CPSC 335
Project 2 – Greedy and Exhaustive Search
Louis Zuckerman
CCWID: 888206315
Email: louiszman@csu.fullerton.edu

1. Greedy Algorithm Code

```
std::unique_ptr<ArmorVector> greedy_max_defense(
    const ArmorVector &armors,
    double total_cost)
{
    // TODO: implement this function, then delete the return statement below
    std::unique_ptr<ArmorVector> result(new ArmorVector);

    double result_cost = 0;     // sc 1
    auto bestArmor = std::shared_ptr<ArmorItem>(new ArmorItem("test helmet", 1, 0
));   // sc 4
    double size = armors.size();      // sc 1

    std::unique_ptr<ArmorVector> todo(new ArmorVector(armors));      // sc 1

    while (!todo->empty())      // while loop condition loop = n+1
    { while_loop_block = 2+1+ (logn-1)*(11)+3+2+6 = 14+11logn-11 = 11logn+3
        double candidate, best = 0;   // sc 2
        int index_best = 0;     // sc 1

        for (int j = 0; j < todo->size(); j++)    // for loop condition logn-1
        { // for block sc = 6+5=11
            candidate = todo->at(j)->defense() / todo->at(j)->cost(); // sc 6

            if (candidate > best)     // if condition sc 1
            { // if block sc = 4
                best = candidate;     // sc 1
                bestArmor = todo->at(j);    // sc 2
                index_best = j;       // sc 1
            }
        }

        todo->erase(todo->begin() + index_best); // sc 3

        double temp_cost = bestArmor->cost();     // sc 2
        if (temp_cost + result_cost <= total_cost)      // if condition sc 2
        { // if block sc = 3
            result->push_back(bestArmor);   // sc 1
            result_cost += bestArmor->cost();     // sc 2
        }
    }
```

CPSC 335
Project 2 – Greedy and Exhaustive Search
Louis Zuckerman
CCWID: 888206315
Email: louiszman@csu.fullerton.edu

```
    return result; // sc 1
}
```

Greedy Algorithm Step Count = 1+4+1+1+ while_loop_block*while loop condition +1

= 8+(11logn+3)* (n+1) = $11logn * n \quad + 11logn + 3n + 3 + 8 = 11logn * n + 11logn + 3n + 11$

➔ $O(nlogn)$

2. Exhaustive Max Defense Algorithm

```
std::unique_ptr<ArmorVector> exhaustive_max_defense(
    const ArmorVector &armors,
    double total_cost)
{
    const int n = armors.size();      // sc 1
    assert(n < 64);      // sc 1

    // TODO: implement this function, then delete the return statement below
    std::unique_ptr<ArmorVector> bestArmor_vec(new ArmorVector);     // sc 1
    double candidate_cost, best_cost, candidate_defense, best_defense;     //sc 4

    for (uint64_t i = 0; i <= (pow(2, n) - 1); i++)
      //for loop condition sc 2^n
    {  for block sc = 1+ (n-1)*(5)+4n-2+4n-2+2n+2=15n-6
        std::unique_ptr<ArmorVector> candidate_vec(new ArmorVector); //sc 1
        for (uint64_t j = 0; j <= n - 1; j++)     //for loop condition n-1
        { for block sc = 5

            if (((i >> j) & 1) == 1) // if condition sc = 3
            { if block sc = 2
                candidate_vec->push_back(armors.at(j)); //sc 2
            }
        }
        sum_armor_vector(*candidate_vec, candidate_cost, candidate_defense);
            // sc 4n-4+2 = 4n-2
        sum_armor_vector(*bestArmor_vec, best_cost, best_defense);
            // sc 4n-2

        if (candidate_cost <= total_cost) // if condition sc = 1
        {    // if block sc = 3+2n-1=2n+2
```

CPSC 335
Project 2 – Greedy and Exhaustive Search
Louis Zuckerman
CCWID: 888206315
Email: louiszman@csu.fullerton.edu

```
            if (bestArmor_vec->empty() || candidate_defense > best_defense)//sc 3
            { if block sc = 1+(n-1)*2=2n-1
                bestArmor_vec->clear();     //sc 1

                for (int k = 0; k < candidate_vec->size(); k++)//for loop
condition sc = n-1
                { // for block sc = 2
                    bestArmor_vec->push_back(candidate_vec->at(k));// sc 2
                }
            }
        }
    }

    return bestArmor_vec; //sc 1
}
```

Exhaustive Max Defense Algorithm Step Count = 1+1+1+4+for_block_outer +1

= 8+$(2^n - 1) * for\_block\_inner\_sc$ +1

= $8 + (2^n - 1) * (15n-6)+1$

$= 8 + 2^n 15n - 6 * 2^n - 15n + 6 + 1 = 15n2^n - 6(2^n) - 15n + 15$

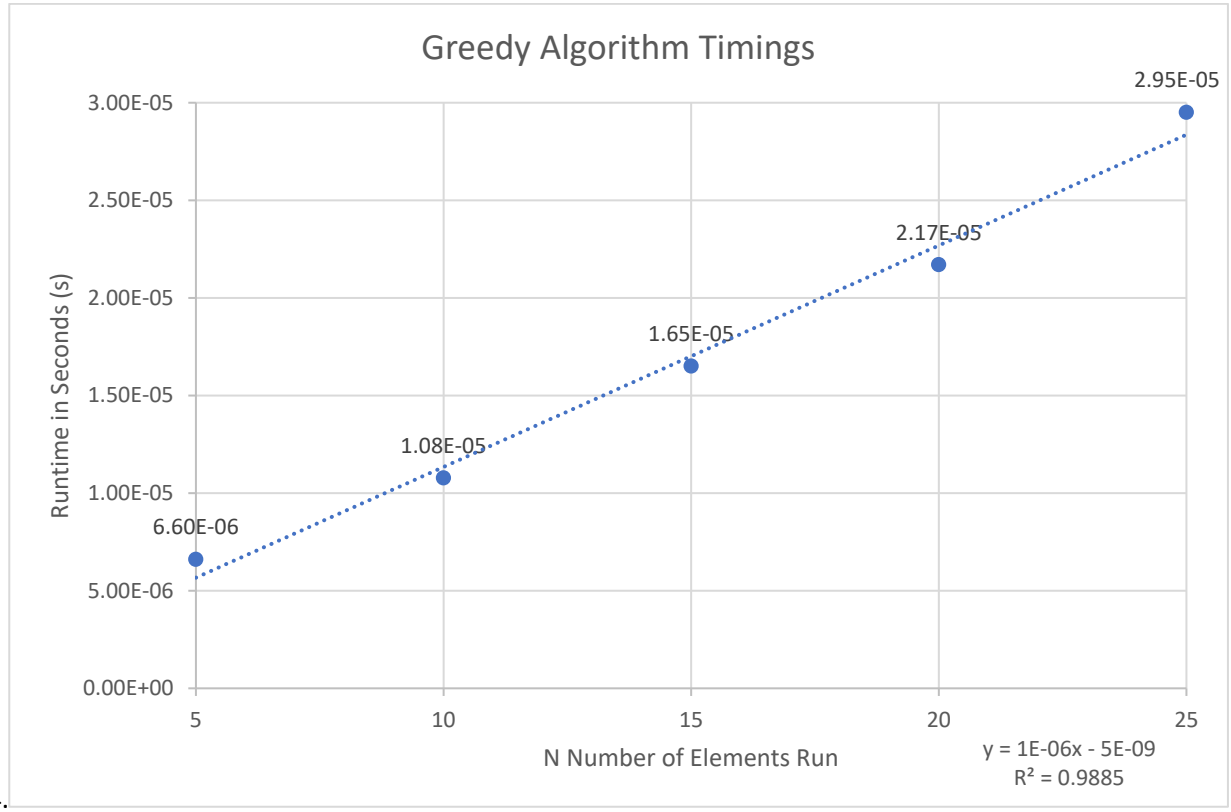
➔ $O(2^n * n)$

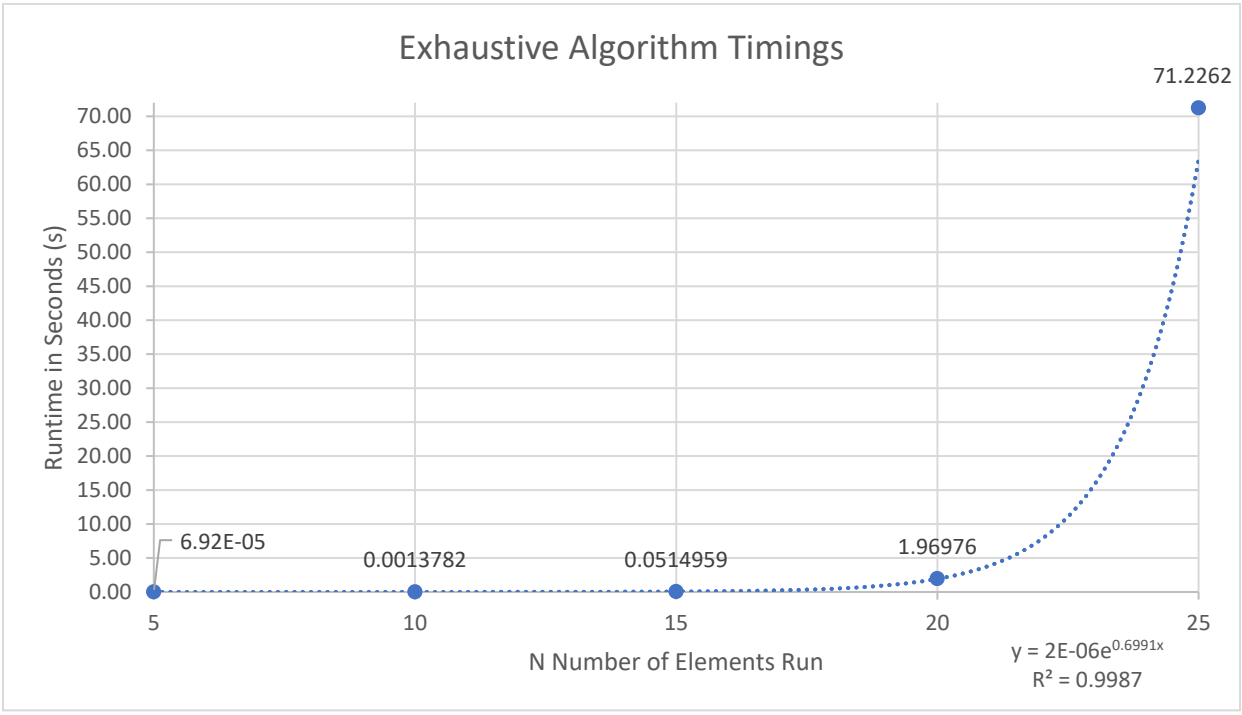
3.

Number of Elements (n)	Greedy Max Defense Timings (s)	Exhaustive Max Defense Timings (s)	Avgerage Greedy Max Defense Timings (s)	Avgerage Exhaustive Max Defense Timings (s)
25	2.98E-05	71.3576		
25	2.93E-05	71.0597	2.95E-05	71.2262
25	2.93E-05	71.2612		
20	2.15E-05	1.97926		
20	2.15E-05	1.95939	2.17E-05	1.96976
20	2.20E-05	1.97063		
15	1.92E-05	0.0519856		
15	1.50E-05	0.0514837	1.65E-05	0.0514959
15	1.53E-05	0.0510185		
10	1.08E-05	0.0013927		
10	1.09E-05	0.0013619	1.08E-05	0.0013782
10	1.06E-05	0.0013799		
5	6.20E-06	7.81E-05		
5	6.80E-06	6.56E-05	6.60E-06	6.92E-05
5	6.80E-06	6.39E-05		

CPSC 335
Project 2 – Greedy and Exhaustive Search
Louis Zuckerman
CCWID: 888206315
Email: louiszman@csu.fullerton.edu

4.





CPSC 335
Project 2 – Greedy and Exhaustive Search
Louis Zuckerman
CCWID: 888206315
Email: louiszman@csu.fullerton.edu

5. The empirically gathered greedy algorithm big-O is O(nlogn), whereas the empirically gathered exhaustive algorithm runtime appears to run with a bit-O of O($n * 2^n$).

The Greedy Algorithm graph follows a best fit of a polynomial which is close to that of O(nlogn) and the Exhaustive Algorithm graph follows a best fit of an exponential which is entirely in line with expectations with a big-O of O($n * 2^n$).

If we take the data from Greedy Algorithm Timings and find the ratio difference between sequential timings, we can show the Big O time efficiency has similar ratios.

$$Greedy\ Empirical\ Percent\ Change\ at\ n = 25: n = 20 - \frac{2.95e-5 - 2.17e-5}{2.95e-5} * 100 = 26.1_7$$

$$Greedy\ Ideal\ Percent\ Change\ at\ n = 25: n = 20 - \frac{25 * \log(25) - 25 * \log(25)}{25 * \log(25)} * 100 = 25.5_4$$

If we take the data from Exhaustive Algorithm Timings and find the ratio difference between sequential timings, we can show the big-O time efficiency has similar ratios.

$$Exhaustive\ Empirical\ Percent\ Change\ at\ n = 25: n = 20 - \frac{71.2262 - 1.96976}{71.2262} * 100 = 97.49_2$$

$$Exhaustive\ Ideal\ Percent\ Change\ at\ n = 25: n = 20 - \frac{2^{25} * 25 - 2^{20} * 20}{2^{25} * 25} * 100 = 97.5$$

Thus, I can conclude that my empirically observed time efficiency data is consistent with my mathematically derived big-O efficiency class for each algorithm.

Questions

1. There is an incredibly noticeable difference between the two algorithms. The greedy search algorithm runs much faster than the exhaustive search algorithm at n > 5, whereas when n<=5 the exhaustive search algorithm appears to be superior in speed. The disparity truly shines at n>20 where the exponentiation of the exhaustive search's time complexity shows and the time to compile skyrockets, whereas the time to compile for the greedy algorithm stays relatively the same.
   At n=25 the greedy algorithm executes in 2.96e-5 seconds, whereas the exhaustive algorithm executes in 71.2262 seconds. This difference in speed is incredibly and definitely surprising. It truly gives me a new appreciation for the difference between big-O efficiency classes.

2. Hypothesis 1 stated: Exhaustive search algorithms are feasible to implement and produce correct outputs. The evidence garnered from the timings data and subsequent scatter plot show that indeed it is feasible to implement and produce correct outputs utilizing an exhaustive search algorithm, albeit with a much worse time complexity than utilizing a greedy algorithm. This is proven by the fact that there is evidence of compilation occurring from the timings data.

CPSC 335
Project 2 – Greedy and Exhaustive Search
Louis Zuckerman
CCWID: 888206315
Email: louiszman@csu.fullerton.edu

Although, at n>25 elements the compute time takes so long as to reach the threshold of feasibility.

3. Hypothesis 2 stated: Algorithms with exponential running times are extremely slow, probably too slow to be of practical use. The evidence from the timings data indicates that this is indeed true because at n=25 or greater the time required to execute the exhaustive search took an inordinate amount of time at 71 seconds compared to the greedy algorithm which took only 2.95e-5 seconds. The feasibility of this algorithm for practical use becomes impractical for n>25.