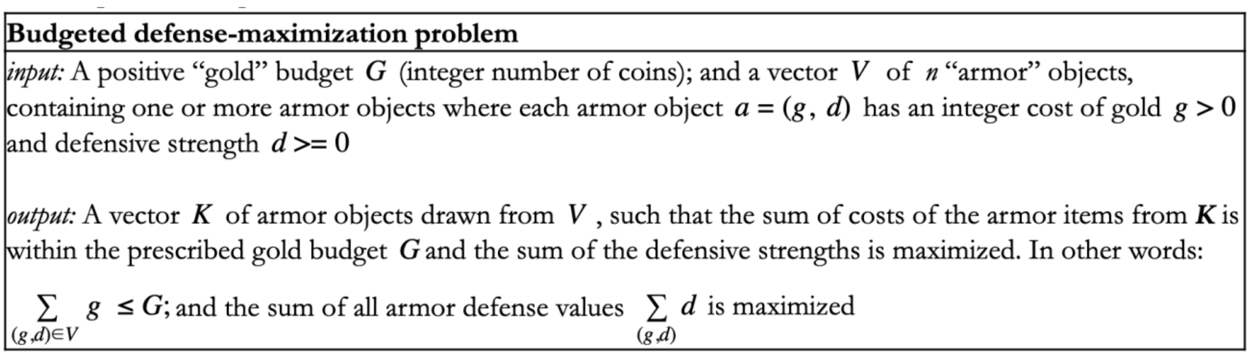
Project 4 – Dynamic vs. Exhaustive

Dynamic and Exhaustive algorithms solve searching in O(n•m) and O(2^n • n) time respectfully. The following report details and examines these algorithms' time complexity.



Budgeted defense-maximization problem

input: A positive "gold" budget G (integer number of coins); and a vector V of n "armor" objects, containing one or more armor objects where each armor object $a = (g, d)$ has an integer cost of gold $g > 0$ and defensive strength $d >= 0$

output: A vector K of armor objects drawn from V, such that the sum of costs of the armor items from K is within the prescribed gold budget G and the sum of the defensive strengths is maximized. In other words:

$\sum_{(g,d)\in V} g \leq G$; and the sum of all armor defense values $\sum_{(g,d)} d$ is maximized

Hypotheses:

1) Exhaustive search algorithms are feasible to implement, and produce correct outputs.
2) Algorithms with exponential running times are extremely slow, probably too slow to be of practical use.

Part 1: Description of the algorithms

1. **Dynamic Algorithm**: Using the knapsack algorithm we are able to get the item(s) with the best defense/cost (value/weight). This is done by creating a cache that contains each maximum defense/cost of each item. The cache is a 2d vector/matrix with *i* number of rows and *j* number of columns. The algorithm starts by populating the cache from bottom right to top left. The knapsack calculation is calculated by comparing the bottom right item, the item above and the item above left. If the item above is equal to the item above left the item is added to the cache via push_back and the column is updated by subtracting the item weight. This will continue to repeat until the total cost reaches 0 .The last row and the last column will return the item with the best defense/cost. The algorithm returns a list of the items that have the best defense/cost which when added will return the best defense.

2. **Exhaustive Algorithm**: By implementing the exhaustive optimized algorithm, we are able to find the best and candidate best armor defense per cost at a given size (n < 64). We solve this by generating subsets of the vector and compute the cost while also comparing against the best defense and cost. Each candidate is enumerated and later verified or accepted to see if that candidate has the best defense per cost in the enumerated list. From there, we achieve in creating the best subset from the vector data and are returned back to the caller.

Part 2: Pseudocode and step count analysis:

1. **Dynamic Algorithm**:

```
                                    total size m
dynamic_max_defense(total_cost, armors):
    source = armor_items   unsorted size n // 1
    finish = Empty Vector // 1
    cache[][] = None // 1
    double result = 0.0 // 1
    double tempCost = total_cost // 1
```

5

```
    for i from 0 to armors.size():          n
        cache.append(vector<double>());
        for i from 0 to total_cost+1:       m
            cache[i].append(0.0); // 1
```

$\rightarrow \left(\frac{n-0}{1}+1\right) = (n+1)(\underset{\uparrow}{1}) = n+1$ inner block

$\left(\frac{m+1-0}{1}+1\right) = m+2\,(1) = m+2$ inner block

$(n+1)(m+2) = nm+2n+m+2$

```
    for i from 1 to armors.size():          n
        for j from 1 to total_cost:         m
            shared_ptr<ArmorItem> item = armors[i-1]; // 1
            if(j - item->cost() >= 0) // 2
                result = max(item->defense() + cache[i-1][j-item->cost()], cache[i-1][j]) // 5
            else
                result = cache[i-1][j]; // 2
            cache[i][j] = result; // 1
```

$\left(\frac{n-1}{1}+1\right) = n$

$\left(\frac{m-1}{1}+1\right) = m$

$2+\max(5,2) = 7+1 = 8+1 = 9$

$n \cdot m \cdot 9$

```
    for i from armors.size() to 0:          n
        if(cache[i][tempCost] == cache[i-1][tempCost]) // 2
            continue; // 1
        else
            finish->push_back(source->at(i-1)); // 2
            tempCost -= (armors.at(i-1))->cost(); // 2
```

$\left(\frac{0-n}{-1}+1\right) = n+1$

$2+\max(1,4) = 6$

$(n+1)(6) = 6n+6$

```
    return finish // 1
```

1

$$SC = \underbrace{5}_{\text{"identifiers"}} + \underbrace{[n \cdot m + 2n + m + 2]}_{\text{"First for-loop"}} + \underbrace{9 \cdot n \cdot m}_{\text{"2nd for-loop"}} + \underbrace{[6n+6]}_{\text{"3rd for-loop"}} + \underbrace{1}_{\text{"return"}}$$

$$SC = \boxed{10n \cdot m + 8n + m + 14} \rightarrow O(n \cdot m)$$

Proof by Definition

$f(n) = 10n \cdot m + 8n + m + 14$

Prove $f(n) \in O(n \cdot m)$

Find a value $c > 0$ and $n_0 > 0$ s.t. $10n \cdot m + 8n + m + 14 \leq c \cdot n \cdot m \; \forall \; n > n_0$

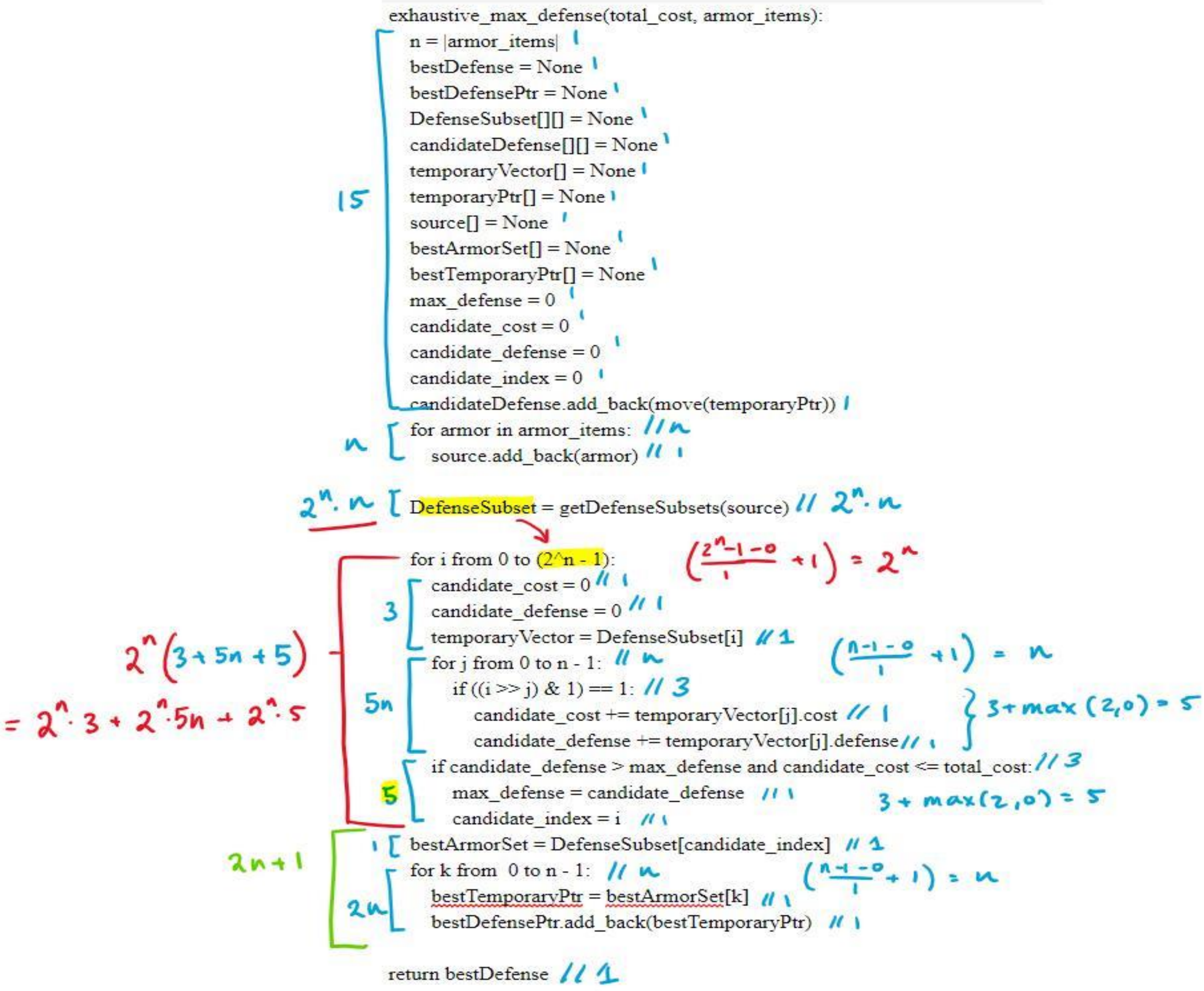
Choose $C = 10 + 8 + 1 + 14 = 33$

$10n \cdot m + 8n + m + 14 \leq 33 \cdot n \cdot m$ true for $n \geq 1$

$10n \cdot m \leq 33 \cdot n \cdot m$

choose $n_0 = 1$

2. **Exhaustive Algorithm:**



```
exhaustive_max_defense(total_cost, armor_items):
    n = |armor_items| 1
    bestDefense = None 1
    bestDefensePtr = None 1
    DefenseSubset[][] = None 1
    candidateDefense[][] = None 1
    temporaryVector[] = None 1
    temporaryPtr[] = None 1
    source[] = None 1
    bestArmorSet[] = None 1
    bestTemporaryPtr[] = None 1
    max_defense = 0 1
    candidate_cost = 0 1
    candidate_defense = 0 1
    candidate_index = 0 1
    candidateDefense.add_back(move(temporaryPtr)) 1
    for armor in armor_items: // n
        source.add_back(armor) // 1

    DefenseSubset = getDefenseSubsets(source) // 2^n · n

    for i from 0 to (2^n - 1):
        candidate_cost = 0 // 1
        candidate_defense = 0 // 1
        temporaryVector = DefenseSubset[i] // 1
        for j from 0 to n - 1: // n
            if ((i >> j) & 1) == 1: // 3
                candidate_cost += temporaryVector[j].cost // 1
                candidate_defense += temporaryVector[j].defense // 1
        if candidate_defense > max_defense and candidate_cost <= total_cost: // 3
            max_defense = candidate_defense // 1
            candidate_index = i // 1
    bestArmorSet = DefenseSubset[candidate_index] // 1
    for k from 0 to n - 1: // n
        bestTemporaryPtr = bestArmorSet[k] // 1
        bestDefensePtr.add_back(bestTemporaryPtr) // 1

return bestDefense // 1
```

15

n

$2^n \cdot n$

$\left(\frac{2^n-1-0}{1}+1\right) = 2^n$

3

$2^n(3+5n+5)$

$= 2^n \cdot 3 + 2^n \cdot 5n + 2^n \cdot 5$

5n

$\left(\frac{n-1-0}{1}+1\right) = n$

$\}\ 3 + \max(2,0) = 5$

5

$3 + \max(2,0) = 5$

1

$2n+1$

2n

$\left(\frac{n-1-0}{1}+1\right) = n$

"identifiers" "source" "subset func" "For-block" "if-block" "return"

$$SC = 15 + n + 2^n \cdot n + [3 \cdot 2^n + 2^n \cdot 5n + 5 \cdot 2^n] + (2n+1) + 1$$

$$SC = \boxed{2^n \cdot 5n + 2^n \cdot 8 + 3n + 17} \rightarrow O(2^n \cdot n)$$

Proof by definition

$f(n) = 2^n \cdot 5n + 2^n \cdot 8 + 3n + 17$

Prove $f(n) \in O(2^n \cdot n)$

Find a value $c > 0$ and $n_0 \geq 0$ s.t. $2^n \cdot 5n + 2^n \cdot 8 + 3n + 17 \leq c \cdot 2^n \cdot n \ \forall\ n > n_0$

Choose $c = 5 + 8 + 3 + 17 = 33$

$2^n \cdot 3n + 2^n \cdot 8 + 3n + 17 \leq 33 \cdot 2^n \cdot n$ true for $n \geq 1$

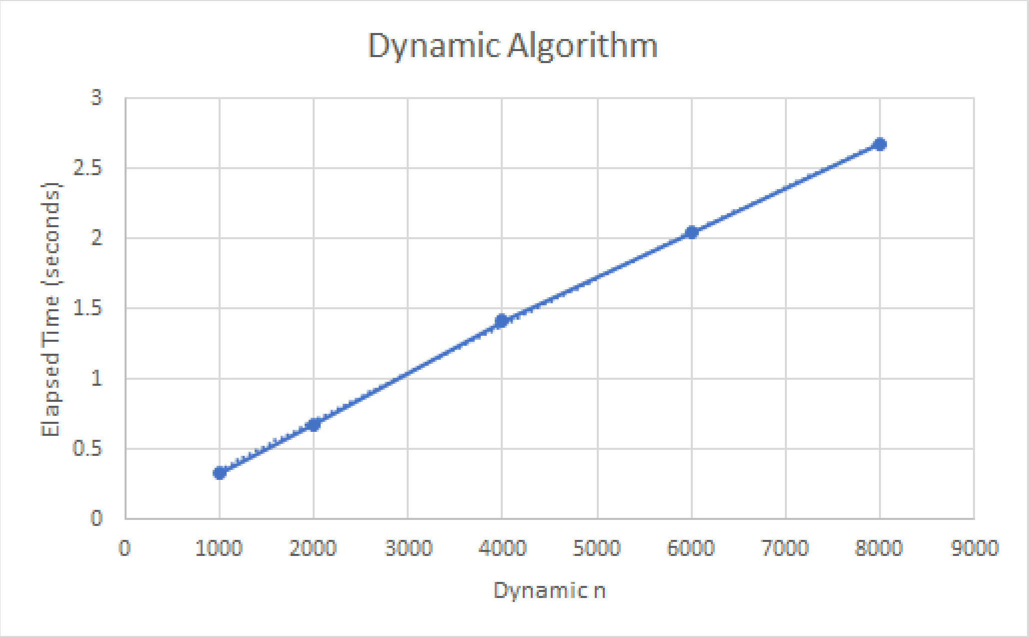
$2^n \cdot 3n \leq 2^n \cdot 33n$

Choose $n_0 = 1$

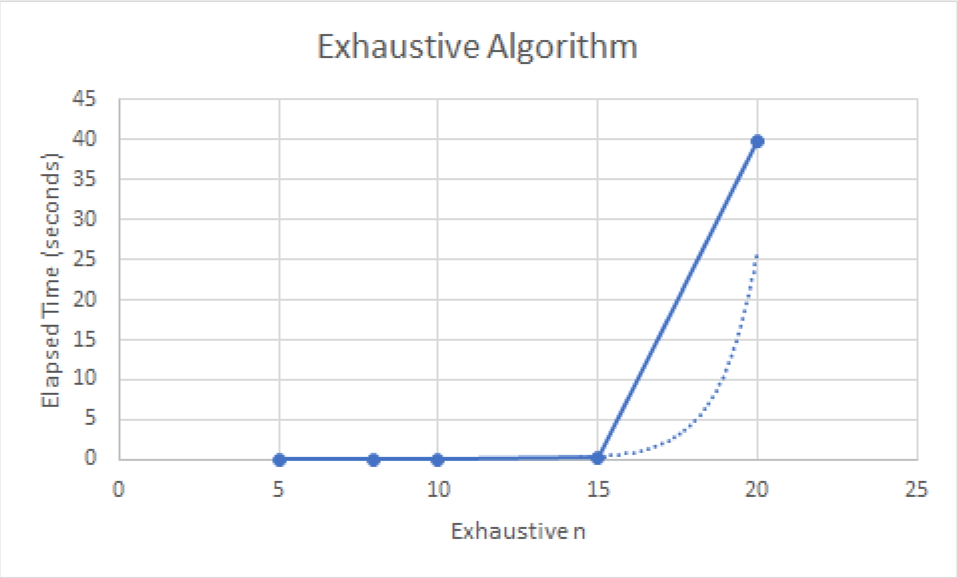
Part 3: Empirical Analysis: Dynamic and Exhaustive Algorithm

Each implemented with a defense interval of 1 to 3000 and a total cost of 2000.

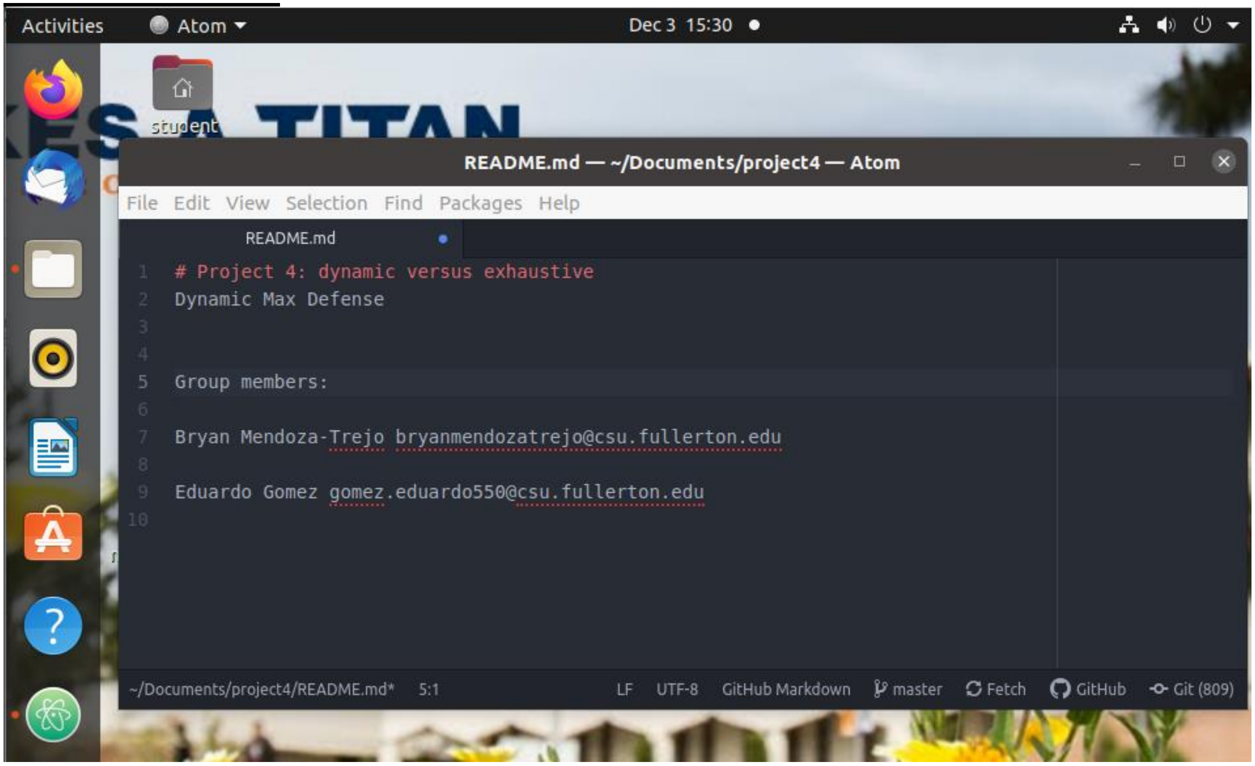
Dynamic n	Elapsed Time (seconds)
1000	0.329716
2000	0.67625
4000	1.41933
6000	2.04733
8000	2.67841



Exhaustive n	**Elapsed Time (seconds)**
5	0.000082877
8	0.000831356
10	0.00377903
15	0.2043
20	39.7663



Part 4: Tuffix Proof


Activities
Atom
Dec 3 15:30
student
README.md — ~/Documents/project4 — Atom
File Edit View Selection Find Packages Help
README.md
~/Documents/project4/README.md* 5:1
LF UTF-8 GitHub Markdown master Fetch GitHub Git (809)

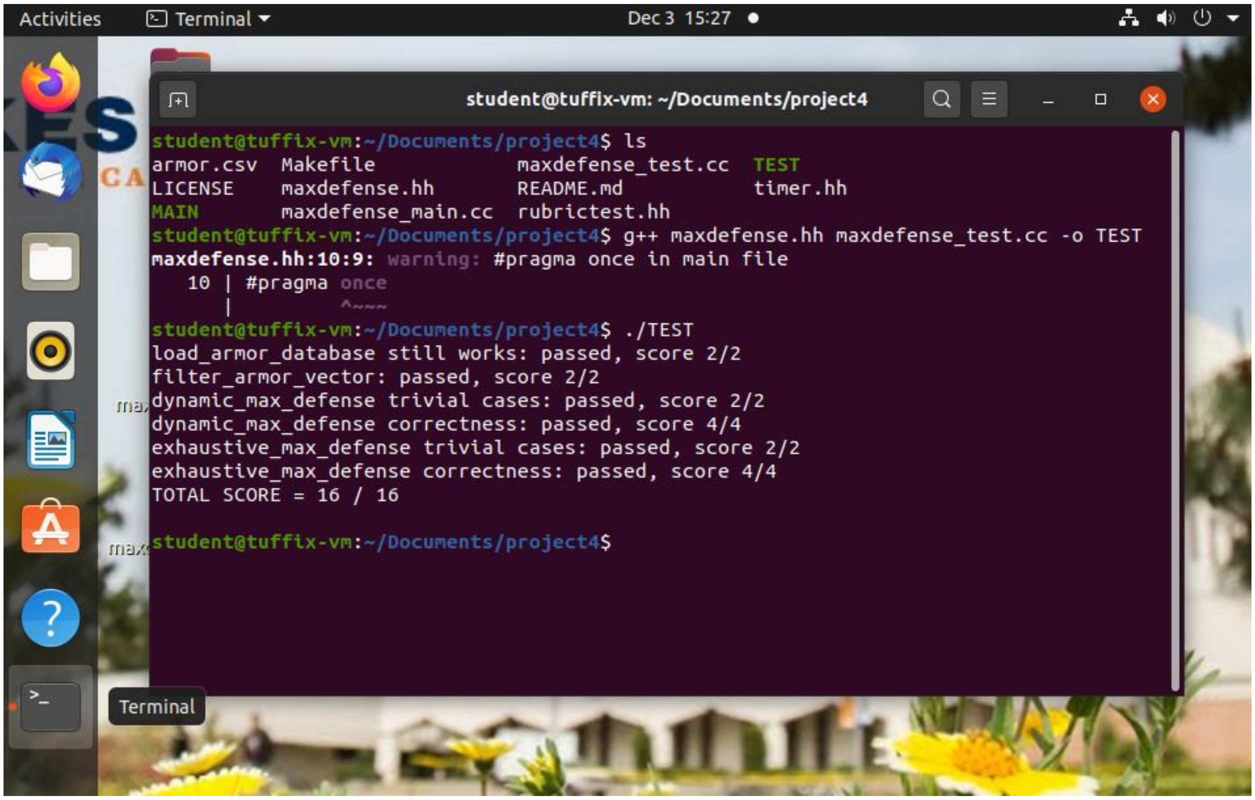
```
# Project 4: dynamic versus exhaustive
Dynamic Max Defense


Group members:

Bryan Mendoza-Trejo bryanmendozatrejo@csu.fullerton.edu

Eduardo Gomez gomez.eduardo550@csu.fullerton.edu

```



```
student@tuffix-vm:~/Documents/project4$ ls
armor.csv  Makefile           maxdefense_test.cc  TEST
LICENSE    maxdefense.hh      README.md           timer.hh
MAIN       maxdefense_main.cc  rubrictest.hh
student@tuffix-vm:~/Documents/project4$ g++ maxdefense.hh maxdefense_test.cc -o TEST
maxdefense.hh:10:9: warning: #pragma once in main file
   10 | #pragma once
      |         ^~~~
student@tuffix-vm:~/Documents/project4$ ./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:~/Documents/project4$
```

Part 5: Questions

1) Is there a noticeable difference in the performance of the two algorithms? Which is faster, and by how much? Does this surprise you?

   Dynamic algorithm is much faster than Exhaustive algorithm as expected from the time complexity difference. The dynamic algorithm is able to have a larger *n* because of its n•m growth but on the other hand the exhaustive algorithm can have *n* <= 15 before its 2^n growth is too long to compute. This isn't surprising because dynamic grows n•m while exhaustive grows 2^n.

2) Are your empirical analyses consistent with your mathematical analyses? Justify your answer.

   The dynamic scatter plot data shows a growth resembling a linear growth which can be expected from the mathematical analysis. The mathematical analysis states the time complexity to be O(n•m) which is consistent with the scatter plot best fit line. The exhaustive scatterplot data shows a 2^n growth. The graph shows a significant growth from n=8000 to n=20 which is consistent with the mathematical analysis that states the time complexity to be O(2^n • n**).**

3) Is this evidence consistent or inconsistent with hypothesis 1? Justify your answer.

   Our first inference of implementing the exhaustive search algorithm to solve our problem was that it would take longer to find the correct outputs. After implementing this exhaustive pattern, we noticed that our evidence is consistent with our hypothesis 1 for the most part. Although the algorithm does produce correct outputs it was not feasible to implement because of its greater time complexity for larger sizes of n comparable to a dynamic pattern.

4) Is this evidence consistent or inconsistent with hypothesis 2? Justify your answer.

   The evidence is consistent with hypothesis 2 because algorithms with exponential time complexity are very slow as the given *n* gets larger. For example, the dynamic algorithm whose time complexity is O(n•m) grows almost linearly but the exhaustive algorithm whose time complexity is O(2^n * n) grows faster than exponentially. This results in exhaustive being too slow to be used since any *n* larger than 15 has a significantly slower result time. For example, when n=8000 dynamic takes 2.67 seconds vs when n=15 exhaustive takes 0.236033 seconds. This clearly shows that dynamic is much faster than exhaustive.