greedy approach is much more efficient than the exhaustive algorithm. The greedy Algorithm shows when n= 20 that it grabs the result at 0.000042 seconds compared to the exhaustive algorithm which shows it grabbing the result at 13.815 seconds when n=20. This is a significant difference and did not surprise us at all.

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

The empirical data is consistent with our mathematical analysis. Our math shows the algorithm performing at O(n^2) for the greedy algorithm and O(2^n) for exhaustive search.

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

The evidence is consistent. Exhaustive search algorithms are feasible to implement and produce correct outputs due to it being possible to implement. The algorithm checks every possible outcome until the end which does give correct solutions.

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

The evidence is consistent that algorithms with exponential running times are extremely slow, probably too slow to be of practical use. The data here shows a significant difference between the two algorithms and how fast they were able to grab the result. The Exhaustive algorithm performed much slower than the greedy algorithm which is shown to be more linear to linear exponential whereas the exhaustive algorithm is fully exponential in its complexity and performed seconds slower than the greedy algorithm which performed at .000042 seconds.

Mathematical Analysis

```
RideVector ride_options = rides;                                        sc: 1
    std::unique_ptr<RideVector> result(new RideVector);                 sc: 1
    double result_cost = 0;                                             sc: 1

    while (ride_options.size() > 0) {                                   sc: 8+ sum_{i=n}^{1} 10=
                                                                        10 * n(n+1)/2 + 8 = 5n^2 + 5n + 8
        auto current_ride = ride_options[0];                            sc: 1
        auto iter = ride_options.begin();                               sc: 1
        auto current_iter = iter;                                       sc: 1
        for (auto ride : ride_options) {
            if (ride->rideTime()/ride->cost() >
            current_ride->rideTime()/current_ride->cost()) {            sc:7+max(2,0)= 9
                current_ride = ride;                                    sc: 1
                current_iter = iter;                                    sc: 1
            }
            Iter++;                                                     sc: 1
        }

        ride_options.erase(current_iter);                               sc: 1
        if (result_cost + current_ride->cost() <= total_cost) {         sc: 3+max(2,0)=5
            result->push_back(current_ride);                            sc: 1
            result_cost += current_ride->cost();                        sc: 1
        }
    }

    return result;
}
```

Total sc: $5n^2 + 5n + 11 \in O(n^2)$

```
    for (uint64_t bits = 0; bits <=                                     sc:
    (pow(2, ride_options.size()) - 1); bits++) {                        2^n * (1 + 4n + 3 + 5) = 2^n * 4n + 2^n * 9
        RideVector candidate;                                                                   sc: 1
        for (size_t j = 0; j <= ride_options.size() - 1; j++) {                                 sc: n * 4 = 4n
            if (((bits >> j) & 1) == 1)                                                         sc: 3+max(1,0)=4
                candidate.push_back(ride_options[j]);                                           sc: 1
        }
        double vector_cost, vector_time, best_cost, best_time;                                  sc: 1
        sum_ride_vector(candidate, vector_cost, vector_time);                                   sc: 1
        sum_ride_vector(*best, best_cost, best_time);                                           sc: 1
        if (vector_cost <= total_cost)                                                          sc: 1+max(4,0)=5
            if (!best->size() || vector_time > best_time)                                       sc: 3+max(1,0)=4
                *best = candidate;                                                              sc: 1
    }

    return best;
}
```

Total sc: $2^n * 4n + 2^n * 9 \in O(2^n * n)$