CPSC 335-01
Project 4
Louis Zuckerman

1. Dynamic Max Defense Code

```
std::unique_ptr<ArmorVector> dynamic_max_defense(
    const ArmorVector &armors,
    int total_cost)
{
    std::unique_ptr<ArmorVector> bestArmor(new ArmorVector);

    // Similar to Knapsack problem, make two arrays, one weighted with gold value
s, other with corresponding defense values
    // Split armors vector into two separate vectors, X(wt) array weight = gold_c
ost of item(), V(val) array value = item_defense()
    // W = total_cost, n = size of armors

    const size_t n = armors.size() + 1;
    const size_t W = total_cost + 1;

    std::vector<std::vector<double>> cache;

    cache.resize(n);

    // initialzie gold cost and defense value vectors to 0
    for (size_t i = 0; i < n; cache[i].resize(W), ++i)
        ;

    for (size_t i = 0; i < W; cache[0][i] = 0, ++i)
        ;

    for (size_t i = 0; i < armors.size(); ++i) // sc dominated by size of armor
vector
    {
        for (size_t j = 0; j < W; ++j) // sc dominated by W
        {
            if (j >= armors[i]->cost())
            {
                cache[i + 1][j] = std::max(cache[i][j], cache[i][j - armors[i]-
>cost()] + armors[i]->defense());
            }
            else
            {
                cache[i + 1][j] = cache[i][j];
            }
        }
    }
```

```
    int max_cache = cache[armors.size()][total_cost];

    auto w = total_cost;

    for (int i = armors.size(); i > 0 && max_cache > 0; i--)
    {
        if (cache[i][w] != cache[i - 1][w])
        {
            bestArmor->push_back(armors.at(i - 1));
            w = w - armors.at(i - 1)->cost();
        }
    }

    return bestArmor;
}
```

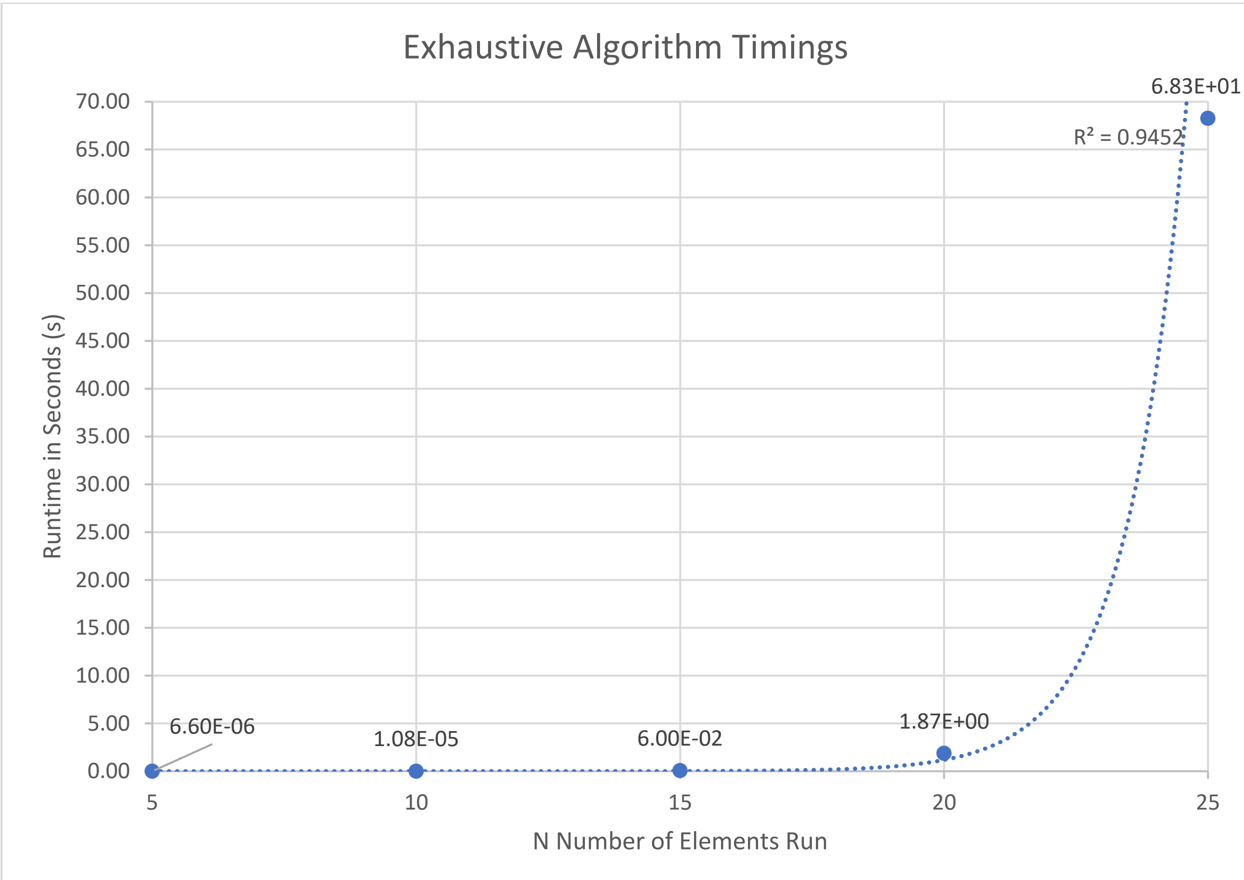
## Big(O)

There are defense * cost total cells in the table. Each cell in the table takes constant time to fill.

The dynamic max defense algorithm reduces to O(n*W) where n is the size of the armors vector and W is the total cost.

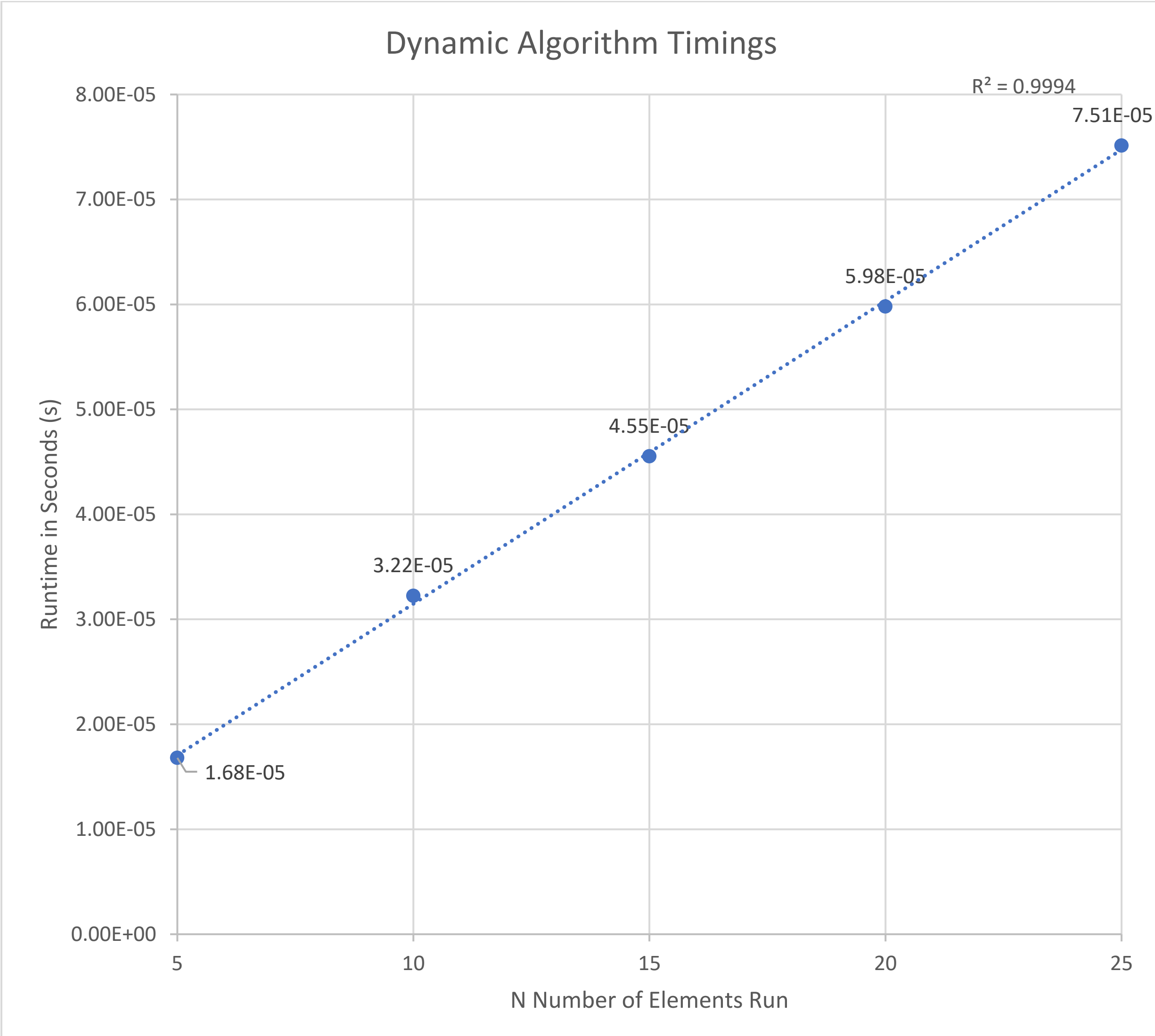
| Number of Elements (n) | Exhaustive Max Defense Timings (s) | Dynamic Max Defense Timings (s) | Average Exhaustive Max Defense Timings (s) | Average Dynamic Max Defense Timings (s) |
|---|---|---|---|---|
| 25 | 68.3254 | 7.55e-05 | | |
| 25 | 68.3666 | 7.46e-05 | 6.83E+01 | 7.51E-05 |
| 25 | 68.0628 | 7.53e-05 | | |
| 20 | 1.87714 | 6.01e-05 | | |
| 20 | 1.86788 | 5.92e-05 | 1.87E+00 | 5.98E-05 |
| 20 | 1.86617 | 6.01e-05 | | |
| 15 | 0.0600902 | 4.54e-05 | | |
| 15 | 0.0598498 | 4.52e-05 | 6.00E-02 | 4.55E-05 |
| 15 | 0.0600943 | 4.6e-05 | | |
| 10 | 0.0132503 | 3.18e-05 | | |
| 10 | 0.0134227 | 3.27e-05 | 1.08E-05 | 3.22E-05 |
| 10 | 0.0131929 | 3.22e-05 | | |
| 5 | 0.0120676 | 1.63e-05 | | |
| 5 | 0.0121679 | 1.69e-05 | 6.60E-06 | 1.68E-05 |
| 5 | 0.0119711 | 1.72e-05 | | |

CPSC 335-01
Project 4
Louis Zuckerman




Follows an exponential curve with $R^2$ =0.9452

CPSC 335-01
Project 4
Louis Zuckerman




Follows a linear curve with $R^2$=0.9994

## Supplementary Analysis:

1. There is an incredibly noticeable difference between the two algorithms. The dynamic search algorithm runs much faster than the exhaustive search algorithm at any value of n. 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 dynamic algorithm stays increases at a linear rate.

   At n=25 the dynamic algorithm executes in 7.51E-05 seconds, whereas the exhaustive algorithm executes in 71.2262 seconds.

CPSC 335-01
Project 4
Louis Zuckerman

2. For larger N, the dynamic algorithm performs exceptionally well and has a constant linear time complexity of $O(n*W)$ versus the exhaustive search algorithm with a time complexity of $O(n*2^n)$.