Sean Javiya

Shreeji Khalasi

Team Name: exhausted-from-greedy

Project 2: Greedy vs. Exhaustive

Description

In this project we have implemented and compared two algorithms that solve the budgeted time-maximization problem. For one algorithm we use the greedy pattern. This algorithm always chooses the “best” (highest time-per-cost) ride item that fits within the budget and keeps selecting ride items in this manner until the budget is reached. For the second algorithm we use the exhaustive pattern, which is far slower that the greedy algorithm. Specifically, among all subsets of rides, the exhaustive algorithm returns the subset whose cost in dollars fits within the budget and whose time is the largest. Comparing both the algorithms and their time complexity, we elaborate and demonstrate the extent that the greedy algorithm outperforms the exhaustive algorithm.

pseudocode: This pseudocode provides a general understanding of our algorithm implementations.

greedy

```
std::unique_ptr<RideVector> greedy_max_time (const RideVector& rides, double total_cost) {
  unique_ptr<RideVector> heap;
  *heap = rides
  make_heap(heap.begin(), heap.end(), [] lhs.rideTime/lhs.cost < rhs.rideTime/rhs.cost);
  std::unique_ptr<RideVector> result;
  double result_cost = 0.0;
  while (!heap.empty()) {
    RideItem greedy_selection = heap.front();
    pop_heap(heap.begin(), heap.end(), [] lhs.rideTime/lhs.cost < rhs.rideTime/rhs.cost);
    heap.pop_back();
    double greedy_selection_cost = greedy_selection.cost();
    if (result_cost + greedy_selection_cost <= total_cost) {
      result.push_back(greedy_selection);
      result_cost += greedy_selection_cost;
    }
  }
  return result;
}
```

exhaustive

```
std::unique_ptr<RideVector> exhaustive_max_time (const RideVector& rides, double total_cost) {
  int n = rides.size();
  if (n > 64) { n = 64; }
  for (size_t bits = 0; bits <= 2^n – 1; bits++) {
    unique_ptr<RideVector> candidate;
    for (int j = 0; j < n; j++) {
      if ((bits >> j & 1) == 1) {
        candidate.push_back(rides[j]);
      }
    }
    double* total_cost_candidate, total_time_candidate;
    sum_ride_vector(candidate);
    double* total_time_main, total_cost_main;
    sum_ride_vector(best)
    if (total_cost_candidate <= total_cost) {
      if (best.empty() || total_time_candidate > total_time_main) {
        *best = *candidate;
      }
    }
  }
  return best;
}
```

}

<u>Math Analysis</u>

We will start by performing a step count on our algorithms. Then we will mathematically prove the big-O notation of each algorithm.

There seems to be a split on how the step count should be performed on our greedy algorithm implementation. The complexity of the make_heap is worst-case 3n and the complexity of pop_heap is worst-case 2 log n. However, we provide a lambda function to those standard library functions when we call them. The look-up operations, division operation, and comparison operation potentially add 6 to the step count, which occurs once for each element in the range. My own research shows that step count for lambda functions may or may not include this trailing 6 steps. I was unable to find a consensus in our textbook or on any of the standard reference sites. For this reason, in the greedy analysis, I have performed the step count for both methods (including and excluding the trailing 6 steps in the lambda function). I have also done the mathematical proof for both methods, and prove that both cases (of mathematical analysis including and excluding the trailing 6 steps) still both belong to the same efficiency class regardless.

<u>greedy</u>

```
std::unique_ptr<RideVector> greedy_max_time ( const RideVector& rides, double total_cost) {
  unique_ptr<RideVector> heap;
  *heap = rides                                                                   Step Count: 1
  make_heap(heap.begin(), heap.end(), [] lhs.rideTime/lhs.cost < rhs.rideTime/rhs.cost);   18n or 3n
  std::unique_ptr<RideVector> result;
  double result_cost = 0.0;                                                                1
  while (!heap.empty()) {                                                                  n
    RideItem greedy_selection = heap.front();                                              2
    pop_heap(heap.begin(), heap.end(), [] lhs.rideTime/lhs.cost < rhs.rideTime/rhs.cost);
                                                                          12 log n or 2 log n
    heap.pop_back();                                                                       1
    double greedy_selection_cost = greedy_selection.cost();                                2
    if (result_cost + greedy_selection_cost <= total_cost) {                               2
      result.push_back(greedy_selection);                                     1 + 1 amortized
      result_cost += greedy_selection_cost;                                                2
    }
  }
  return result;
```

}

Total step count:

T(n) = 2n log n + 14n + 2 || 12n log n + 29n + 2

Big O notation: O(n * log n)

I will prove that the algorithm f(n) belongs to the order of O(g(n)). I will use L'hospital's rule.

$$\lim_{n \to \infty} \frac{F(n)}{G(n)} != \infty$$

If the limit as n approaches infinity, of f(n) / g(n) is defined and non-negative then f(n) belongs to the order of O(g(n)).

<u>Case 1)</u>
Let f(n) = $2n\log n + 14n + 2$
Let g(n) = $n\log n$

Using L'Hospital's Rule

$$\lim_{n \to \infty} \frac{F(n)}{G(n)} = \frac{(2n\log n + 14n + 2)}{(n\log n)}$$

$$\lim_{n \to \infty} \frac{F'(n)}{G'(n)} = \frac{(-14n - 2 - 2\ln(n))}{(\ln(2)n^2\log(n)^2)}$$

<u>Case 2)</u>
Let f(n) = $12n\log n + 29n + 2$
Let g(n) = $n\log n$

Using L'Hospital's Rule

$$\lim_{n \to \infty} \frac{F(n)}{G(n)} = \frac{(12n\log n + 29n + 2)}{(n\log n)}$$

$$\lim_{n \to \infty} \frac{F'(n)}{G'(n)} = \frac{(-29n - 2 - 2\ln(n))}{(\ln(2)n^2\log(n)^2)}$$

limit = 0 as n approaches infinity

The limit is defined and non-negative, therefore this algorithm belongs to the order of O(n*2^n).

<u>exhaustive</u>

```
std::unique_ptr<RideVector> exhaustive_max_time (const RideVector& rides, double total_cost) {
  int n = rides.size();                                          Step Count: 2
  if (n > 64) { n = 64; }                                                    2
  for (size_t bits = 0; bits <= 2^n – 1; bits++) {                           2^n
    unique_ptr<RideVector> candidate;
    for (int j = 0; j < n; j++) {                                      n
      if ((bits >> j & 1) == 1) {                                     3
        candidate.push_back(rides[j]);                        1 + 1 amortized
      }
    }
    double* total_cost_candidate, total_time_candidate;
    sum_ride_vector(candidate);                                        n
    double* total_time_main, total_cost_main;
    sum_ride_vector(best)                                               n
    if (total_cost_candidate <= total_cost) {                          1
      if (best.empty() || total_time_candidate > total_time_main) {   3
        *best = *candidate;                                            1
      }
    }
  }
  return best;
}
```

Total step count:

T(n) = (2^n)7n + (2^n)5 + 4

T(n) = $\left(2^n\right)7n+\left(2^n\right)5+4$

Big O notation: O(n * 2^n)

I will prove that the algorithm f(n) belongs to the order of O(g(n)). I will use L'hospital's rule.

$$\lim_{n\to\infty}\frac{F(n)}{G(n)}!=\infty$$

If the limit as n approaches infinity, of f(n) / g(n) is defined and non-negative then f(n) belongs to the order of O(g(n)).

Let f(n) = $\left(2^n\right)7n+\left(2^n\right)5+4$
Let g(n) = $\left(2^n\right)n$

Using L'Hospital's Rule

$$\lim_{n\to\infty}\frac{F(n)}{G(n)}=\frac{\left(\left(2^n\right)7n+\left(2^n\right)5+4\right)}{\left(\left(2^n\right)n\right)}$$

$$\lim_{n \to \infty} \frac{F'(n)}{G'(n)} = \frac{\left(-\ln(2) * 2^{(n+2)} n - 5 * 4^{n} - 2^{(n+2)}\right)}{\left(4^{n} * n^{2}\right)}$$

limit = 0 as n approaches infinity

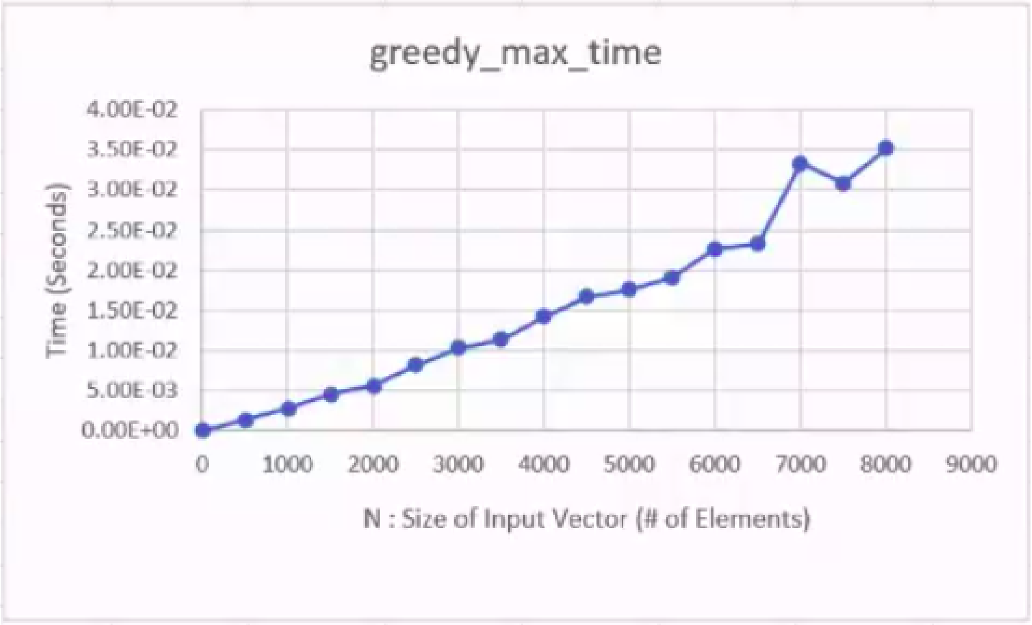
The limit is defined and non-negative, therefore this algorithm belongs to the order of O(n*2^n).

<u>Empirical Data</u> (Scatter Plots)

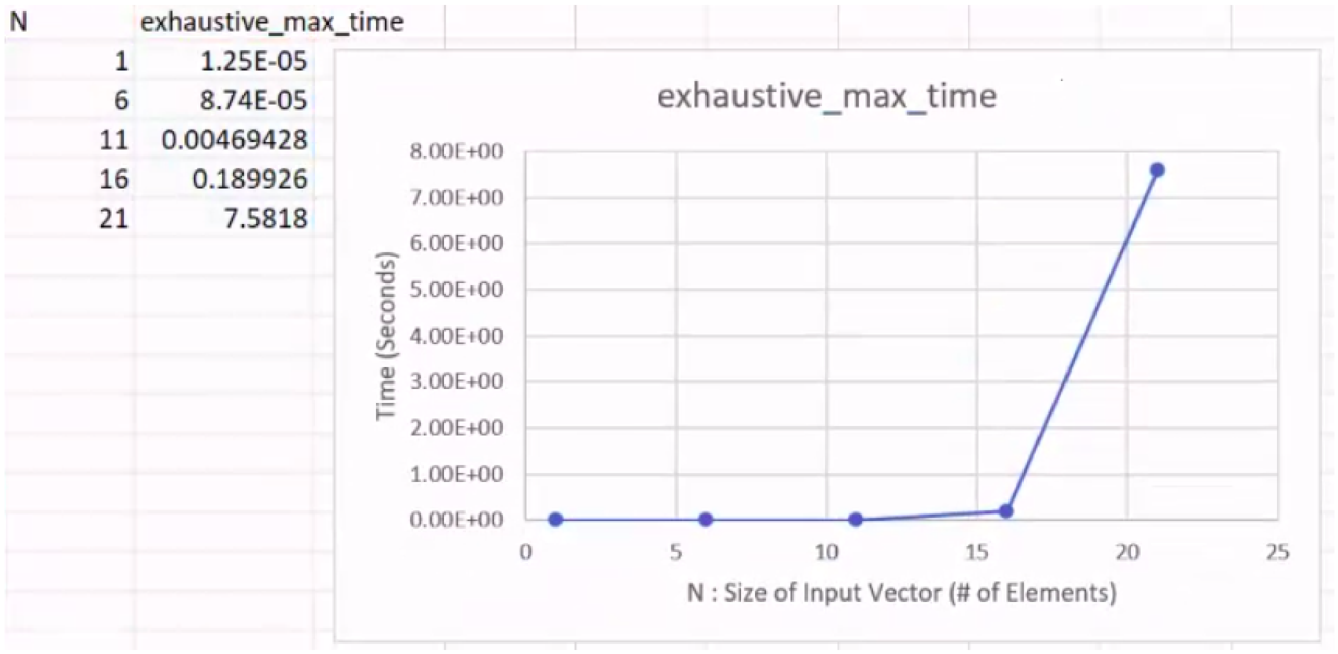
The tables contain the N values (size of input vector) in the left column and the time in seconds (obtained from running each algorithm with an input size of “n”) in the right column.

greedy

N	greedy_max_time
1	1.79E-06
501	0.001329
1001	0.002782
1501	0.004496
2001	0.005573
2501	0.008179
3001	0.01035
3501	0.011365
4001	0.014249
4501	0.016749
5001	0.017584
5501	0.019136
6001	0.02264
6501	0.023286
7001	0.033366
7501	0.030822
8001	0.035201



exhaustive



N	exhaustive_max_time
1	1.25E-05
6	8.74E-05
11	0.00469428
16	0.189926
21	7.5818

Questions:

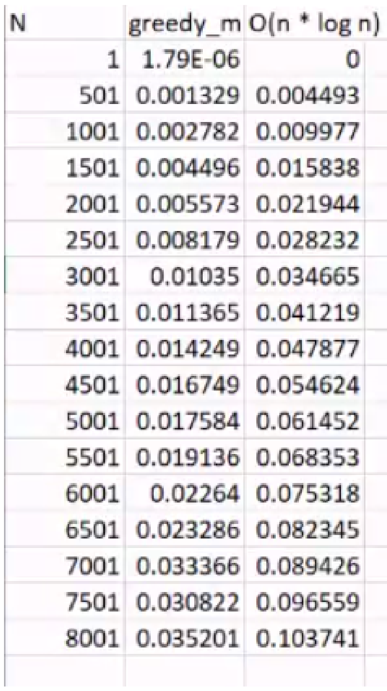
a) There is a noticeable difference in the performance of the two algorithms. The greedy selection algorithm is much faster in terms of run-time. This does not surprise us. Our greedy algorithm is O(n log n) while our exhaustive algorithm is O(n*2^n). The exhaustive algorithm's complexity is much larger than the greedy algorithm's complexity. This becomes most obvious when we compare the graphs of both functions side-by-side.

b)

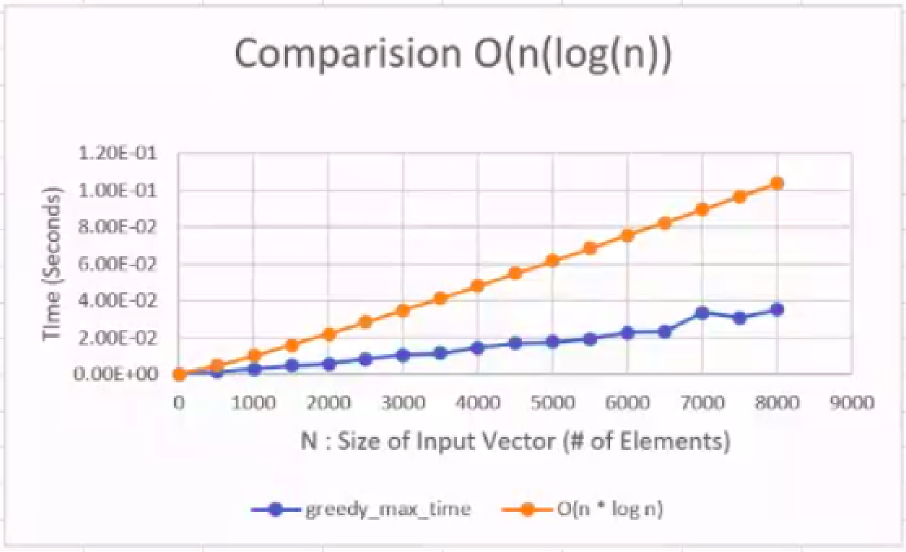
Analysis Conclusion

greedy

We took (n(log(n)) for our various input sizes, and scaled down the result (by 1 million) to represent microseconds. So for n = 2, we took 2 * 0.000001. Then we plotted that line against our empirically-observed data.



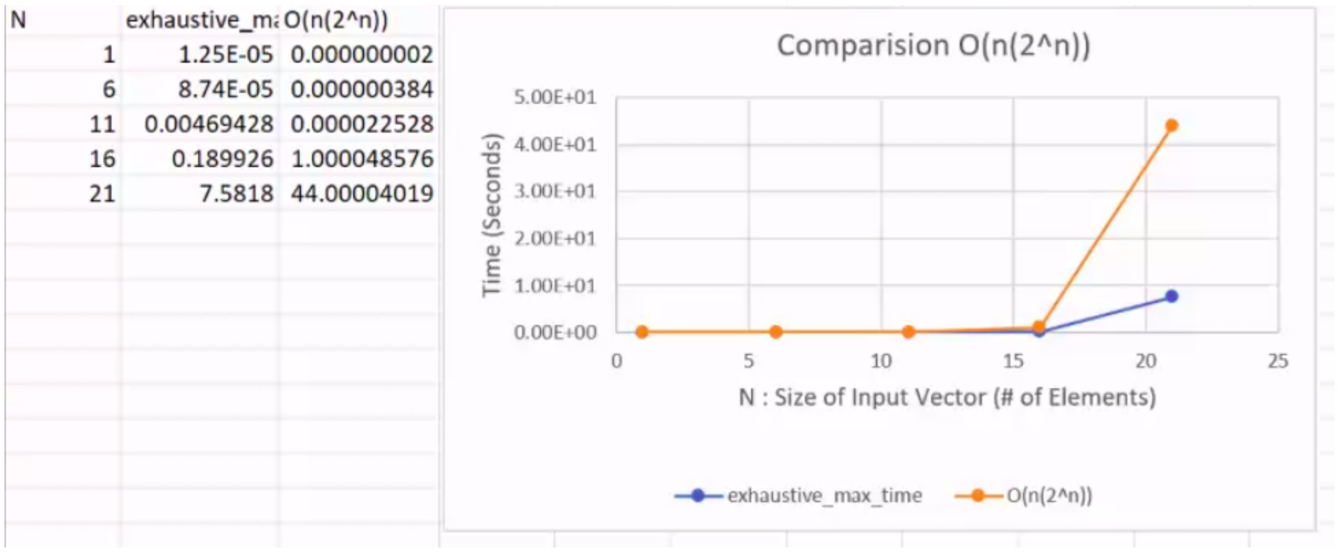
N	greedy_m	O(n * log n)
1	1.79E-06	0
501	0.001329	0.004493
1001	0.002782	0.009977
1501	0.004496	0.015838
2001	0.005573	0.021944
2501	0.008179	0.028232
3001	0.01035	0.034665
3501	0.011365	0.041219
4001	0.014249	0.047877
4501	0.016749	0.054624
5001	0.017584	0.061452
5501	0.019136	0.068353
6001	0.02264	0.075318
6501	0.023286	0.082345
7001	0.033366	0.089426
7501	0.030822	0.096559
8001	0.035201	0.103741



From this graph, we can see that our empirically-observed data is consistent with our mathematically-derived big-O efficiency class.

exhaustive

We took (n(2^n)) for our various input sizes, and scaled down the result (by 1 billion) to represent nanoseconds. So for n = 2, we took 2 * 0.000000001. Then we plotted that line against our empirically-observed data.



N	exhaustive_m	O(n(2^n))
1	1.25E-05	0.000000002
6	8.74E-05	0.000000384
11	0.00469428	0.000022528
16	0.189926	1.000048576
21	7.5818	44.00004019

From this graph, we can see that our empirically-observed data is consistent with our mathematically-derived big-O efficiency class.

c) We will restate the hypothesis.

Exhaustive search algorithms are feasible to implement, and produce correct outputs.

The evidence is consistent with the hypothesis. This exhaustive search algorithm was feasible to implement. In fact, the algorithm was quite straightforward to implement. I believe a similar algorithm could be easily implemented for any similar problems of this nature. The outputs were correct each time.

d) We will restate the second hypothesis.

Algorithms with exponential running times are extremely slow, probably too slow to be of practical use.

The evidence is consistent with the second hypothesis. The exhaustive algorithm, with an exponential running time, was extremely slow and was in fact too slow of any practical use. When applied to a sample size of only 21 elements, our run time was over 7 seconds. At 26 elements, our operating systems would kill the process for taking too long. When you look at the sharp increase in running time when we change element sizes from 16 to 21, the impracticality becomes obvious. At 16 elements, the running time was still less than 1/5th of a second. When we increase the elements to 21 (only 5 more elements), the running time shoots up to 7 seconds. From this trend, we can see why 26 elements was taking too long to execute. Our greedy algorithm, by comparison, was able to execute on an sample with over 8,000 elements in about 1/33rd of a second. In conclusion, algorithms with exponential running times are extremely slow, and this particular algorithm is too slow to be of practical use.