Michael Gresham

Greshammichael@csu.fullerton.edu

Project 2: Maximizing Armor Strength

Greedy Algorithm

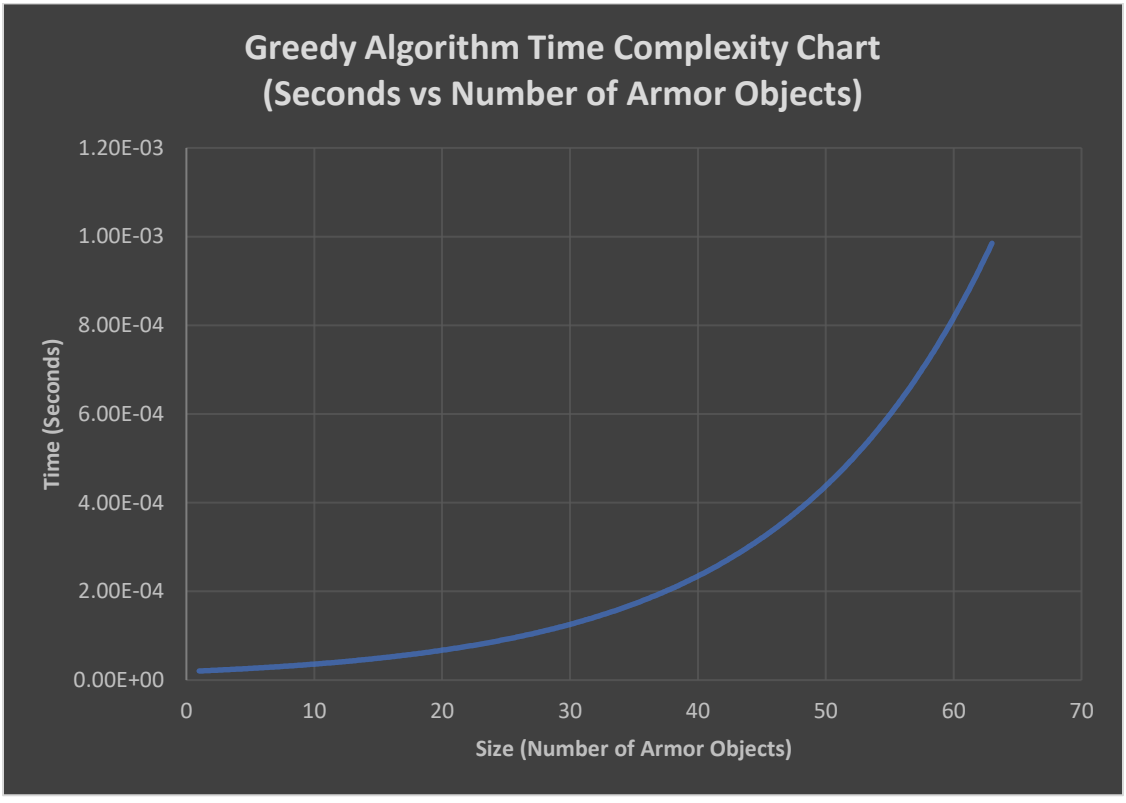
a) Mathematical Analysis

```
ArmorVector sortArmor(const ArmorVector armors) {


        ArmorVector temp_armor = armors;   // 1 t.u.
        bool was_switched = false;   // 1 t.u

        for (int i = 0; i < temp_armor.size(); i++) {    // (n-1)-0+1 = n
             for (int j = 0; j < temp_armor.size()-1; j++) {     // (n-1-1)-0+1 = n-1

               float armorcost1 = temp_armor.at(j)->defense() / temp_armor.at(j)->cost();
                      // 2 t.u.
               float armorcost2 = temp_armor.at(j + 1)->defense() / temp_armor.at(j + 1)-
               >cost();   // 4 t.u. counting the addition to look ahead by 1
                      if (armorcost1 < armorcost2) {   // 1 t.u.

                             std::shared_ptr<ArmorItem> temp = temp_armor.at(j); // 1 t.u.
                             temp_armor.at(j) = temp_armor.at(j + 1);  // 2 t.u.
                             temp_armor.at(j + 1) = temp;              // 2 t.u.

                             was_switched = true;  // 1 t.u.

                      }
               }

             if (was_switched == false) {  // 1 t.u.
                      break;  // array is sorted.    // 1 t.u
               }

        }


        return temp_armor;   // 1 t.u

}


// 2 + (n)((n-1)(2+4+max(6,0)+1)+2) + 1 = 3 + (n)((n-1)(13)+2) = 3 + n (13n-13+2)  =  3 + 13n^2 – 11n

//13n^2 -11n + 3

// O(n^2)
```

```
def   greedy_max_defense (armors, total_cost):


ArmorVector result  =  {}   # set up a new empty armor vector to be returned   // 1 t.u.
ArmorVector temp  = armors  #  temporary armor vector            // 1 t.u.
temp =  sortArmor(temp)       #  calls sortArmor function which is   O(n^2) complexity + 1 t.u.
int size = temp.size()      // 1 t.u.
for i = 0 to n-1 do:      // n times
          if(temp.at(i)'s cost   <=    total_cost)       // 1 t.u.
                    push temp.at(i) into result        //  1 t.u.
                    total_cost -= temp.at(i) cost       // 1 t.u
return result   // 1 t.u.
```

3n + 4 + n^2   = O(n^2) total time complexity

b) Scatterplot Graph



Exhaustive Optimization Algorithm

a) Mathematical Analysis

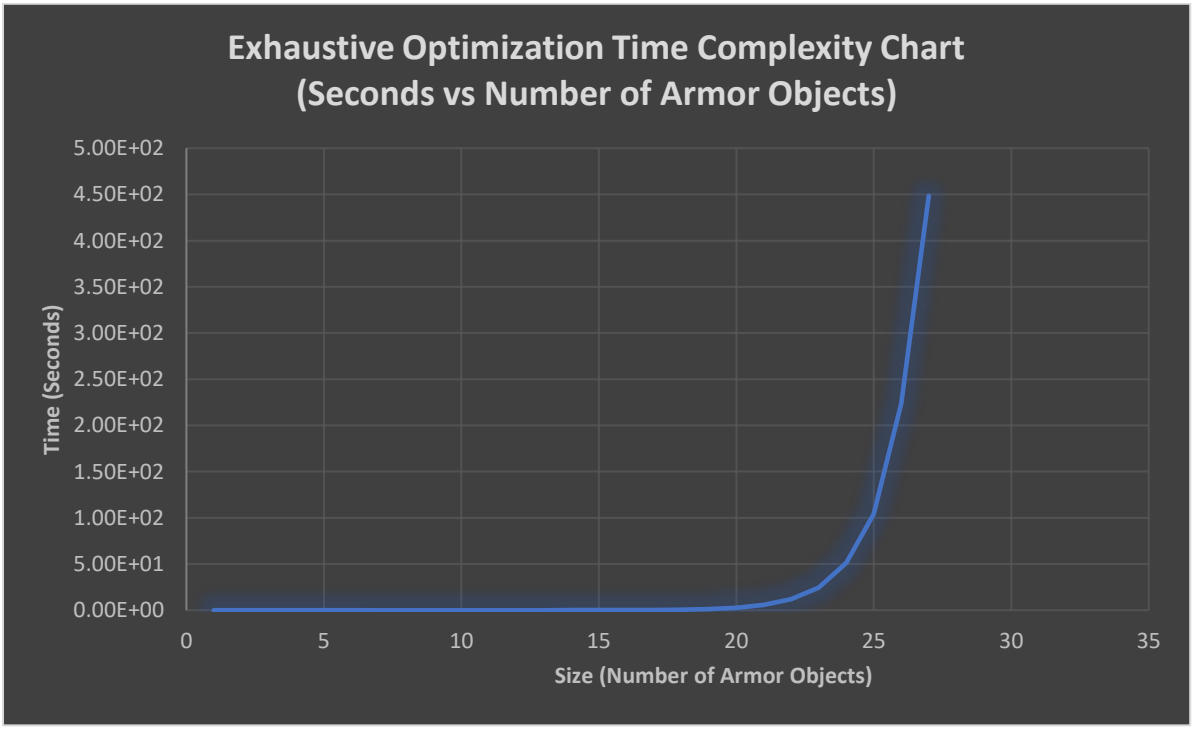
def Exhaustive(armor, total_cost):

```
const int n = armors.size();  // 1 t.u.


int counter = 0;   // 1 t.u.

ArmorVector best;

for int i = 0 to 2^n -1 do:        // 2^n times
       ArmorVector candidate;


       for int j = 0 to n-1 do:    // n times

              if (((i >> j)) & 1  == 1):   // 3 t.u.
                     candidate.push_back(armors.at(j));




       double candidate_cost = 0;         // 1 t.u.
       double candidate_defense = 0;      // 1 t.u
       sum_armor_vector(candidate, candidate_cost, candidate_defense);  // n times

       double best_cost = 0;              // 1 t.u.
       double best_defense = 0;           // 1 t.u.

       if(!best.empty()):   // 1 t.u
              sum_armor_vector(best, best_cost, best_defense);// O(n) times

       if (candidate_cost <= total_cost) {       // 1 t.u.
              if (candidate_defense > best_defense) {     // 1 t.u.
                           best = candidate;              // 1 t.u.


              }

       }


}

std::unique_ptr<ArmorVector> best_subset(new ArmorVector);


for (int i = 0; i < best.size(); i++) {    //  n - 1 - 0 + 1 = n

       best_subset->emplace_back(best.at(i));      // O(1)

}
```

```
        // TODO: implement this function, then delete the return statement below
        return best_subset;  // 1 t.u.
```

total step count = 2+n+2^n(5n)+8(2^n) → O(n(2^n))

b) Scatterplot Graph



Questions

a) Comparing the exhaustive and greedy algorithms I would say that there is a vast difference in terms of speed. Whereas the greedy algorithm runs at a polyrhythmic speed the exhaustive optimization search takes exponential time. For example, at 27 different armor items the greedy algorithm takes about 0.000109 seconds to run. The exhaustive algorithm, however, takes about 448 seconds. Which equates to about a 4 million times increase in run time compared to the greedy method. Albeit, at lower number of armors this difference will diminish. Overall, while I'm not surprised that the exhaustive algorithm turns out to be slower than the greedy algorithm, I was surprised at how slow the exhaustive search became after it reached the 22-size mark.

b) Yes, I do believe that the empirical results turns out to be similar to the mathematical analysis. For one the greedy algorithms mathematically runs at a polynomial time complexity which in this is O(n^2). The scatter plots illustrates this perfectly as the trend line resembles a polyrhythmic function. The curve also isn't steep and wide like I would expect a n^2 polynomial would be. Instead each points are relatively

close to each other even when it reaches up to 64 items. The exhaustive search however is clearer on the comparison between the mathematical and the empirical results. For example, the data points on the exhaustive graph increase by about double between each point which is what you would expect from an exponential function with 2 as the base number (2^n). Not only that but the curve is very steep showing a clear rapid rise as the number of armor objects passes the 20 mark. As the times between points become significantly greater. Overall, the scatter points do a great job of showing the time complexity of the functions and is definitely consistent.

c) I think this experiment does show that exhaustive search algorithms are feasible to implement and produce correct outputs. As the exhaustive search algorithm was able to produce correct if not better results than the greedy algorithm albeit at significantly slower times. This due to the fact that the exhaustive search considered all possibilities whereas the greedy algorithm sorted all the armors into a list of the most cost-effective armors and added them one by one until it spent all of the available funds. The reason why the exhaustive search may be better in some cases is because maybe buying the most cost-effective armor isn't as good as buying a not as cost-effective armor with a high defensive rating. So, the exhaustive search will always produce the best answer, and the greedy will sometimes produce a sub-optimal solution. While this experiment does prove that exhaustive searches are feasible it doesn't mean that it's the best choice in all cases. As you can end up having exponential or even factorial time complexity depending on the case you need to search through. For example, in the armor case if you needed to search over 100 different armor it would take months if not years to find the best set. Therefore, it's important to consider what type of case you need to search through, and how many objects you're dealing with in determining if you should use exhaustive search.

d) The second hypothesis is definitely true. In the experiment when I was gathering empirical data it took a little over seven minutes to perform an exhaustive search on 27 different items. Which is impractical as in the real-world people need to get results quickly or risk losing money. If GPS systems in cars took exponential time to find the best possible path you would likely see that just finding the best would be longer than taking a random path to your destination. Likewise, in the armor case if you had to search through hundreds of different armors you probably would be dead or have switched to another game by the time you figured out the best armor configuration for your character. So, the actual real-world applicability of function with exponential time complexity is very small if not non-existent. As you would only be able to effectively use the function when given a very small number of objects. Which isn't common in real world applications that has to deal with a tremendous number of objects like Facebook. Which if you consider every profile to be a user object has over 2 billion active users.