Project 2: Greedy versus Exhaustive (50 points)

CPSC 335 - Algorithm Engineering

Fall 2020

Instructor: Doina Bein ([dbein@fullerton.edu](mailto:dbein@fullerton.edu)), Mike Peralta ([miperalta@fullerton.edu](mailto:miperalta@fullerton.edu))

# Abstract

In this project you will implement and compare two algorithms that solve the same problem. For this problem, you will design two separate algorithms, describe the algorithms using clear pseudocode, analyze them mathematically, implement your algorithms in C++, measure their performance in running time, compare your experimental results with the efficiency class of your algorithms, and draw conclusions. The first is a greedy algorithm with a fast (i.e. polynomial) running time, while the second is an exhaustive search algorithm with a slow (i.e. exponential) running time.


Conquistador Armor Image

Licensed under the Creative Commons Attribution-Share Alike 4.0 International license.
https://creativecommons.org/licenses/by-sa/4.0/deed.en

Author: Asele S. (www.todocolección.net)
Sources:
 https://www.pinterest.co.uk/pin/461689399287498749/?lp=true
 https://www.todocoleccion.net/militaria-cascos/vii-armadura-casco-yelmo-conquistador-espanol~x17081000#relacionados
 https://commons.wikimedia.org/wiki/File:Conquistador_Armor.jpg



Both algorithms solve the problem of maximizing the defensive strength of a character in your favorite video game. More specifically, given a set of many different armor items available to purchase, these algorithms pick a subset of armor items that fit within a given budget while maximizing the defensive strength attained.

*Conquistador Armor Image: Licensed under the Creative Commons Attribution-Share Alike 4.0 International license (*[*https://creativecommons.org/licenses/by-sa/4.0/deed.en*](https://creativecommons.org/licenses/by-sa/4.0/deed.en)*); Author: Asele S. (*[*www.todocolección.net*](about:blank)*); Sources:* [*https://www.pinterest.co.uk/pin/461689399287498749/?lp=true*](https://www.pinterest.co.uk/pin/461689399287498749/?lp=true)*;* [*https://www.todocoleccion.net/militaria-cascos/vii-armadura-casco-yelmo-conquistador-espanol~x17081000#relacionados*](https://www.todocoleccion.net/militaria-cascos/vii-armadura-casco-yelmo-conquistador-espanol~x17081000#relacionados)*;* [*https://commons.wikimedia.org/wiki/File:Conquistador\_Armor.jpg*](https://commons.wikimedia.org/wiki/File:Conquistador_Armor.jpg)

# The Hypothesis

This experiment will test the following 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.

# The Problem

Both algorithms will be used to solve an interesting problem. Suppose the following. You have started to play an online game which features customizable players. In this game, you start with some amount of “gold” which you may use to purchase armor items to protect your virtual player. First, you have allocated some budget of gold to acquire “armor” items, to increase the defensive strength of your player. There are many different types of armor items available to purchase from many different virtual shops and other players, with varying costs and benefits.



For the sake of simplicity, we shall assume each armor item is “stackable”; you can purchase+wear multiple armor types and receive additional benefit. For example, if you purchase two different helmets of defensive strength +2 and +3, you would enjoy a total defensive strength of +5 (as opposed to the more common experience where only one helmet may be worn at a time).

From the many virtual shops, you have a large list of various types of armor to purchase, and you have a budget, so you cannot buy everything you want. There are two algorithms to help you pick a subset of the available armor items, so that you can maximize your player’s defensive strength while staying within budget.

A description of the problem at hand is as follows:

|  |
| --- |
| **Budgeted defense-maximization problem** |
| *input:* A positive “gold” budget (integer number of coins); and a vector of *n* “armor” objects, containing one or more armor objects where each armor object has an integer cost of gold and defensive strength  *output:* A vector of armor objects drawn from , such that the sum of costs of the armor items from ***K*** is within the prescribed gold budget and the sum of the defensive strengths is maximized. In other words:  and the sum of all armor defense values is maximized |

Note that we require each armor’s gold cost to be positive; we are omitting zero-cost armor items that games sometimes contain.

# The Algorithms

You must implement the following two algorithms for the *Budgeted defense-maximization problem* problem.

The first algorithm uses the greedy pattern. The greedy heuristic is to always choose the “best” (highest defense-per-cost) armor item that fits within the gold budget, and keep selecting the best armor items until you do not have enough gold to purchase any:

greedy\_max\_defense(G, armor\_items):

todo = armor\_items

result = empty vector

result\_cost = 0

while todo is not empty:

Find the armor item “a” in todo of maximum defensiveness per its cost.

Remove “a” from todo.

Let g be a’s cost in gold.

if (result\_cost + g) <= G:

result.add\_back(a)

result\_cost += g

return result

The time complexity of the greedy algorithm depends on the data structures that are used to implement it. A naive approach using unsorted vectors and sequential search to find “a” takes time. This is acceptable but not ideal. Using a heap, binary search tree, or sorting algorithm in a fairly straightforward way can speed this up to

The second uses a proper exhaustive search.

exhaustive\_max\_defense(G, armor\_items):

best = None

for candidate in subsets(armor\_items):

if total\_gold\_cost(candidate) <= G:

if best is None or

total\_defense(candidate) > total\_defense(best):

best = candidate

return best

As discussed in section 7.5.4 of ADITA, subsets(armor\_items) can be implemented using bitwise operations.

exhaustive\_max\_defense(G, armor\_items):

n = |armor\_items|

best = None

for bits from 0 to (2n -1):

candidate = empty vector

for j from 0 to n-1:

if ((bits >> j) & 1) == 1:

candidate.add\_back(armor\_items[j])

if total\_gold\_cost(candidate) <= G:

if best is None or

total\_defense(candidate) > total\_defense(best):

best = candidate

return best

For this to work, the bits loop counter variable needs to be able to store the quantity A good way of ensuring that is to use the largest integer data type in C++, which is the [uint64\_t type](http://en.cppreference.com/w/cpp/types/integer) that is 64 bits wide. This creates a limitation that the exhaustive search algorithm can only handle This is unlikely to be a practical problem, because the time complexity of this algorithm is

Our theory predicts that the exhaustive search algorithm will be far slower than the greedy algorithm with its or time complexity. Your experiment will show whether this is the case.

# Implementation

You are provided with the following files.

1. armor.csv contains over 8,000 armor items.
2. maxdefense.hh is a C++ header that defines skeleton functions for the two algorithms described above. In addition, there is a skeleton function filter\_armor\_vector that filters down a large vector of armor items into a smaller and more manageable set suitable for the exhaustive search algorithm. You are responsible for implementing these three functions (filter\_armor\_vector and the two algorithms). filter\_armor\_vector is intended to be a warmup to get you familiar with the ArmorItem and ArmorVector data types used throughout maxdefense.hh. There are also some provided pre-written functions, for example to load and print an ArmorVector.
3. maxdefense\_test.cc is a C++ program with a main() function that performs unit tests on the functions defined in maxdefense.hh to see whether they work, prints out the outcome, and calculates a score for the code. You can run this program to see whether your algorithm implementations are working correctly.
4. maxdefense\_main.cc is an empty file which also contains a main() function. You may use it together with the timer inside timer.hh, filter\_armor\_vector, and other helper functions to run your algorithm implementations against various lengths of the armor database, in order to gather empirical data for your report. You may run it by executing make, then executing the “experiment” executable that is built.
5. rubrictest.hh is the unit test library used for the test program; you can ignore this file.
6. timer.hh contains a small Timer class that implements a precise timer using the std::chrono library in C++17.
7. README.md contains a brief description of the project, and a place to write the names and CSUF email addresses of the group members. You need to modify this file to identify your group members.

# Obtaining and Submitting Code

This document explains how to obtain and submit your work:

[GitHub Education Instructions](https://docs.google.com/document/d/1UrmpTBOqI3-A6sfvrppEjsaYG1f2-Oa4gLYffiigcyY/edit?usp=sharing)

Here are the invitation links for the projects:

* Doina Bein’s Class; Section 01: <https://classroom.github.com/g/DgFuTwfd>
* Mike Peralta’s Class; Section 04: <https://classroom.github.com/g/s_piXZKY>

(*caution*: Make sure you use the correct link for your section!)

# What to Do

First, add your group member names to README.md. Implement all the skeleton functions in the provided header file. Use the test program to check whether your code works.

When you go to commit files in git, make sure you are not checking any generated executable/object files into git. Use the *.gitignore* file to accomplish this.

Once you are confident that your algorithm implementations are correct, do the following:

1. Analyze your greedy algorithm code mathematically to determine its big-O efficiency class, probably or
2. Analyze your exhaustive optimization algorithm code mathematically to determine its big-O efficiency class, probably .
3. Gather empirical timing data by running your implementations for various values of
4. Draw a scatter plot for each algorithm and fit line for your timing data. The instance size should be on the horizontal axis and elapsed time should be on the vertical axis. Each plot should have a title; and each axis should have a label and units of measure.
5. Conclude whether or not your empirically-observed time efficiency data is consistent, or inconsistent, with your mathematically-derived big- efficiency class for each algorithm.

Finally, produce a brief written project report ***in PDF format***. Your report should be submitted as a checked-in file in GitHub. Your report should include the following:

1. Your names, CSUF-supplied email address(es), and an indication that the submission is for project 2.
2. Two scatter plots meeting the requirements stated above.
3. Answers to the following questions, using complete sentences.
   1. Is there a noticeable difference in the performance of the two algorithms? Which is faster, and by how much? Does this surprise you?
   2. Are your empirical analyses consistent with your mathematical analyses? Justify your answer.
   3. Is this evidence consistent or inconsistent with hypothesis 1? Justify your answer.
   4. Is this evidence consistent or inconsistent with hypothesis 2? Justify your answer.

# Grading Rubric

Your grade will consist of three parts: *Form,* *Function,* and *Analysis.*

*Function* refers to whether your code works properly as defined by the test program. We will use the score reported by the test program as your Function grade.

*Form* refers to the design, organization, and presentation of your code. A grader will read your code and evaluate these aspects of your submission.

*Analysis* refers to the correctness of your mathematical and empirical analyses, scatter plots, question answers, and the presentation of your report document.

The grading rubric is below.

1. Function - 16 points:
   1. load\_armor\_database passes all tests: 2 points (this function is given)
   2. filter\_armor\_vector passes all tests: 2 points
   3. greedy\_max\_defense trivial tests: 2 points
   4. greedy\_max\_defense passes all additional tests: 4 points
   5. exhaustive\_max\_defense trivial tests: 2 points
   6. exhaustive\_max\_defense passes all additional tests: 4 points
2. Form - 12 points
   1. README.md complete: 3 points
   2. Style (whitespace, variable names, comments, etc.): 3 points
   3. Design (where appropriate, uses encapsulation, helper functions, data structures, etc.): 3 points
   4. Craftsmanship (no memory leaks, gross inefficiency, taboo coding practices, etc.): 3 points
3. Analysis - 22 points
   1. Report document presentation: 6 points
   2. Two scatter plots: 6 points each plot
   3. Question answers: 4 points

# Deadline

The project deadline is Friday, October 16th, 11:59 pm.

You will be graded based on what you have pushed to GitHub as of the deadline. Commits made after the deadline will not be considered. Late submissions will not be accepted.