Hacettepe University - Computer Engineering

BBM204 Software Practicum II - Spring 2022

PROGRAMMING ASSIGNMENT 2

OPERATION STRONGHOLD

They will come to destroy us.
We will fight back.

CLASSIFIED CLASSIFIED TOP SECRET CLASSIFIED CLASSIFIED CLASSIFIED

Topics: Hashing, Dynamic Programming, Greedy Programming

Course Instructors: Assoc. Prof. Dr. Erkut Erdem, Prof. Dr. Suat Özdemir, Asst. Prof. Dr. Adnan Özsoy
TAs: Alperen Çakın, Selma Dilek
Programming Language: Java 1.8.0
Due Date: **Wednesday, 30.03.2022 (23:59:59)**

Operation Stronghold

Despite all hopes of the people around the world, who have been exhausted by years of pandemics and everything bad it brought along, the year of 2022 is nothing near promising, happy, prosperous, or at least peaceful. As if a dangerous mutating virus was not enough of a problem, new wars broke out too, caused and commanded by power-greedy world leaders. The world is at a brink of disaster, maybe the biggest one a humankind has ever faced.

The lunatic billionaire [redacted] [redacted], whose wife and daughter had been killed in a terrible road rage-caused traffic accident, lost all hope in humanity after the maniac killer driver was released on the grounds of his fragile mental state. His defense claimed he had been psychologically harmed beyond repair during the pandemics and all subsequent worldwide events that caused detachment from reality through fear and hatred, so much so that he was not able to think rationally anymore, and therefore, was not fit to stand trial. That was the final straw for [redacted]. In his mind, there was no hope left for humanity, it simply had to cease to exist. Being one of the wealthiest people in the world, he assembled a private army of android soldiers, equipped with the most technologically advanced weapons and one mission only: kill everyone. The lethal attacks will be preceded by cyberattacks to cripple the countries' defensive strategies and ensure swift destruction.

We believe that all hope is not lost for humankind, and that it is our duty to protect innocent people. We will have to defend ourselves against the enemy attacks through a combination of defensive techniques including efficient cyberattack detection algorithms, and automated defensive strategies using heavy artillery. The brave and smart students of Hacettepe Computer Engineering Department have been tasked with this crucial operation with three missions.

The fate of the country is now in your capable hands.

1 Mission Firewall

In this part, you will practice hashing.

Hashing - Useful Background

Hashing is a technique in computer programming that transforms given data into a usually shorter and fixed-length string that makes the data easier to find and store, using a hash function. A hash function generates new values according to a mathematical hashing algorithm, known as a hash value or simply a hash. To prevent the conversion of hash back into the original key, a good hash always uses a one-way hashing algorithm. Hashing is used in multiple fields of computer programming such as cryptography, cybersecurity, and data storage. In this mission, you will use a custom hash function to perform a simplified version of malware verification. [3]

As the first sign of the inbound offense, multiple lines of messages have been captured for analysis, sent from IP addresses that belong to the enemy. Even though they seem like innocent messages, some of them are very special in a way; they are designed to trigger backdoors within commonly used software in our computers: Operating Systems, Word Processors, Web Browsers, even the BIOS software. When the TCP packets with those trigger-messages hit our network interfaces, those specific programs will know when it's time to strike and perform their vile duties. Your job is to analyze those captured messages and compare their hashes to an already existing malware database, detected and recorded by our brave and hardworking allies.

1.1 Reading the Malware Database

The malware database is stored as an *.xml file* whose name is specified in the first command-line argument. Your program should parse the malware database into a dictionary for easily checking the entries, as the first step. The dictionary should store malware hashes as keys and malware objects as values.

1.2 Reading Captured Messages and Identification

The captured messages are stored as a *.txt file* whose name is specified in the second command-line argument. Each line in this file represents a received message. Your program should read this file line by line and calculate the hash of each message using the hash function explained in the next section. After calculating the hash of a message, your program should try to locate it in the malware database. If the message is in the malware database then the message should be identified as malware.

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
    <row>
        <title>Trojan.W97M.CVE202140444.A</title>
        <hash>86c8500002aad</hash>
        <level>1</level>
    </row
    ...
    <row>
        <title>Ransom.Win32.LOCKBIT.YEBGW</title>
        <hash>8674b00002a6b</hash>
        <level>1</level>
    </row>
</root>
```

Figure 1: Mission Firewall: A Sample from the Malware Database

```
I used to be an adventurer like you, until I took an arrow to the knee.
Science isn't about why! It's about why not!
It's dangerous to go alone, take this!
```

Figure 2: Mission Firewall: A Sample from the Captured Messages

1.3 Hash Function

Your program should implement and use the hash function to calculate the hash of each message, demonstrated in the pseudocode given below, based on [4]:

Algorithm 1 Turbo-64 Hash Algorithm

```
procedure TURBO-64(input_str)
    MOD_TURBO ← 4294967291
    a ← 0
    b ← 1
    for c in input_str do
        a ← (a + int(c)) % MOD_TURBO
        b ← (a + b) % MOD_TURBO
    end for
    return hex((b << 32) | a)
end procedure
```

1.4 Output

Your program must have two types of output during the identification process explained in section 1.2; a scan log written to a new file named *scanLog.txt* and console output. The scan log should have the following format:

```
<hash of the identified malware>@<line number>
i.e.,
21d6f000013d7@2
14216300004470@14
...
```

Figure 3: Mission Firewall: Scan Log Format

The beginning of the scan log illustrated in the above figure indicates that the captured messages in lines 2 and 14 are identified as malware with hash values "21d6f000013d7" and "14216300004470" respectively.

Your program should generate a console output of the following format that informs us about additional details of the identified malware (**note that your output format must match the given format to pass the output test**):

```
##MISSION FIREWALL INITIATED##
Started scanning...
--------------------------------------------------------------------------------
Detected malware!
Name: Zombifier.Win32.AMDMB2022.C
Threat Level: 5
Hash:21d6f000013d7
--------------------------------------------------------------------------------
Detected malware!
Name: BatteringRam.Win64.CPUKILLER.YXBDN
Threat Level: 8
Hash:14216300004470
--------------------------------------------------------------------------------
Scan complete! 5 threats are found and eliminated. Generating log file...
##MISSION FIREWALL COMPLETED##
```

Figure 4: Mission Firewall: Sample Console Output

2 Mission Nuke'm

In this part, you will practice dynamic programming.

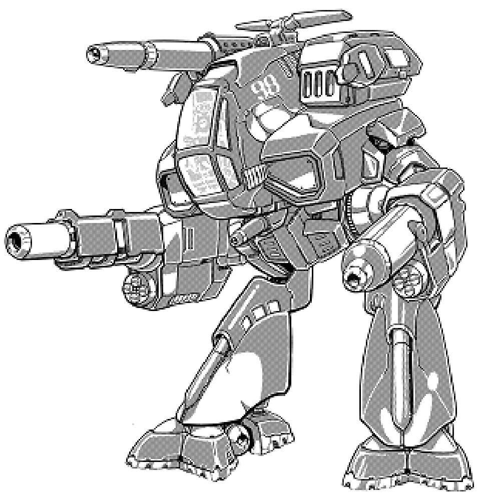
"Those who cannot remember the past are condemned to repeat it."

Dynamic Programming - Useful Background

Dynamic Programming is a technique in computer programming that helps to efficiently solve a class of problems that have overlapping subproblems and optimal substructure property. The majority of the Dynamic Programming problems can be categorized into optimization and combinatorial problems. The optimization problems expect you to select a feasible solution, so that the value of the required function is minimized or maximized. In this mission, you will apply a dynamic programming approach to an optimization problem. The two main approaches to dynamic programming are memoization (the top-down approach) and tabulation (the bottom-up approach). You will employ a dynamic programming approach on the problem at hand. To do that, you will be given a pseudocode of a problem solution, in which you will have to make sure you cache the solutions to the sub-problems and use them to find the optimal solution overall.

When the enemies understand that their cyberattacks have failed miserably, they will send a number of troops of their most vicious combatants. Each of their troops will be deployed over the course of N hours such that there will be exactly one hour between each troop's arrival. We denote the arrival function as $E(t)$, where $E(t)$ is the number of enemy soldiers arriving at hour t. Thanks to your superior hacking skills, while fending off the cyberattacks, you will have hacked into their system and captured their troop deployment schedule stored in **troopsDeploymentSchedule.dat** as an array of integer numbers separated by a space (e.g.: 3 10 10 6 6). This file will be given as the third command line argument. An example how to interpret such a schedule is given below.

```
                        +-----------------------------+
        HourOfAttack (t) |  1  |  2  |  3  |  4  |  5  |
                        +-----------------------------+
#OfEnemiesArriving E(t) |  3  | 10  | 10  |  6  |  6  |
                        +-----------------------------+
```



The given example schedule can be interpreted as follows: one hour into the attack, the first troop with 3 soldiers will arrive. The second troop will arrive one hour later and will consist of 10 soldiers, etc.

The best weapon at our disposal for this kind of an attack is a huge Marauder II BattleMech that carries a massive Blackwell Arms Thunderfist Heavy Gauss Rifle. The gauss rifle can be very powerful and destroy a large number of enemies, but depending on the amount of time it has been allowed to charge. The power $P(i)$ is measured in the amount of enemy soldiers it can destroy with a single shot, and it can be calculated using this formula: $P(i) = i^2$, where i is the number of hours it was allowed to recharge.

```
                           +-----------------------------+
ChargingTimeInHours (i)    |  1  |  2  |  3  |  4  |  5  |
                           +-----------------------------+
#OfEnemiesItCanKill P(i)   |  1  |  4  |  9  | 16  | 25  |
                           +-----------------------------+
```

More specifically, if the rifle is used in the i^{th} hour and j hours have passed since the previous discharge, then it can kill at most $\min[E(i), P(j)]$ enemy soldiers, after which the rifle will be completely drained and will have to recharge for at least one hour before it can be fired again. **Note that only enemies arriving at that specific hour can be killed, any remaining enemies not killed in previous hours will have taken the cover**.

You can assume that the rifle is completely drained at the very beginning (hour 0) of the attack. It can't be used earlier than hour 1 of the attack (exactly when the first enemy troop arrives), and if it is used in the i^{th} hour for the first time, it can kill up to $P(i)$ enemy soldiers.

For the given example enemy deployment schedule, the optimal solution would be to fire the rifle at hours 3 and 5, for a total enemy kill count of 13. If the rifle is activated in the first hour it can only kill one enemy soldier, and if it is activated in the second hour for the first time it can only kill 4 enemies even though 10 enemy soldiers are deployed, because the charging time was only 2 hours ($P(2) = 2^2 = 4$). By activating the rifle in hours 3 and 5, we can obtain a total of $\{\min[E(3) = 10, P(3) = 9] + \min[E(5) = 6, P(5-3) = P(2) = 4]\} = 13$

As you can see, this optimization problem is computationally very complex to use a brute force approach in order to solve it. Any combination of hours at which the rifle is activated can be a potential optimal solution, and if we were to check each possibility, there would be $2^N - 1$ possible rifle usages to inspect for N hours of attack. An example for $N = 3$ would lead into inspection of seven sets of hours for firing: $\{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}$, which would lead to a terrible exponential computational complexity as N grows. You will implement the dynamic programming approach given in Algorithm 2 to solve the problem with a better time complexity.

Algorithm 2 Optimal Defense Solution Dynamic Programming

```
1: procedure GetOptimalDefenseSolutionDP
2:     $SOL(0) \leftarrow 0$
3:     $HOURS(0) \leftarrow []$
4:     for j← 1, . . . , N do
5:         $SOL(j) \leftarrow \max_{0<=i<j}[(SOL(i) + \min[E(j), P(j-i)]]$
6:         $HOURS(j) \leftarrow [HOURS(i), j]$
7:     end for
8:     return $SOL, HOURS$
9: end procedure
```

The dynamic programming approach suggested in the given pseudocode works as follows: At each hour j, check already stored optimal solutions for each hour i from 0 to $j-1$ ($0 \leq i < j$). If the attack ends at hour j, there is no reason not to fire the rifle at hour j as we don't have to save the charge for any future enemy arrivals. Therefore, the optimal solution at any hour j will be the maximum of all solutions over the hour interval ($0 \leq i < j$) given as the recurrence relation $SOL(j) \leftarrow \max_{0<=i<j}[(SOL(i) + \min[E(j), P(j-i)]]$; i.e., the maximum number of soldiers killed at hour i plus the number of soldiers that can be killed at most at hour j with the rifle charged for $(j-i)$ hours, which is $\min[E(j), P(j-i)]$. A visualization of this reasoning is given below for hour 5.

```
Assume we used the given algorithm and obtained the optimal solutions for hours 1,..,4:
SOL(1) =  1, HOURS(1) = [1]
SOL(2) =  4, HOURS(2) = [2]
SOL(3) =  9, HOURS(3) = [3]
SOL(4) = 10, HOURS(4) = [3,4]

Then the optimal solution for hour 5 is the best (max) option among these available ones:
Option 1: Use SOL(1) and shoot at 5 -> SOL(1) + min[E(5)=6, P(5-1)=P(4)=16] =  1 + 3 =  4
Option 2: Use SOL(2) and shoot at 5 -> SOL(2) + min[E(5)=6, P(5-2)=P(3)=9]  =  4 + 6 = 10
Option 3: Use SOL(3) and shoot at 5 -> SOL(3) + min[E(5)=6, P(5-3)=P(2)=4]  =  9 + 4 = 13 <- MAX
Option 4: Use SOL(4) and shoot at 5 -> SOL(4) + min[E(5)=6, P(5-4)=P(1)=1]  = 10 + 1 = 11

Since the option 3 results in the maximum value: SOL(5) = 13, HOURS(5) = [HOURS(3), 5] = [3,5]
```

Mission Steps Summarized

- Given the information about deployment of enemy troops each hour $E(t)$, and the gauss rifle power recharge function $P(i)$, find the maximum number of enemies that can be killed during Mission Nuke'm by firing the rifle at certain hours of the attack.
- Use the provided Dynamic Programming pseudocode and implement your solution by completing the ***getOptimalDefenseSolutionDP*** function in ***DefenseAgainstEnemyTroops.java***. **A brute force solution will not be accepted and will be graded 0.**
- The ***getOptimalDefenseSolutionDP*** function should return two pieces of information: the maximum number of enemies that can be killed, and an array of hours at which rifle should be fired for the calculated maximum damage. This will be achieved by having this method return an instance of ***OptimalEnemyDefenseSolution*** (see ***OptimalEnemyDefenseSolution.java***).

The expected output for the given sample inputs is given below and should be printed to the STDOUT immediately after the output from the Mission Firewall. **Note that your output format must match the given format to pass the output test**.

```
##MISSION NUKE'M INITIATED##
The total number of enemies arrived: 35
Maximum number of enemies killed: 13
Hours at which the weapon should be fired: 3, 5
The number of enemies left alive: 22
##MISSION NUKE'M COMPLETED##
```

3 Mission Exterminate

In this part, you will practice greedy programming.

Greedy Programming - Useful Background

Greedy Programming is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. So the problems where choosing locally optimal step also leads to the global optimal solution are best fit for Greedy Programming. In this last mission, you will employ greedy programming approach to solve another optimization problem.

After Mission Nuke'm, some of the enemy soldiers will have survived. In order to ensure the safety of everyone, each enemy must be exterminated, because those android soldiers will stop at nothing as long as they are operational. They are only following their programming which is to kill all human targets they encounter.

For this mission we have a certain number of Armed Unmanned Aerial Vehicles (AUAVs) at our disposal, which can carry bombs. AUAVs have a specific load capacity of **10 kg each**, whereas bombs differ in weight. Prior to the defensive mission you will be given the updated information about the number of available AUAVs and the respective weights of the bombs that need to be loaded on the AUAVs. This information will be stored in **AUAVBombs.dat** input file (given as the fourth command line argument) such that the first row will store the number of the available AUAVS, while the second row will store the weights of the bombs that are necessary to exterminate all enemies, in a random order and separated by a space. An example input file that informs about 8 available AUAVs and 9 bombs with weights of 2, 5, 4, 7, 1, 3, 8, 10, and 1, respectively, will look as follows:

```
8
2 5 4 7 1 3 8 10 1
```

We are not rich like our enemy. We only have limited resources and flying an AUAV is very expensive. For this reason, we want to minimize the number of AUAVs we should deploy, such that we can load all the bombs. Your task in this mission is to use a greedy approach to find the minimum number of AUAVs on which all bombs can be loaded. You can use an approach based on the one given in Pseudocode 3. **A brute force solution will not be accepted and will be graded 0.**

Algorithm 3 Optimal Final Defense Greedy Programming

```
1: procedure GETMINNUMBEROFAUAVSTODEPLOY(maxNumOfAvailableAUAVs: int, AUAV_CAPACITY: int)
2:     First sort all bombs by their weights in decreasing order    ▷ So that we can load the largest bombs first.
3:     Create an array to store remaining space in all available AUAVs
4:     for i← 1, . . . , numBombs do
5:         Find the first AUAV that can accommodate bomb i ▷ When processing the next bomb, scan the previous AUAVs in order, and load the bomb onto the first AUAV it fits in.
6:         Get a new AUAV only if the bomb does not fit in any of the used AUAVs    ▷ Don't forget to check if there is still at least one available AUAV and if the bomb weight doesn't exceed AUAV capacity.
7:     end for
8:     return minNumberOfAUAVsToDEploy or -1 if all bombs could not be loaded.
9: end procedure
```

The given approach is greedy because it is trying to handle the heaviest bombs first, and is taking the local optimal decision at each step without any regards about weather it will lead to a solution that is not the best one overall. However, we can take our chances as this approach will give us a very fast solution. Otherwise, this decision problem is computationally NP-hard and finding the best solution could take time we don't have.

The greedy approach described above will produce the following solution for the given sample input file (see Fig. 5).

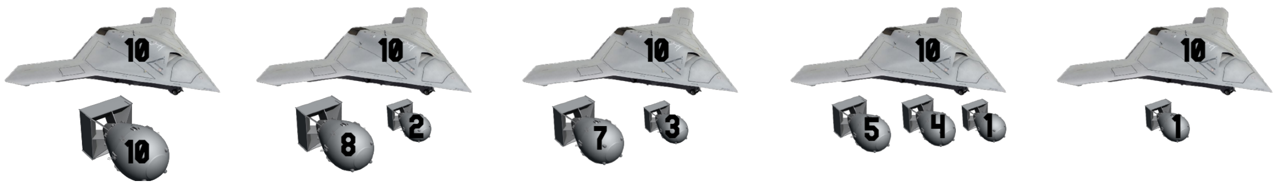


Figure 5: OptimalFinalDefenseGP solution for the given sample input.

Mission Steps Summarized

- Given the information about the number of available AUAVs and the weights of all bombs that need to be loaded, find the minimum number of AUAVs that should be deployed to exterminate the enemy.
- Use the provided Greedy Programming pseudocode and implement your solution by completing the ***getMinNumberOfAUAVsToDeploy*** function in ***OptimalFinalDefenseGP.java***. **A brute force solution will be graded 0!**
- The ***getMinNumberOfAUAVsToDeploy*** function should return an integer: either the minimum number of AUAVs that need to be deployed if all bombs can be loaded successfully, or **-1** if the loading failed for some reason.

For the given sample input, the expected output is given below. The output should be printed to the STDOUT immediately after the output from the Mission Nuke'm. **Note that your output format must match the given format to pass the output test**.

```
##MISSION EXTERMINATE INITIATED##
The minimum number of AUAVs to deploy for complete extermination of the enemy army: 5
##MISSION EXTERMINATE COMPLETED##
```

In case all bombs cannot be loaded on the available AUAVs (not enough AUAVs or the weight of at least one bomb exceeds the AUAV load capacity), the output should be as follows:

```
##MISSION EXTERMINATE INITIATED##
We cannot load all the bombs. We are doomed.
##MISSION EXTERMINATE COMPLETED##
```

MISSION EXTERMINATE COMPLETED

Must-Use Starter Codes

You MUST use this starter code. All classes should be placed directly inside your **zip** archive. Feel free to create other additional classes if necessary, but they should also be directly inside **zip**.

Grading Policy

- Submission: 1%
- Implementation of the algorithms: 90%
  - Hashing tests: 30%
  - Dynamic programming tests: 30%
  - Greedy programming tests: 30%
- Output test: 9%

Important Notes

- **Brute force solutions for dynamic and greedy programming parts will not be accepted and will be graded with 0.**
- Do not miss the deadline: **Wednesday, 30.03.2022 (23:59:59)**.
- Save all your work until the assignment is graded.
- The assignment solution you submit must be your original, individual work. Duplicate or similar assignments are both going to be considered as cheating.
- You can ask your questions via Piazza (`https://piazza.com/hacettepe.edu.tr/spring2022/bbm204`), and you are supposed to be aware of everything discussed on Piazza.
- You can only test your code via `http://159.223.4.213/PA2.html` (**does not count as submission!**).
- You must submit your work via `https://submit.cs.hacettepe.edu.tr/` with the file hierarchy given below:
  - **b<studentID>.zip**
    - Main.java <FILE>
    - XMLParser.java <FILE>
    - Malware.java <FILE>
    - MalwareScanner.java <FILE>
    - Util.java <FILE>
    - OptimalEnemyDefenseSolution.java <FILE>
    - DefenseAgainstEnemyTroops.java <FILE>
    - OptimalFinalDefenseGP.java <FILE>
- The name of the main class that contains the main method should be **Main.java**. **You MUST use** this starter code. The main class and all other classes should be placed directly in your **zip** archive. Feel free to create other additional classes if necessary, but they should also be inside the **zip**.
- This file hierarchy must be zipped before submitted (not .rar, only .zip files are supported).
- **Usage of any external libraries is forbidden.**

Run Configuration

Your code will be compiled and run as follows:

```
javac -cp . Main.java
java -cp . Main <malwareDatabase> <capturedMessages> <enemyDeploymentSchedule> <AUAVSBombsData>
```

Academic Integrity Policy

All work on assignments **must be done individually**. You are encouraged to discuss the given assignments with your classmates, but these discussions should be carried out in an abstract way. That is, discussions related to a particular solution to a specific problem (either in actual code or in pseudocode) **will not be tolerated**. In short, turning in someone else's work (including work available on the internet), in whole or in part, as your own will be considered as **a violation of academic integrity**. Please note that the former condition also holds for the material found on the web as everything on the web has been written by someone else.

The submissions will be subjected to a similarity check. Any submissions that fail the similarity check will not be graded and will be reported to the ethics committee as a case of academic integrity violation, which may result in suspension of the involved students.

References

[1] Jeff Miller, "Ghost Recon: Future Soldier", https://www.artstation.com/artwork/OyLbJ, Last Accessed: 04/03/2022.

[2] Bahadır Çakın, "VIRUS", https://www.artstation.com/artwork/YKb503, Last Accessed: 04/03/2022.

[3] "Hashing Definition", https://searchsqlserver.techtarget.com/definition/hashing, Last Accessed: 04/03/2022.

[4] "Adler-32", https://en.wikipedia.org/wiki/Adler-32, Last Accessed: 04/03/2022.

[5] "Dynamic Programming", https://www.programiz.com/dsa/dynamic-programming, Last Accessed: 05/03/2022.

[6] "Introduction to Dynamic Programming 1", https://www.hackerearth.com/practice/algorithms/dynamic-programming/introduction-to-dynamic-programming-1/tutorial/, Last Accessed: 05/03/2022.

[7] Luis Robaina, "A Systematic Approach to Dynamic Programming", https://betterprogramming.pub/a-systematic-approach-to-dynamic-programming-54902b6b0071, Last Accessed: 05/03/2022.

[8] "Marauder II", https://www.sarna.net/wiki/Marauder_II, Last Accessed: 05/03/2022.

[9] "Greedy Algorithms", https://www.geeksforgeeks.org/greedy-algorithms/, Last Accessed: 05/03/2022.

[10] "X-47 UAV Drone UCAV 3D Model", https://free3d.com/3d-model/x-47-uav-drone-ucav-1741.html, Last Accessed: 05/03/2022.