Chase Huante
Shaochen Ren

Project 4 Report

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

```
                                                                                                  S.C.
std::unique_ptr<RideVector> dynamic_max_time(const RideVector &rides, int total_cost){           0
  std::unique_ptr<RideVector> final(new RideVector);                                              1
  int n = rides.size();                                                                           1
  std::vector<std::vector<double>    T (n+1, std::vector<double> (total_cost +1));                1
  for (int x=0; x<=n; x++) {                                                                      8065
    for (int y=0; y<=total_cost; y++{                                                             8065
        T [x][y]=0;                                                                               1
    }
  }
  for (int x=1; x<= n; x++){                                                                      n
    for (int y = 1; y<=total_cost ;y++) {                                                         n
      if (rides[x-1] -> cost() <= y) {                                                            2
        T[x][y]=std::max(T[x-1][y], T[x-1][y- rides[x-1] -> cost()] + rides[x-1]-> time());       7
      }
      else
        T[x][y] = T[x-1][y];                                                                      2
    }
  }
  double result = T[n][total_cost];                                                               2
  int cost = total_cost;                                                                          1
  for (int i = n; i >0 && result >0 && cost >0; i-- ){                                            n
    if (result != T[i-1][cost] {                                                                  2
      answer-> pushback (rides[i-1]);                                                             1
      result = T[i-1][cost - rides[i-1]->cost()];                                                 4
      cost = cost - rides[i-1]-> cost ();                                                         3
    }
  }
  Return answer;
}
```

1 = 1 x 8065 = 8065

2 = 8065 x 8065 = 65,044,225

3 = 2 + max (7,2) = 2 + 7 = 9

4 = n x 9 = 9n

5 = n x 9n = $9n^2$

6 = 2 + max (8,0) = 2+8 = 10

7 = n x 10 = 10n

8 = 5 + 10n + $9n^2$ + 65,044,225 => $O(n^2)$ Time Complexity

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 doSC 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)SC 1
                        if(best.empty() || total_time(candidate) > total_time(best))SC  2
                                *best = *candidate; SC 1
```

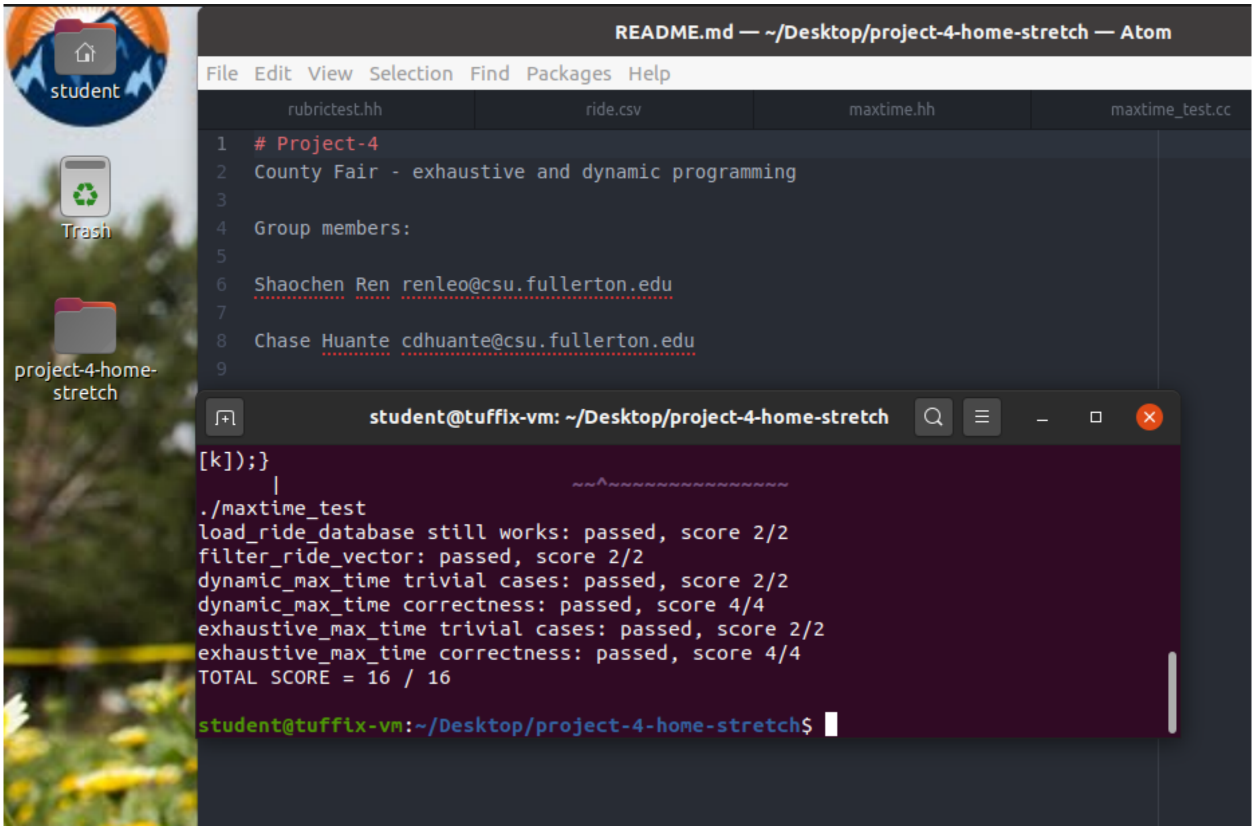
Step count: first for loop
(2^n +1)* 1 * n *(3+1)*(1+2+1)
= (2^n+1)*16n

Proof:
(2^n+1)*16n
16n*2^n+16n belongs to O(2^n*n)
Find c > 0 and n0> n st 16n*2^n+16n
Choose c = 16+16 = 32
16n*2^n+16n <= 32*2^n*n
16*2^n*n-16n*2^n+16*2^n*n-16n >=0 V n>=0 n0=1
Bu definition that 16n*2^n+16n belongs to O(2^n*n)

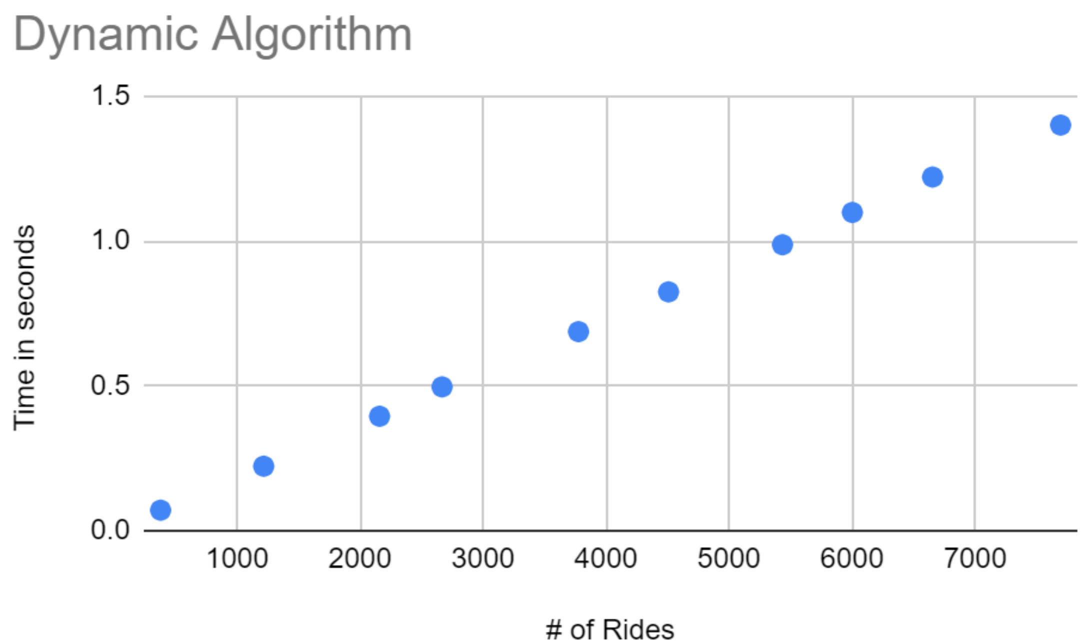
1. Your names, CSUF-supplied email address(es), and an indication that the submission is for project 4.

Shaochen Ren renleo@csu.fullerton.edu

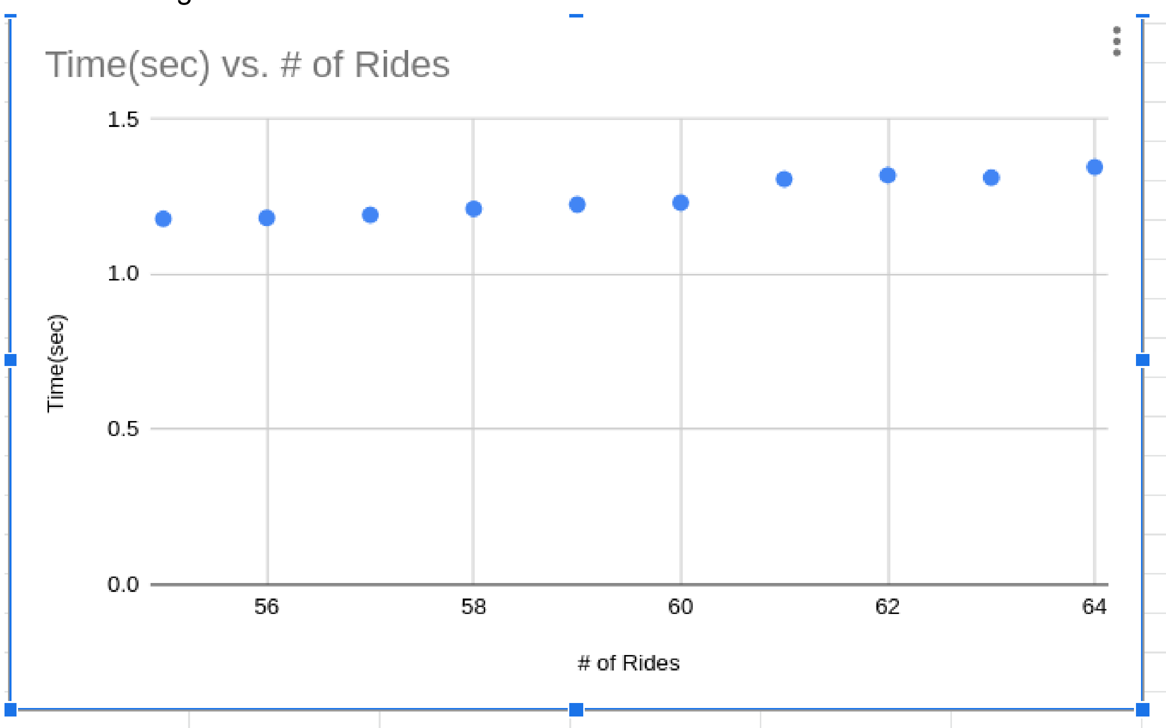
Chase Huante cdhuante@csu.fullerton.edu



2. Two scatter plots meeting the requirements stated above.



Exhaustive Algorithm



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?

To no surprise the dynamic algorithm computes faster than the exhaustive search. The reason being is that O(2^n*n) is exponentially slower than O(n^2). The reason why this isn't a surprise is because this project is teaching how successful polynomial time with dynamic programming can be used.

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(nlogn) search which can relate to a graph when substituting enough n values for o(nlog(n)) with a slope of zero.

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

This is correct in the sense of providing a correct output but ultimately not feasible to implement. What project 4 shows is that the same problem can be solved and provided the same answer without the exponential time complexity

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.