

Chase Huante
Shaochen Ren

Team Cactus Project 2 Report

Finally, produce a brief written project report **in PDF format**. Your report should be submitted as a checked-in file in GitHub. Your report should include the following:

1. Analyze your greedy algorithm code mathematically to determine its big-O efficiency class, probably $O(n^2)$ or $O(n \log n)$.

GREEDY

```
Std::unique_ptr< Ride Vector > greedy_max_time (const Ride Vector& rides, double total_cost)
5 {
    auto todo = rides; // 1
    Std::unique_ptr< Ride Vector > result (new Ride Vector); // 0
    double result_cost = 0; // 1
    while (!todo.empty()) { // ? (log(n)+1)
        auto start = todo[0]; // 1
        auto iter = todo.begin(); // 1
        auto current = iter; // 1
        for (auto c: todo) { // n
            4 3 {
                2 if (c->rideTime() / c->cost() > start->rideTime() / start->cost()) { // 3 IF
                    start = c; // 1
                    current = iter; // 1
                }
                3 iter++; // 1
            }
            todo.erase(current);
            1 {
                if (result_cost + start->cost() <= total_cost) { // 2 IF
                    result->push-back(start); // 1
                    result_cost += start->cost(); // 1
                }
                3
            }
        }
    }
}
```

1 = 2 + max(2, 0) = 2 + 2 = 4
2 = 3 + max(2, 0) = 3 + 2 = 5
3 = (n - 64 + 1) × (5 + 1) = 63n × 6 = 378n
4 = (log n) × (63n × 6) = 63n × 6 log n
5 = 2 + 63n × 6 log(n) = $O(n \log(n))$

2. Analyze your exhaustive optimization algorithm code mathematically to determine its big-O efficiency class, probably $O(2^n \cdot n)$.

```

auto todo = rides; SC 1
int n= rides.size(); SC 1
if(n >= 64){SC 1
    return nullptr;SC 0
else
    assert(n < 64);SC 1
    std::unique_ptr<RideVector> best(new RideVector); SC 0
    for 0 to 2^n do SC 2^n +1
        candidate = empty(); SC 1
        for 0 to n-1 do {SC n
            if(((i >> j) & 1) == 1) SC 3
                candidate.push_back(todo[j]) SC 1
        if (total_cost(candidate) <= total_cost(best)) SC 1
            if(best.empty() || total_time(candidate) > total_time(best)) SC 2
                *best = *candidate; SC 1

```

Step count: first for loop

$$(2^n + 1) * 1 * n * (3 + 1) * (1 + 2 + 1)$$

$$= (2^{n+1}) * 16n$$

Proof:

$$(2^{n+1}) * 16n$$

$$16n * 2^n + 16n \text{ belongs to } O(2^{n+1} * n)$$

Find $c > 0$ and $n_0 > n$ st $16n * 2^n + 16n$

$$\text{Choose } c = 16 + 16 = 32$$

$$16n * 2^n + 16n \leq 32 * 2^n * n$$

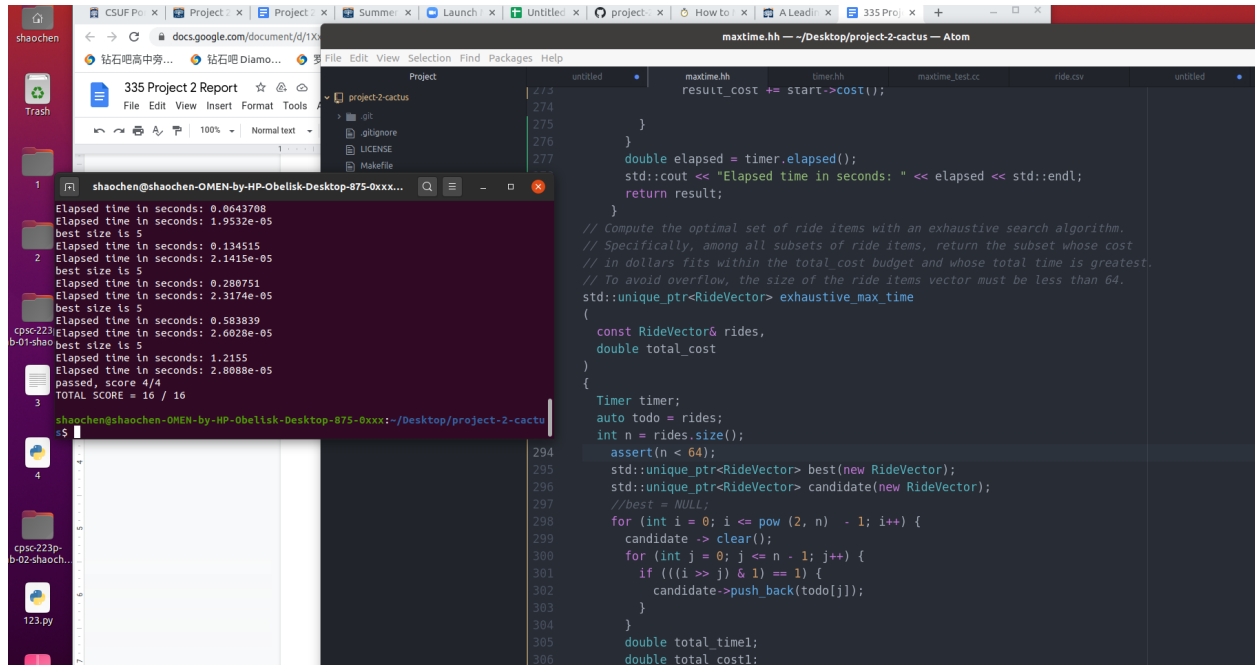
$$16 * 2^n * n - 16n * 2^n + 16 * 2^n * n - 16n \geq 0 \quad \forall n \geq 0 \quad n_0 = 1$$

By definition that $16n * 2^n + 16n$ belongs to $O(2^{n+1} * n)$

1. Your names, CSUF-supplied email address(es), and an indication that the submission is for project 2.

Chase Huante
Shaochen Ren

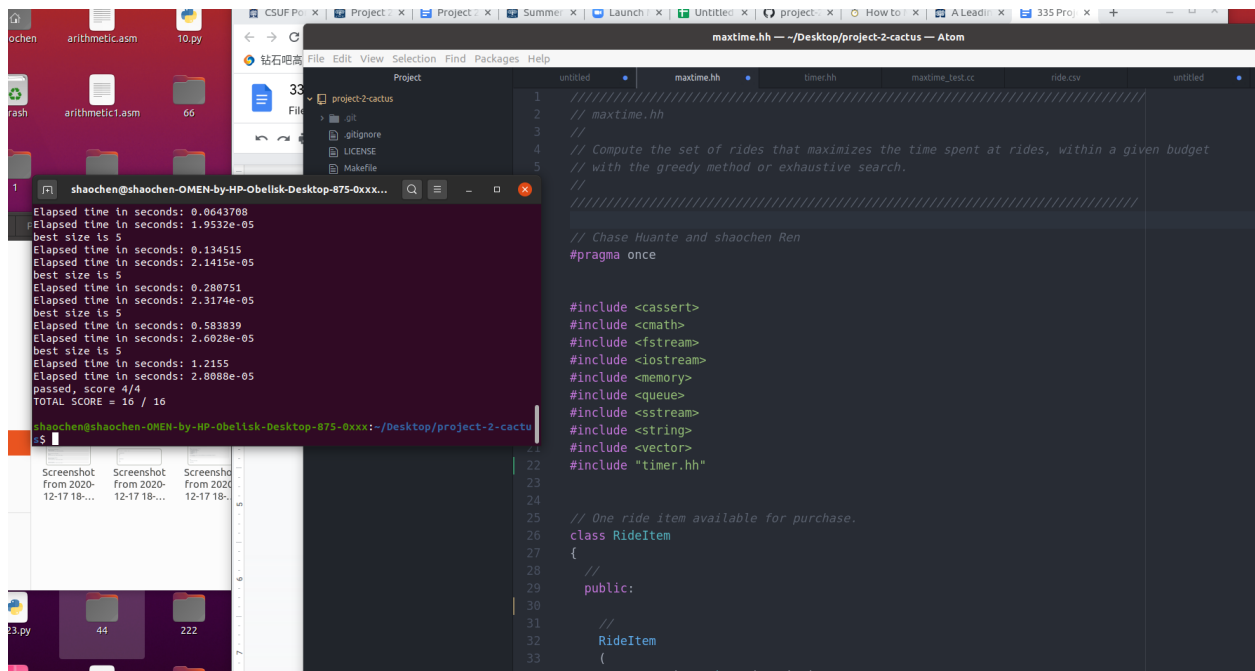
cdhuante@csu.fullerton.edu
renleo@csu.fullerton.edu



The screenshot shows a Linux desktop environment. In the background, there is a Google Docs window titled "335 Project 2 Report". In the foreground, a code editor (Atom) is open, displaying a C++ file named "maxtime.hh". The code defines a class "RideItem" and a function "compute_optimal_set". A terminal window is open in the foreground, showing the output of a program. The output displays elapsed time in seconds and the best size for various test cases. The file explorer shows a directory structure for "project-2-cactus".

```
maxtime.hh -- ~/Desktop/project-2-cactus -- Atom
// Compute the optimal set of ride items with an exhaustive search algorithm.
// Specifically, among all subsets of ride items, return the subset whose cost
// in dollars fits within the total cost budget and whose total time is greatest.
// To avoid overflow, the size of the ride items vector must be less than 64.
std::unique_ptr<RideVector> exhaustive_max_time
(
    const RideVector& rides,
    double total_cost
)
{
    Timer timer;
    auto todo = rides;
    int n = rides.size();
    assert(n < 64);
    std::unique_ptr<RideVector> best(new RideVector);
    std::unique_ptr<RideVector> candidate(new RideVector);
    //best = NULL;
    for (int i = 0; i <= pow(2, n) - 1; i++) {
        candidate->clear();
        for (int j = 0; j <= n - 1; j++) {
            if (((i >> j) & 1) == 1) {
                candidate->push_back(todo[j]);
            }
        }
        double total_time;
        double total_cost;
        compute_optimal_set(candidate, total_time, total_cost);
        if (total_cost < best->total_cost) {
            best = candidate;
        }
    }
    return best;
}
```

```
shaochen@shaochen-OMEN-by-HP-Obelisk-Desktop-875-0xxx:~/Desktop/project-2-cactus$
Elapsed time in seconds: 0.0643708
Elapsed time in seconds: 1.9532e-05
best size is 5
Elapsed time in seconds: 0.134515
Elapsed time in seconds: 2.1415e-05
best size is 5
Elapsed time in seconds: 0.280751
Elapsed time in seconds: 2.3174e-05
best size is 5
Elapsed time in seconds: 0.583839
Elapsed time in seconds: 2.6028e-05
best size is 5
Elapsed time in seconds: 1.2155
Elapsed time in seconds: 2.8088e-05
passed, score 4/4
TOTAL SCORE = 16 / 16
```



The screenshot shows a Linux desktop environment. In the background, there is a Google Docs window titled "335 Project 2 Report". In the foreground, a code editor (Atom) is open, displaying a C++ file named "maxtime.hh". The code defines a class "RideItem" and a function "compute_optimal_set". A terminal window is open in the foreground, showing the output of a program. The output displays elapsed time in seconds and the best size for various test cases. The file explorer shows a directory structure for "project-2-cactus".

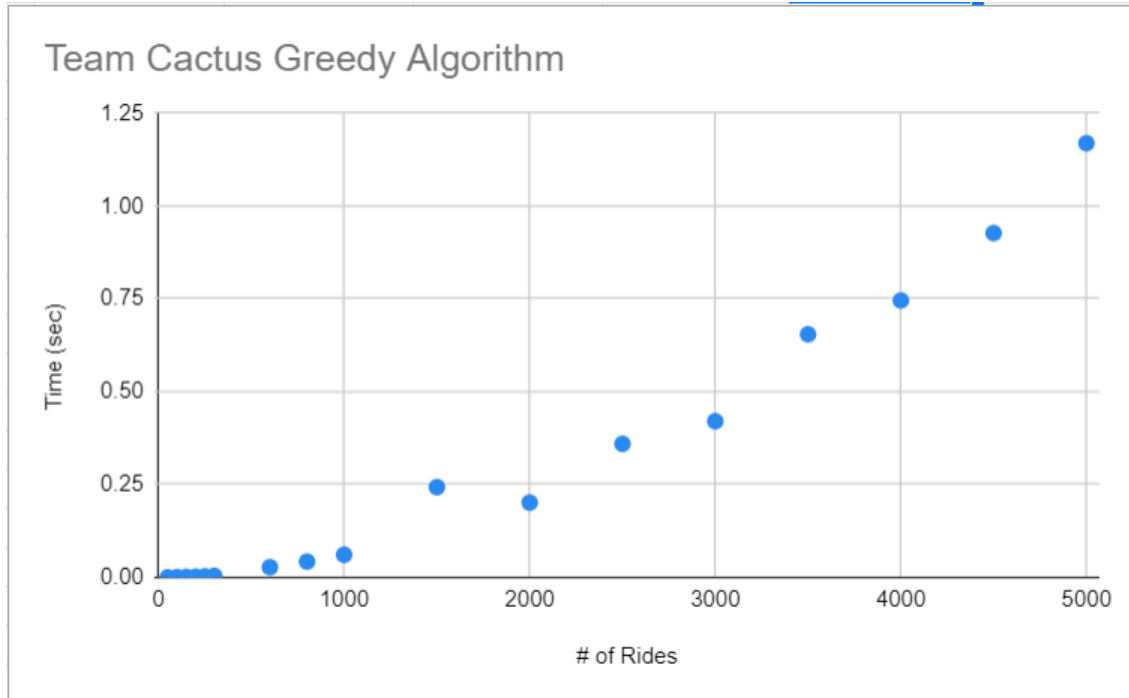
```
maxtime.hh -- ~/Desktop/project-2-cactus -- Atom
// Compute the set of rides that maximizes the time spent at rides, within a given budget
// with the greedy method or exhaustive search.
//
// Chase Huante and shaochen Ren
#pragma once

#include <cassert>
#include <cmath>
#include <fstream>
#include <iostream>
#include <memory>
#include <queue>
#include <sstream>
#include <string>
#include <vector>
#include "timer.hh"

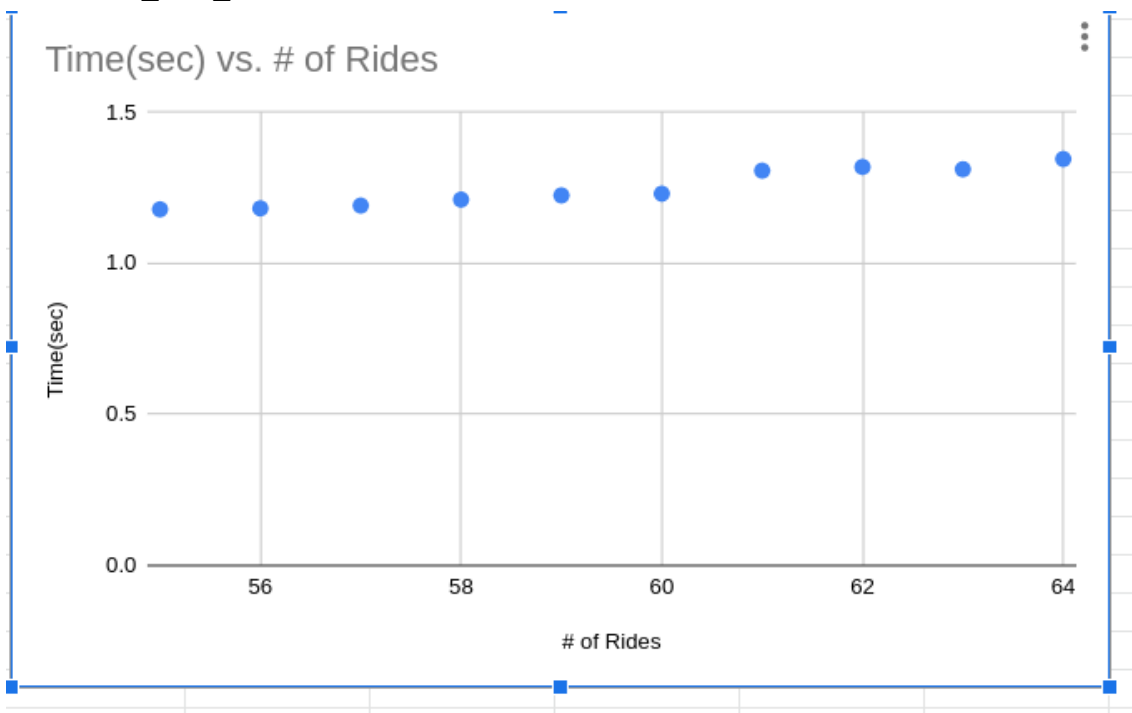
// One ride item available for purchase.
class RideItem
{
public:
    //
    RideItem
    (
        const std::string& description
    )
    {
        //
    }
};
```

```
shaochen@shaochen-OMEN-by-HP-Obelisk-Desktop-875-0xxx:~/Desktop/project-2-cactus$
Elapsed time in seconds: 0.0643708
Elapsed time in seconds: 1.9532e-05
best size is 5
Elapsed time in seconds: 0.134515
Elapsed time in seconds: 2.1415e-05
best size is 5
Elapsed time in seconds: 0.280751
Elapsed time in seconds: 2.3174e-05
best size is 5
Elapsed time in seconds: 0.583839
Elapsed time in seconds: 2.6028e-05
best size is 5
Elapsed time in seconds: 1.2155
Elapsed time in seconds: 2.8088e-05
passed, score 4/4
TOTAL SCORE = 16 / 16
```

2. Two scatter plots meeting the requirements stated above.
Greedy



exhaustive_max_time



3. Answers to the following questions, using complete sentences.

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

When comparing the 2 functions, greedy is considerably faster than exhaustive search. Reason being is because of exhaustive's exponential power to n which is why we had to bring the list down to 64. Anything longer and the search could be too demanding. Although this would be a surprising moment, we've been anticipating these results the whole time so no we aren't surprised by the time discrepancy.

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

When comparing the empirical analysis to the mathematical analysis they both correlate to a similar median with a few outside outliers. An example is how our greedy function is an $O(n \log n)$ search which can relate to a graph when substituting enough n values for $O(n \log(n))$ with a slope of zero.

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

Consistent to some degree but ideally inconsistent. The reason being is because exhaustive search produces a more correct answer than greedy but with real world implications we cannot always use exhaustive search whether it be time or resource constraints.

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

Hypothesis 2 is correct. Exhaustive for example. Once we entered an exponential time complexity when looping all subsets of an n -element sequence, the time considerably jumped per ride. To the knowledge given in the class, $O(c^n)$ is the highest time complexity.