Sean Javiya

Shreeji Khalasi

Team Name: sean-shreeji

Project 4: sean-shreeji

Description

In this project we have implemented and compared two algorithms that solve the budgeted time-maximization problem. For one algorithm we use the dynamic pattern. This algorithm creates a cache of the “best” (highest time-per-cost) ride item that fits within the budget and keeps selecting ride items from that cache. For the second algorithm we use the exhaustive pattern, which is far slower that the dynamic 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 dynamic algorithm outperforms the exhaustive algorithm.

exhaustive

```
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
      }
    }
    int* total_cost_candidate, total_time_candidate;
    sum_ride_vector(candidate);                                            n
    int* 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) = $(2^n)7n+(2^n)5+4$

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

Let g(n) = $(2^n)n$

Using L'Hospital's Rule

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

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

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).

dynamic

```
std::unique_ptr<RideVector> dynamic_max_time (const RideVector& rides, int total_cost) {
  std::vector<std::vector<double>> cache;
  std::unique_ptr<RideVector> source(new RideVector(rides));
  std::unique_ptr<RideVector> result(new RideVector);
  for (int i=0; i<=rides.size(); i++) {                                   Step Count: n + 1 * (
    cache.push_back(std::vector<double>());                                          1 amortized
    for (int j=0; j<=total_cost+1; j++)                                                m + 2 * (1)
      cache.at(i).push_back(0.0);                                                    1 amortized
  }
  double results = 0.0;                                                              1
  for (int i = 1; i<= rides.size();i++) {                                          n * (m * nm+8)
    for (int j =1; j<= total_cost; j++) {                                           m *(nm+8)
      std::shared_ptr<RideItem> item = rides[i-1];                                  2
      if ((j-item→cost())>=0)                                                  2 + max(2, nm+3)
        results = std::max((item->time()+ cache[i-1][j-item->cost()]), cache[i-1][j]);      n*m + 3
      else
        results = cache[i-1][j];                                                    3
      cache[i][j]=results;                                                          1
    }
  }
  double tempcost = total_cost;                                                     1
  for (int i = rides.size(); i >0; i--) {                                           n * (9)
    if (cache[i][tempcost]== cache[i-1][tempcost]){                  3 + max (0, 1 amortized + 5)
      continue;
    }
    else {
    result-> push_back(source→at(i-1));                                  1 amortized + 2
    tempcost-=(rides.at(i-1))→cost();                                               3
    }
  }
return result;
}
```

Total step count:

T(n) = m^2n^2+9mn+m+12n+5

T(n) = $m^2n^2+9mn+m+12n+5$

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

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) = $m^2n^2+9mn+m+12n+5$

Let g(n) = m^2n^2

Using L'Hospital's Rule

$$\lim_{n\to\infty}\frac{F(n)}{G(n)}=\frac{(m^2n^2+9mn+m+12n+5)}{(m^2n^2)}$$

$$\lim_{n\to\infty}\frac{F'(n)}{G'(n)}=\frac{(2m*2n+10)}{(2m*2n)}$$

limit = 1 as n approaches infinity

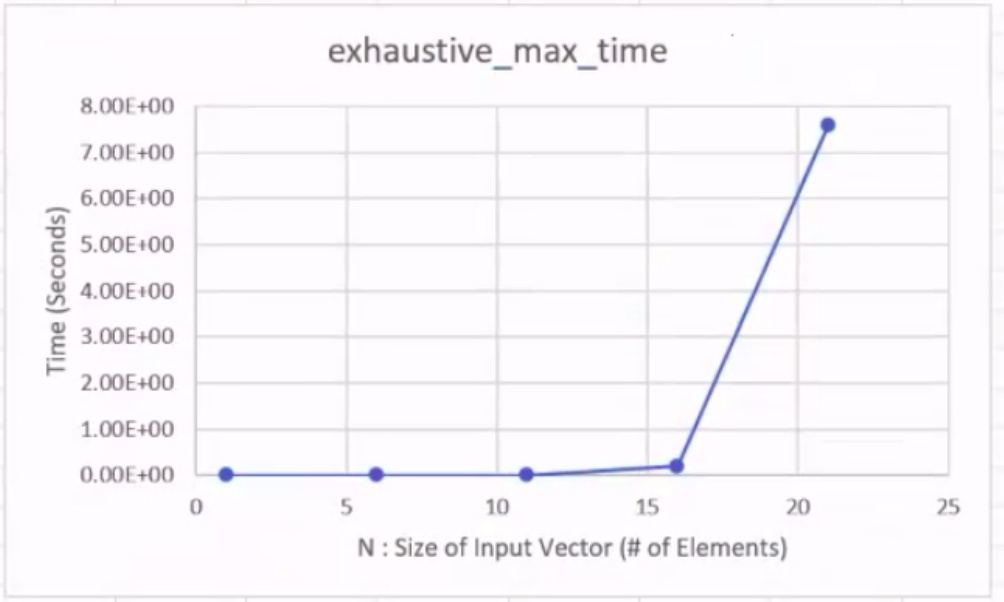
The limit is defined and non-negative, therefore this algorithm belongs to the order of O(n^2 m^2).

Empirical Data (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.

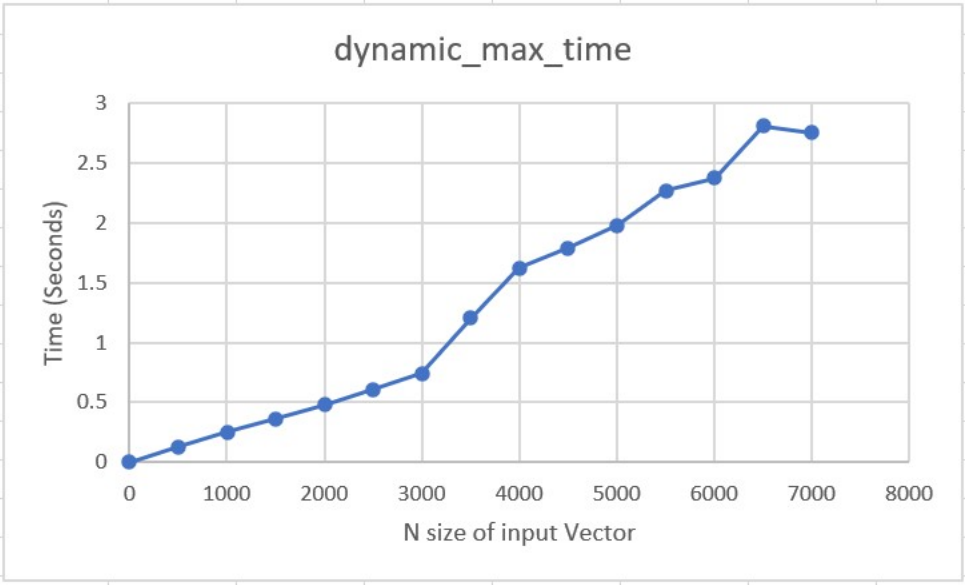
exhaustive

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



dynamic

N	dynamic_max_time
1	0.000323
501	0.128361
1001	0.251114
1501	0.360281
2001	0.477116
2501	0.60919
3001	0.740434
3501	1.20519
4001	1.61931
4501	1.78977
5001	1.97958
5501	2.271
6001	2.37207
6501	2.81157
7001	2.75202



Questions:

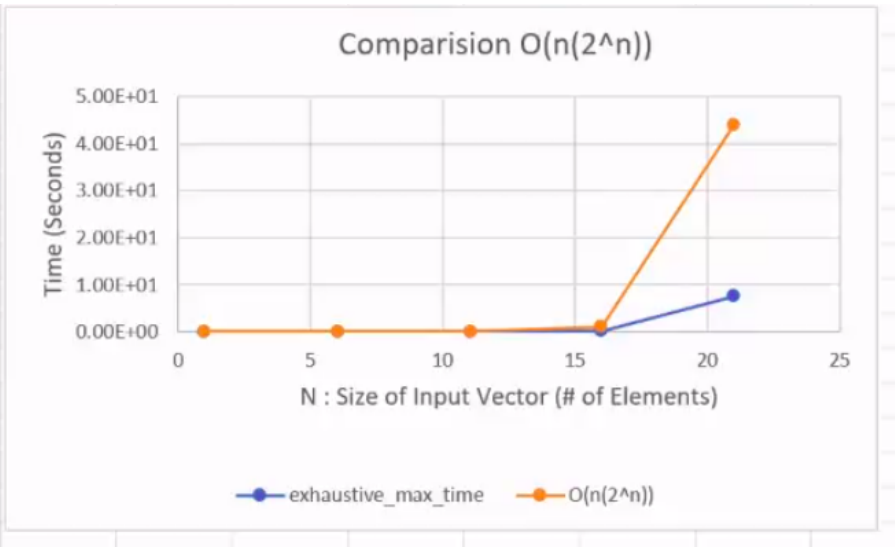
a) There is a noticeable difference in the performance of the two algorithms. The dynamic selection algorithm is much faster in terms of run-time. This does not surprise us. Our dynamic algorithm is O(n^2 m^2) while our exhaustive algorithm is O(n*2^n). The exhaustive algorithm's complexity is much larger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

b)

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

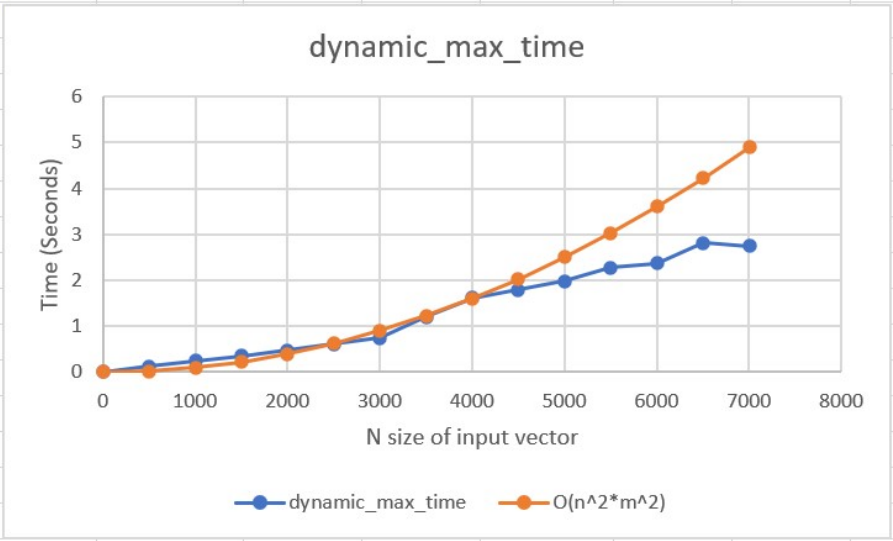


From this graph, we can see that our empirically-observed data is consistent with our mathematically-derived big-O efficiency class.

dynamic

We took m as 1000 for our various input sizes, and derived O(n^2 m^2), and scaled down the result (by 10 trillion). So for n = 2, we took 2^2 * 1000^2 * 0.00000000000001. Then we plotted that line against our empirically-observed data.

N	dynamic_	O(n^2*m^2)
1	0.000323	1E-07
501	0.128361	0.0251
1001	0.251114	0.1002
1501	0.360281	0.2253
2001	0.477116	0.4004
2501	0.60919	0.6255
3001	0.740434	0.9006
3501	1.20519	1.2257
4001	1.61931	1.6008
4501	1.78977	2.0259
5001	1.97958	2.501
5501	2.271	3.0261
6001	2.37207	3.6012
6501	2.81157	4.2263
7001	2.75202	4.9014



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 dynamic algorithm, by comparison, was able to execute on an sample with over 7,000 elements in about 3 seconds. In conclusion, algorithms with exponential running times are extremely slow, and this particular algorithm is too slow to be of practical use. The dynamic algorithm is quadratic, not exponential. This algorithm can be of practical use, by comparison.