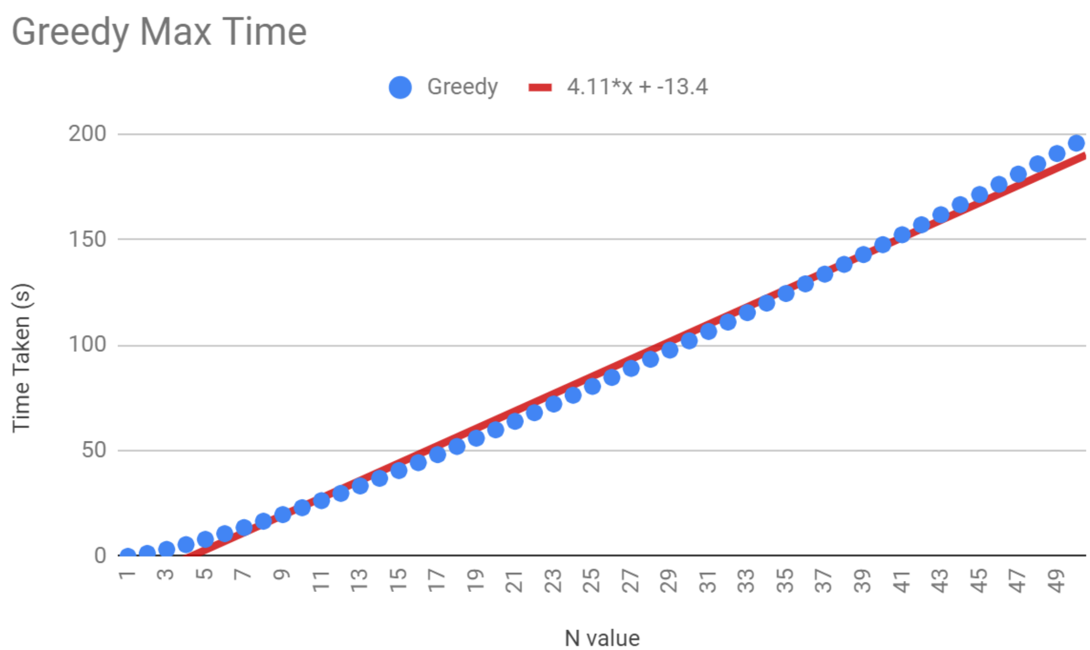
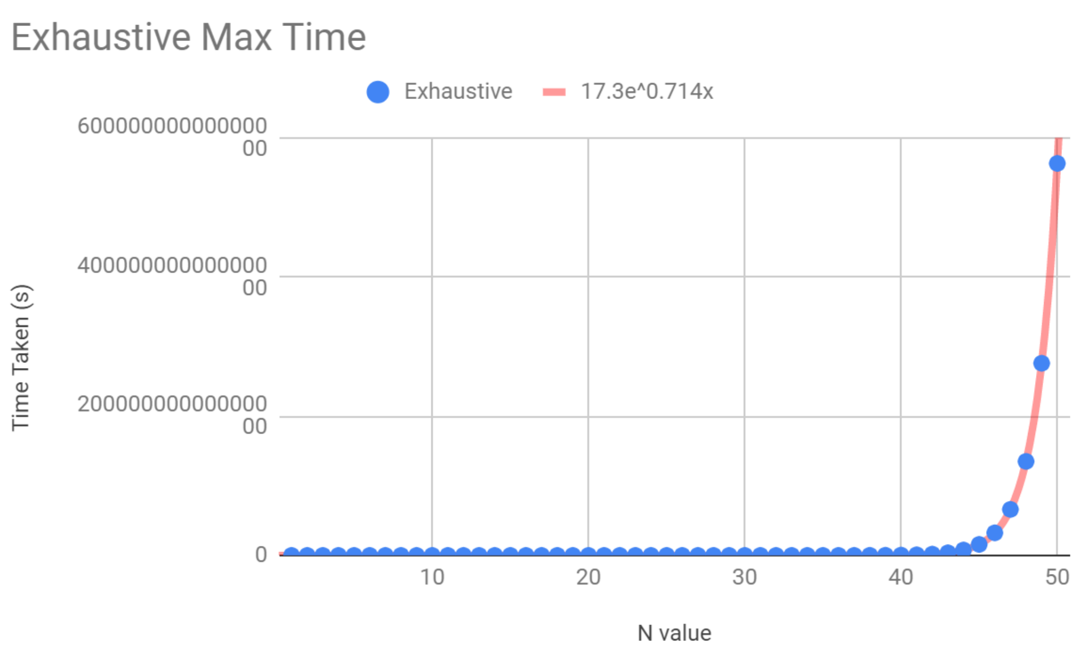
Project 2 Report

Yujin Chung yujin.chung@csu.fullerton.edu
Jason Liu jliu96@csu.fullerton.edu

SCATTERPLOT





EMPIRICAL TIMING DATA GATHERED

N value	Greedy (s)	Exhaustive (s)	Difference in seconds
1	0	2	2
2	1.386294	8	6.613706
3	3.295837	24	20.704163
4	5.545177	64	58.454823
5	8.04719	160	151.95281
6	10.75056	384	373.24944
7	13.62137	896	882.37863
8	16.63553	2048	2031.36447
9	19.77502	4608	4588.22498
10	23.02585	10240	10216.97415
11	26.37685	22528	22501.62315
12	29.81888	49152	49122.18112
13	33.34434	106496	106462.6557
14	36.9468	229376	229339.0532
15	40.62075	491520	491479.3793
16	44.36142	1048576	1048531.639
17	48.16463	2228224	2228175.835
18	52.02669	4718592	4718539.973
19	55.94434	9961472	9961416.056
20	59.91465	20971520	20971460.09
21	63.93497	44040192	44040128.07
22	68.00293	92274688	92274620
23	72.11637	192937984	192937911.9
24	76.27329	402653184	402653107.7
25	80.4719	838860800	838860719.5
26	84.71051	1744830464	1744830379
27	88.9876	3623878656	3623878567
28	93.30173	7516192768	7516192675
29	97.65158	15569256448	15569256350
30	102.0359	32212254720	32212254618
31	106.4536	66571993088	66571992982
32	110.9035	1.37E+11	1.37E+11

33	115.3847	2.83E+11	2.83E+11
34	119.8963	5.84E+11	5.84E+11
35	124.4372	1.20E+12	1.20E+12
36	129.0067	2.47E+12	2.47E+12
37	133.604	5.09E+12	5.09E+12
38	138.2283	1.04E+13	1.04E+13
39	142.8789	2.14E+13	2.14E+13
40	147.5552	4.40E+13	4.40E+13
41	152.2565	9.02E+13	9.02E+13
42	156.9821	1.85E+14	1.85E+14
43	161.7316	3.78E+14	3.78E+14
44	166.5043	7.74E+14	7.74E+14
45	171.2998	1.58E+15	1.58E+15
46	176.1175	3.24E+15	3.24E+15
47	180.9569	6.61E+15	6.61E+15
48	185.8176	1.35E+16	1.35E+16
49	190.6992	2.76E+16	2.76E+16
50	195.6012	5.63E+16	5.63E+16

MATHEMATICAL ANALYSIS

Greedy Algorithm

```
std::unique_ptr<RideVector> todo(new RideVector(rides));     1
std::unique_ptr<RideVector> result(new RideVector);          1
double result_cost = 0;                                      1    3

while(!(*todo).empty())
{
    double maxTPC = 0;                                       1
    int indexMaxTPC = 0;                                     1

    // Find the ride item "a" in todo of maximum time per its cost
    for (int i = 0; i < (*todo).size(); i++)                 todo.size + 1   (todo.size = n)
    {
        if (i == 0)                                          1
        {
            maxTPC = (*todo)[i]->rideTime()/(*todo)[i]->cost();   2
        }
        else
        {
            double tpc = (*todo)[i]->rideTime() / (*todo)[i]->cost();   2
            if (tpc > maxTPC)                                1
            {
                maxTPC = tpc;                                1
                indexMaxTPC = i;                             1    6
            }
        }
    }
    double c = (*todo)[indexMaxTPC]->cost();                 1

    // if the cost of ride isnt over total_cost then add ride into result
    if ((result_cost + c) <= total_cost)                     2
    {
        (*result).push_back((*todo)[indexMaxTPC]);           1
        result_cost += c;                                    2
    }
    (*todo).erase((*todo).begin() + indexMaxTPC);            1    (1 + 6)
}
return result;
```

$(todo.size + 1)(6)$, $todo.size = n$

$3 + while(6n + 6 + 7)$

$3 + while(6n + 13)$

$3 + n \cdot log6n + n \cdot log13$

$O(n \cdot logn)$

Complexity = O(n*logn)

Proof: 3+nlog(6n) + nlog(13) ∈ O(nlogn) using limit theorem

$$\lim_{n \to \infty} \frac{3 + nlog(6n) + nlog(13)}{nlog(n)} = \lim_{n \to \infty} \frac{log(6n) + 1 + log(13)}{log(n) + 1} = \lim_{n \to \infty} \frac{1/n}{1/n} = 1 \geq 0 \; and \; defined$$

Exhaustive Search Algorithm

```
std::unique_ptr<RideVector> best1(new RideVector);   — 1
int n = rides.size();                                 — 1
double candidateTotalCost = 0;                        — 1
double candidateTotalTime = 0;                        — 1
double bestTotalCost = 0;                             — 1
double bestTotalTime = 0;                             — 1

//ride items vector must be less than 64 to avoid overflow
if (rides.size() >= 64)                               — 1
{
      exit(1);    // if ride size is greater than 64, exit program   — 1
}
                                                      8
for (uint64_t bits = 0; bits < pow(2, n); bits++)     — 2^n + 1
{
      std::unique_ptr<RideVector> candidate1(new RideVector);   — 1

      for (int j = 0; j < n; j++)                     — n+1
      {
            if (((bits >> j) & 1) == 1)               — 3
            {
                  (*candidate1).push_back(rides[j]);  — 1
            }
      }                                               4n+4

      // calculate total cost and total time of candidate and best
      sum_ride_vector(*candidate1, candidateTotalCost, candidateTotalTime);   — 3
      sum_ride_vector(*best1, bestTotalCost, bestTotalTime);                — 3

      // move candidate to best if within budget and has greater total time than current best
      if (candidateTotalCost <= total_cost)           — 1
      {
            if ((*best1).empty() || candidateTotalTime > bestTotalTime)   — 3
            {
                  *best1 = *candidate1;               — 1
            }
      }                                               11
}
return best1;
```

$$8+(2^n+1)(1+4n+4+11)$$

$$8+(2^n+1)(4n+16)$$

$$8+2^n\cdot 4n+2^n\cdot 16+4n+16$$

$$2^n\cdot 4n+2^n\cdot 16+4n+24$$

$$O(2^n\cdot n)$$

Complexity = $O(n*2^n)$

Proof 1: $2^n4n+ 2^n16 + 4n + 24 \in O(2^nn)$

Choose c = $|8| + |32| + |4| + |24| = 68$

So $2^n4n+ 2^n16 + 4n + 24 \quad \forall \; 2^nn \geq n_0$

$\Leftrightarrow 68*2^nn - 2^n4n - 2^n16 - 4n - 24 \geq 0$

$\Leftrightarrow 136 - 8 - 32 - 4 -24 \geq 0 \quad \forall \; n \geq 1$

$68 \geq 0 \qquad\qquad n_0 = 1$

QUESTIONS

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?

   Once n gets large enough, there is a noticeable difference in the performance of the two algorithms. Greedy algorithm is faster because its complexity is O(nlogn) while exhaustive is $O(2^n * n)$. It is difficult to give an absolute value to the difference between the two algorithms because as n grows larger, exhaustive search will become slower.

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

   Our empirical analysis is consistent with the mathematical. This is because the shape of our graph from the empirical analysis is the same as our mathematical analysis. For example, our greedy calculation came out to a complexity O(nlogn), which is the same as the shape of our graph. Likewise, our exhaustive calculation came out to exponential O(n^2 *n) which is also reflected in the shape of our exhaustive search empirical graph.

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

   With our program we were able to prove that it is possible to implement an exhaustive search algorithm that also produces outputs that are correct. The outputs for both greedy and exhaustive were the exact same.

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

   After calculating the run time of exhaustive search it is clear that a value of roughly n=5 the amount of time it takes would be extremely impractical to use in real world applications.