Jessica Vargas
mvrgsj@csu.fullerton.edu
Dylan Nguyen
dylan-nguyen@csu.fullerton.edu

Team Yuh:Project 2 Analysis

1. Filter Armor Vector

```
def filter_armor_vector(source, min_def, max_def, total_size):

    FilteredArmor = source |1

    for i=0 to source.size() do |1
        if (source[i].defense > 0 AND FilteredArmor.size < total_size) |1
            if (min_def < source[i].defense AND source[i].defense <= max_def) |1
                FilteredArmor.add_back(source[i]) |1
            endif
        endif
    endfor
    return FilteredArmor |1
```

Complexity:

Final - Initial / Step + 1
(n-0)/1 + 1 = n+1
4(n+1) =4n+5 = **O(n)**

Mathematical Analysis:

In the Filter_Armor_Vector function, it has the complexity of O(n) due to our singular for-loop. We start from 0 to n with 4 steps inside the loop. Using our given formula, we send with 4((n-0)/1 +1) which simplifies to 4(n+1). Then we add the two steps that are not within the for loop, and we end 4(n+1) +2 which equals 4n+6. We drop to the dominant term 4n, which gives the complexity of O(n).

2. Greedy Max Defense

```
def greedy_max_defense(armors, total_cost):
    todo = armors |1
    result = None |1
    budget=0 |1
    index=0 |1
```

```
while(todo NOT empty AND budget<=total_cost)
        max=0 |1
        for i=0 to todo.size() do |1
                next = (todo[i].defense/todo[i].cost) |1
                if(max < next) |1
                        index = i |1
                        max = next |1
                endif
        endfor

        g = todo[index].cost |1
        if((g + budget) <= total_cost) |1
                result.add_back(todo[index]) |1
                budget += todo[index].cost |1
        endif
        todo[index].erase |1
endwhile
return result |1
```

Complexity:

For loop calculated by (Final-Initial)/1 + 1
(m-0)/ 1 + 1 = m-1
5(m-1) = 5m-5

Sigma 0 to n-1 (7 + (5m-5))
Sigma 0 to n-1 (5m + 2)
5m((n-1) - 0 + 1) + 2((n-1) - 0 + 1)
5m(n-1) + 2(n-1)
5mn-5m+2n-2 → mn-m+n → mn → **O(n^2)**

Mathematical Analysis:
Our function uses nested loops. The outer loop is a while-loop that proceeds when list todo is not empty and the budget is less than or equal to the total_cost. The inner loop is a for-loop that goes from i to the size of todo, or m. Starting with the inner for loop, the steps are found to be 5m-5. The while loop starts from 0 to n-1. Calculating the inner loop plus the while loop results in 5mn-5m+2n-2. Dropping multiplicative values results in mn-m+n. Dropping to the dominant term results in mn. Assuming that m and n become a similar value given that budget <= total_cost and worst case scenario of a while loop, mn can be converted to nn, which is n^2. Therefore, the greedy takes O(n^2).

3. Exhaustive Max Defense

```
# greedy
bestset = None |1
candidate = None |1
for x in subSetAmt: |1

Set candidate = None |1
  for armor in armory:  |1

          if index & 1 << armor|1
            candidate = armor|1
          endif
endfor
          sum_armor_vector(candidate)|1
  if defense > best && budget < total_cost:|1
          best = candidate|1
          endif
endfor
          return best|1
```

SC= Sigma 0 to n (4+ Sigma 0 to m (2(m-0+1)))

3+ (n-0 (4+ 2m+1))= 3+4n+2mn +n-> 2mn + 5n +3 ->n = 2^n

= 2 *2^n * n = **O(n*2^n)**

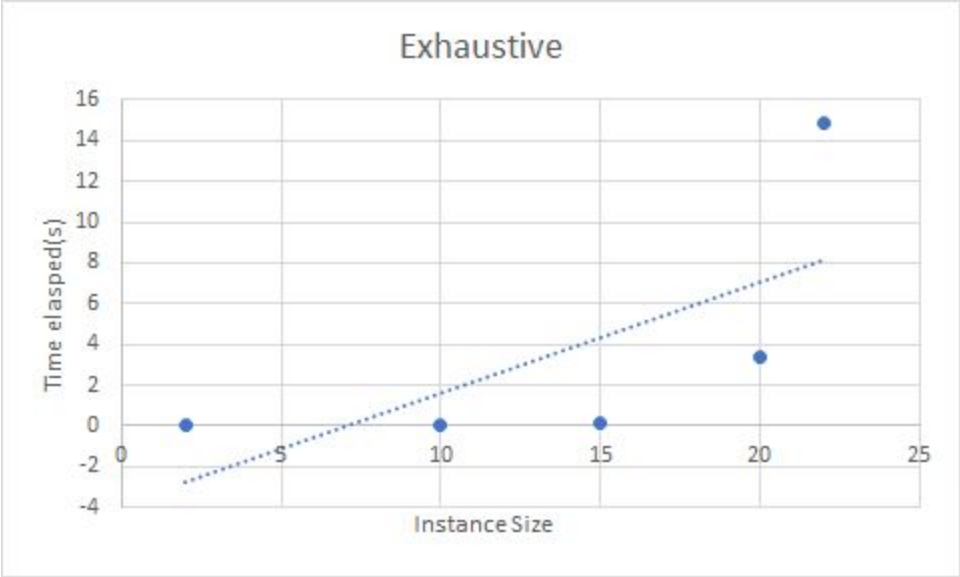
Mathematical Analysis:

For the loop 0 to N, there is another nested “for-loop” for 0 to m. After calculating the inner “for-loop” steps to be 2m+1, we add that to the steps of the outer “for-loop” and come out with a subvalue of 2mN +5N + 3. If we take a look at n represents, N is the amount of subsets that are being made for the list. This value will always be 2^the amount of items in the list, or for short 2^n. M simply represents the amount of items in a list. Thus we can convert 2mN +5N + 3 to 2(2^n)m + 5(2^n) + 3. Dropping to the dominant term we get 2^n * m, or rather O(n * 2^n) as n and m are equivalent in this instance.

4. Graph

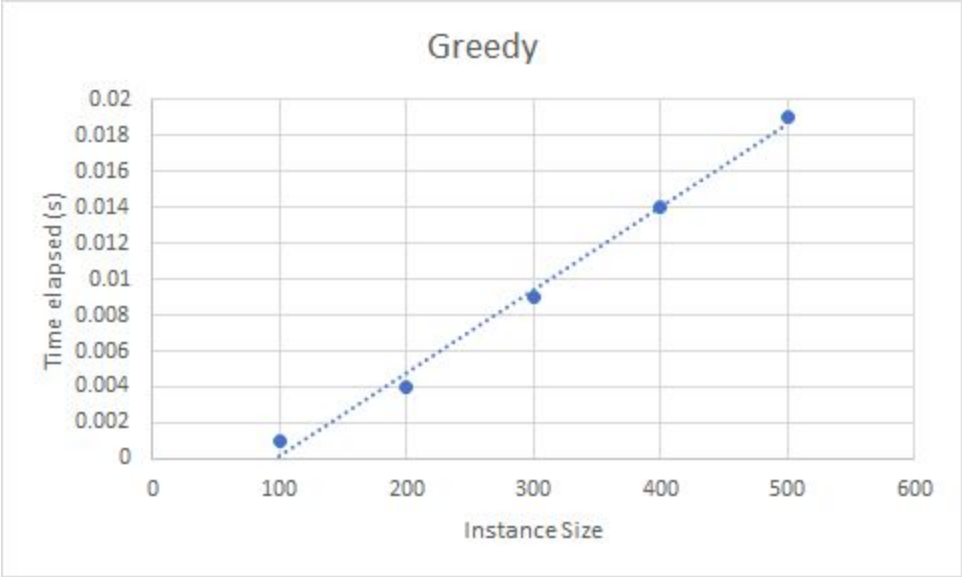
Exhaustive:

Items	Time(s)
2	0
10	0.003003
15	0.096219
20	3.38874
22	14.85



Greedy:

Items	Time(s)
100	0.001
200	0.003999
300	0.009001
400	0.014002
500	0.019002



5. Conclusion

Answers to the following questions, using complete sentences.

a. Is there a noticeable difference in the performance of the two algorithms? Which is faster, and by how much? Does this surprise you?
Yes there is a noticeable difference in the performance of our two algorithms. For all sizes, greedy tends to be the faster executing algorithm, by a significant amount. No, because the time complexity of our greedy algorithm is O(n^2), while our exhaustive is O(n*2^n). We used the complexities to predict that greedy would generally be faster and it has proven true.

b. Are your empirical analyses consistent with your mathematical analyses? Justify your answer.
As stated briefly above, and with our graphs, the data collected is consistent with our predictions on speed and with our mathematical analysis. The best fit lines are consistent with the time complexities that were found for each algorithm, that of which being O(n^2) and O(n*2^n). The best fit of the exhaustive follows the expectations that it will be exponential, while the best fit of the greedy follows a linear path.

c. Is this evidence consistent or inconsistent with hypothesis 1? Justify your answer.
We have created an exhaustive search algorithm that is implementable and is able to create correct outputs. So the evidence is consistent with hypothesis 1. Our exhaustive search algorithm goes through every possible set of a given power set and chooses the optimal subset available.

d. Is this evidence consistent or inconsistent with hypothesis 2? Justify your answer.
Hypothesis 2 states that algorithms with exponential run times are too slow for practical use. This hypothesis is mostly true. Using exponential run time algorithms are still practical for small sets of inputs, but for greater sizes it is inefficient. One can assume an algorithm to be of “practical use,” meaning all inputs will be ran efficiently, so yes, the evidence is consistent that exponential run time algorithms are too slow for full practical use.