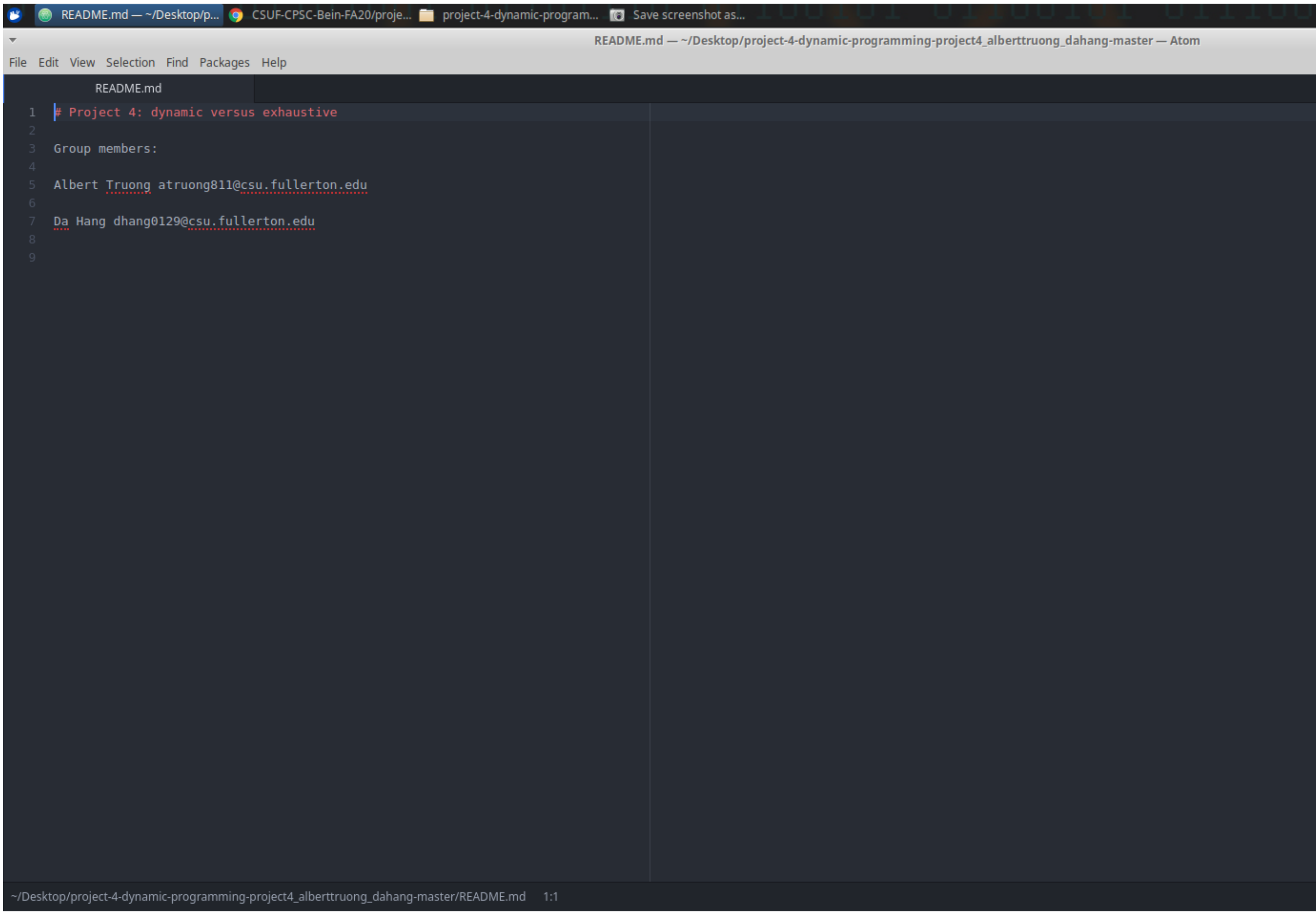
CPSC 335 Project 4 Analysis

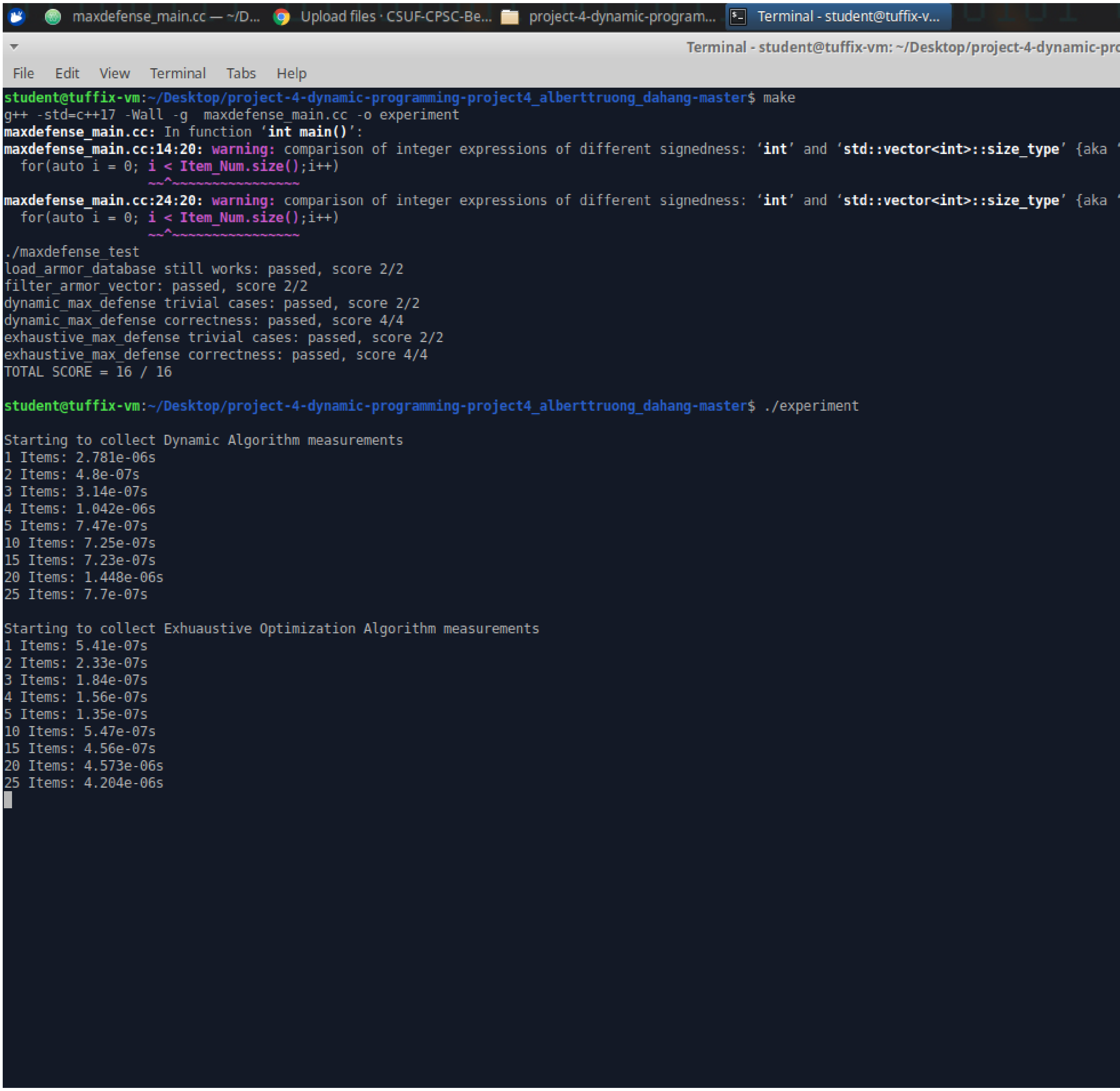
Albert Truong

atruong811@csu.fullerton.edu

Da Hang

dhang0129@csu.fullerton.edu





```
student@tuffix-vm:~/Desktop/project-4-dynamic-programming-project4_alberttruong_dahang-master$ make
g++ -std=c++17 -Wall -g  maxdefense_main.cc -o experiment
maxdefense_main.cc: In function 'int main()':
maxdefense_main.cc:14:20: warning: comparison of integer expressions of different signedness: 'int' and 'std::vector<int>::size_type' {aka '
  for(auto i = 0; i < Item_Num.size();i++)
                  ~~^~~~~~~~~~~~~~~~~
maxdefense_main.cc:24:20: warning: comparison of integer expressions of different signedness: 'int' and 'std::vector<int>::size_type' {aka '
  for(auto i = 0; i < Item_Num.size();i++)
                  ~~^~~~~~~~~~~~~~~~~
./maxdefense_test
load_armor_database still works: passed, score 2/2
filter_armor_vector: passed, score 2/2
dynamic_max_defense trivial cases: passed, score 2/2
dynamic_max_defense correctness: passed, score 4/4
exhaustive_max_defense trivial cases: passed, score 2/2
exhaustive_max_defense correctness: passed, score 4/4
TOTAL SCORE = 16 / 16

student@tuffix-vm:~/Desktop/project-4-dynamic-programming-project4_alberttruong_dahang-master$ ./experiment

Starting to collect Dynamic Algorithm measurements
1 Items: 2.781e-06s
2 Items: 4.8e-07s
3 Items: 3.14e-07s
4 Items: 1.042e-06s
5 Items: 7.47e-07s
10 Items: 7.25e-07s
15 Items: 7.23e-07s
20 Items: 1.448e-06s
25 Items: 7.7e-07s

Starting to collect Exhuaustive Optimization Algorithm measurements
1 Items: 5.41e-07s
2 Items: 2.33e-07s
3 Items: 1.84e-07s
4 Items: 1.56e-07s
5 Items: 1.35e-07s
10 Items: 5.47e-07s
15 Items: 4.56e-07s
20 Items: 4.573e-06s
25 Items: 4.204e-06s
```

Exhaustive Optimized Algorithm:

n = |armor_items| //SC: 1

best = None //SC: 1

```
double totalGold= 0.0 //SC: 1

totalDef=0.0 //SC: 1

totalDefB =0.0 //SC: 1


for bits = 0 to (2^n -1): //SC: ((2^n -1)-0+1) = 2^n

        candidate = empty vector //SC: 1

        for j = 0 to n-1: //SC: (n-1)-0+1 = n

                if (bits >> j) & 1 == 1: //SC: 3

                        candidate.add_back(armor_items[j]) //SC: 1

                        sum_armor_vector(candidate,totalGold,totalDef); //SC: 1

                End if

        End for

        if totalGold <= total_cost //SC: 1

                        if best is None || totalDef > totalDefB  && candidate is not empty //SC: 3

                                best = candidate //SC: 1

                                totalDefB = totalDef //SC: 1

                        End if

        End if

End for

return best
```

SC: $1+1+1+1+1+2^n(1+n(3+\max(1+1,0)) + 1 + \max(3 + \max(1+1,0) , 0))$

$5 + 2^n(1+n(3+2) + 1 + \max(3+2,0))$

$5 + 2^n(1+5n + 1 + 5)$

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

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

$O(2^n * n)$

Dynamic Programming

```
Def dynamic( armor cost):
r = armor.size()
c = cost
cache = initialize 2d array with size
for int i = 1 to r+1
    item_defense = armors[i-1].defense
    item_cost = armors[i-1].cost
    for int j = c+1
        up = cache[i-1][j]
        //if up_left is invalid,continue
        If(j- item_cost < 0)
            Cache[i][j]: up left
            Continue
        End if
        Up_left = cache[i-1][j-item_cost]
        Up_left_total = up_left + item_defense
        Cach[i][j] = max(up, up_left_total)
    End for
End for
//Start-Over
i = r-1
```

```
j = c-1
items = new ArmorVector
while(i>0 && j>0)
    item_defense = armors[i+1].defense
    item_cost = armors[i-1].cost
    up = cache[i-1][j]
        // if up_left column
        If( j-item_cost < 0)
            i--
        end if
    up_left = cache[i - 1][j-item_cost]
    up_left_total = up_left + item_defense
        if (up < up_left_total)
            items.add(armors[i-1])
            i--
            j=j-item_cost
        else
            i--
        end if
end while
return items
```

SC: $3 + \sum_{i=1}^{r+1}[2 + \sum_{j=1}^{c+1}(7)]$

$= 3 + (\sum_{i=1}^{r+1}[2 + 7(c+1)])$

$= 3 + \sum_{i=1}^{r+1}[7c + 9]$

=3 + (r+1)(7c+9)

= 7rc + 21r + 7c

Since number of r, and number of c is the same. We set r and c to n

= $7n^2 + 21n + 7n$

=$n^2 + 28n$

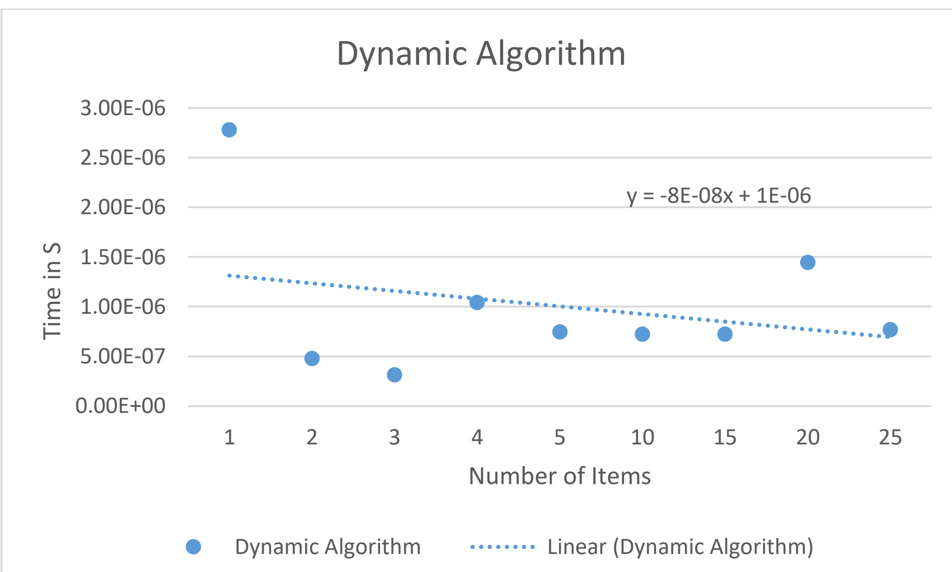
O(n^2)

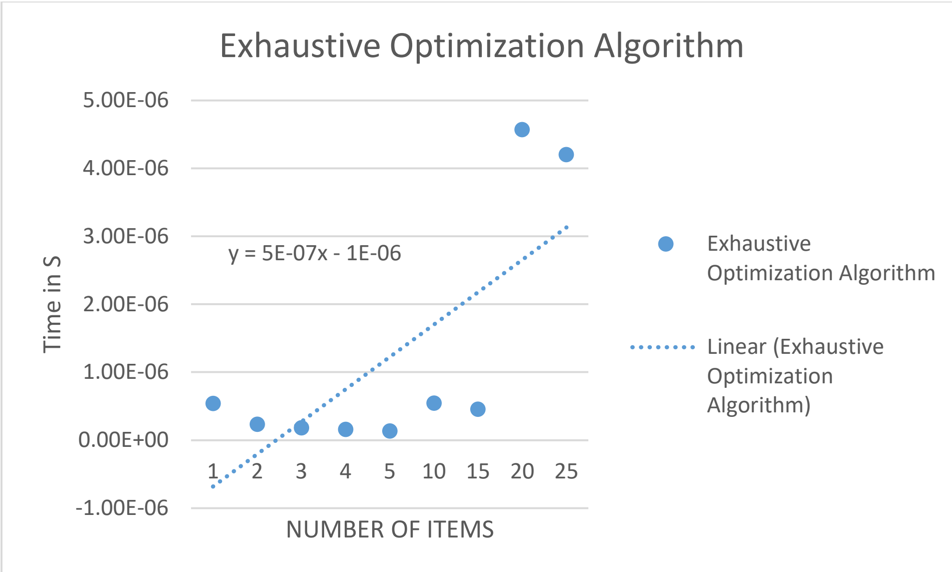
Scatter Plot Graphs:

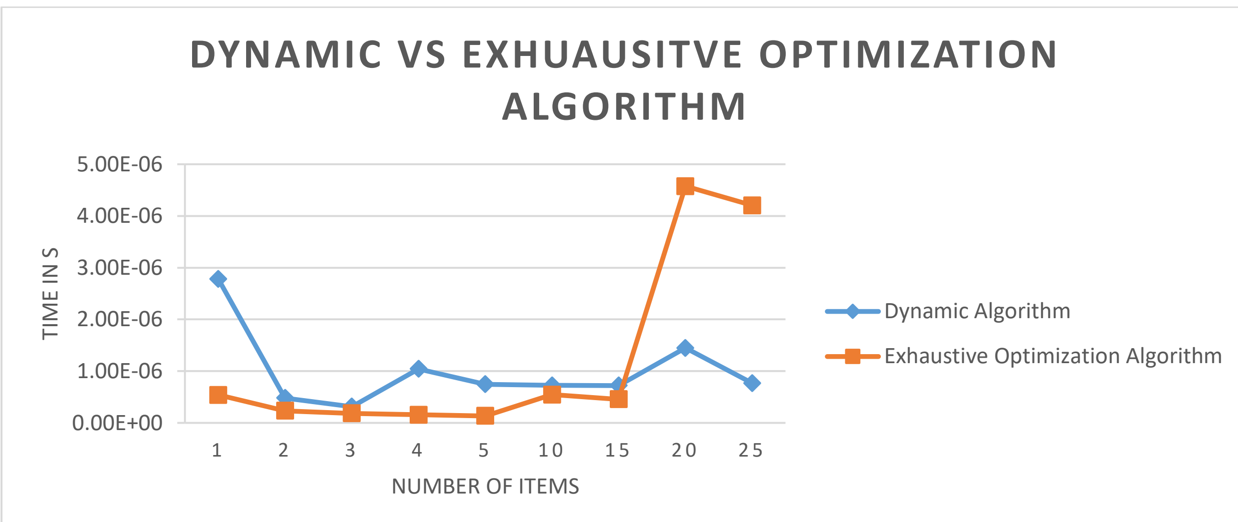
Data:

	1	2	3	4	5	10	15	20	25
Dynamic Algorithm	2.78E-06	4.80E-07	3.14E-07	1.04E-06	7.47E-07	7.25E-07	7.23E-07	1.45E-06	7.70E-07
Exhaustive Optimization Algorithm	5.41E-07	2.33E-07	1.84E-07	1.56E-07	1.35E-07	5.47E-07	4.56E-07	4.57E-06	4.20E-06

Plot:







Questions:

a. There is a noticeable difference in the performance of the two algorithm. It only becomes noticeable once the size of n goes over 15. Other than the spike after 15 items in the exhaustive search where it may be explained by a background process slowing the computer, at 20 or 25 items the process may be 4 times slower than the dynamic algorithm. When comparing the mathematically-derived big O efficiency class for each algorithm $O(2^n * n)$ vs $O(n^2)$ it is not surprising that the $O(2^n * n)$ of the Exhaustive Optimization Algorithm would be the slower algorithm.

b. The empirical analyses are not consistent with the mathematical analyses of the dynamic algorithm but may be consistent with the exhaustive search algorithm. The reason for this is most likely due to the size of n chosen for the graphs. When doing the empirical analysis, it was determined that the max size of n would be chosen to be 25 due to the time it would take to obtain the data above 25. That size may be too small to obtain an empirical analyses data that are consistent with the mathematical analyses.
c. Based on the graph of the Dynamic Algorithm, the empirically-observed time efficiency data is inconsistent, with the mathematically-derived big O efficiency class for the algorithm. It may be because of background process slowing the first test of n. The size of n may not be large enough to obtain an accurate look at the big O efficiency. The top size of n at 25 was chosen because above 25 the time it took for exhaustive optimization algorithm was too long.
d. Based on the graph of the Exhaustive Optimization Algorithm, the empirically-observed time efficiency data is potentially consistent. As the size of n went up it a considerable amount was above 15. At size 20 there appears to be a spike that may have been caused by a background process slowing the computer. Above that at 25 there were a slight dip in time. But the overall graph is climbing exponentially which is relatively consistent with the big O efficiency class of $O(2^n * n)$. If the size of n went above 25, it would be graphically more consistent with the Big O, but due to amount of time it took above 25. It was determined that n of 25 would be the largest size of n.