**COMP338 - Project #1 Due: Saturday, Jan 11, 2025**



**DEPARTMENT OF COMPUTER SCIENCE**

**COMP338 - Artificial Intelligence**

**Course Project 1**

**Prepared by**

**Name:** Malak Mustafa **– Student ID:** 1212986

**Name:** Layal Abed **– Student ID: 1220480**

**Instructor’s Name:** Mohammad Helal

**Section: 2**

**Date: 30/12/2024**

**Background on Genetic Algorithms:**

First step, we generate the passcode randomly using random class, and inside constructor generate best parent randomly and best fitness.

Second step, generate parent1 from best parent and parent2 randomly to do the genetic algorithm.

Third step, we start the genetic algorithm, first do crossover between parent1 and parent2, then do mutate for the crossover child.

Fourth step, calculate fitness and convergence rate for each child.

Fifth step, compare if the best fitness is large or equal child fitness, then the passcode still not found, it will stop when the child equals the passcode that’s mean the best fitness equal 32.

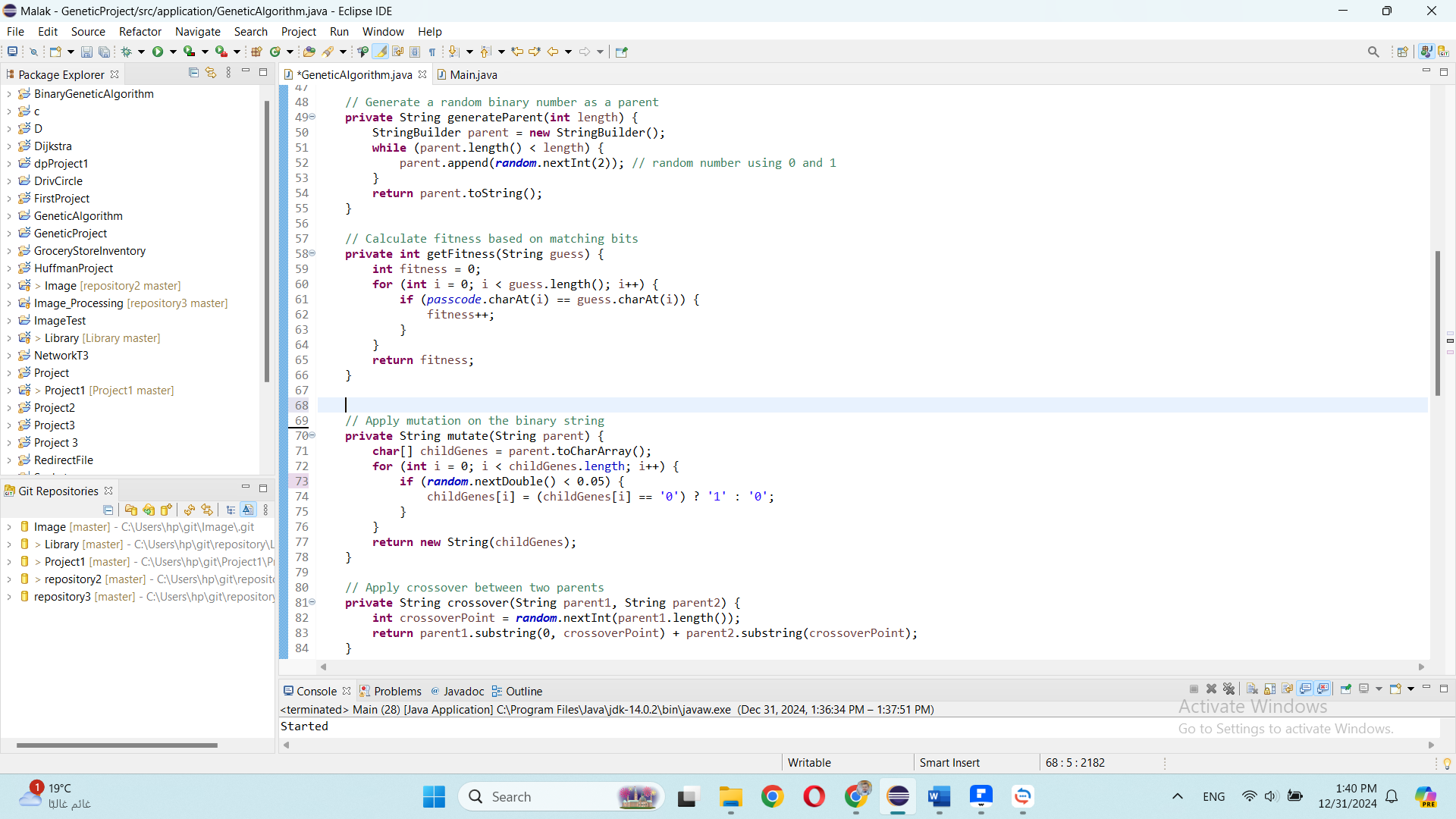
**The problem formulation, what does the chromosome represent, what does the gene represent**

The problem involves discovering a 32-bit binary passcode using genetic algorithms that mimic natural selection. The algorithm starts with a single random chromosome as the best parent, where the chromosome represents a potential solution (a 32-bit binary string), and the gene represents a single unit within the chromosome (a binary bit: 0 or 1) that corresponds to a part of the passcode.

**Represent the solution**

This is the genetic algorithm that successfully came up with a 32-bit passcode through iteratively improving solutions via crossovers and mutations. In this case, convergence to the right passcode shows the effectiveness of the algorithm in studying the solution space. The number of generations required and the convergence rate recorded denote how fast the solution was improved. Crossover and mutation helped to maintain diversity, preventing the algorithm from sticking in local optima. This illustrates the possibility of the algorithm to efficiently solve binary optimization problems.

**Include and explain the code, each stage of the Genetic Algorithm**

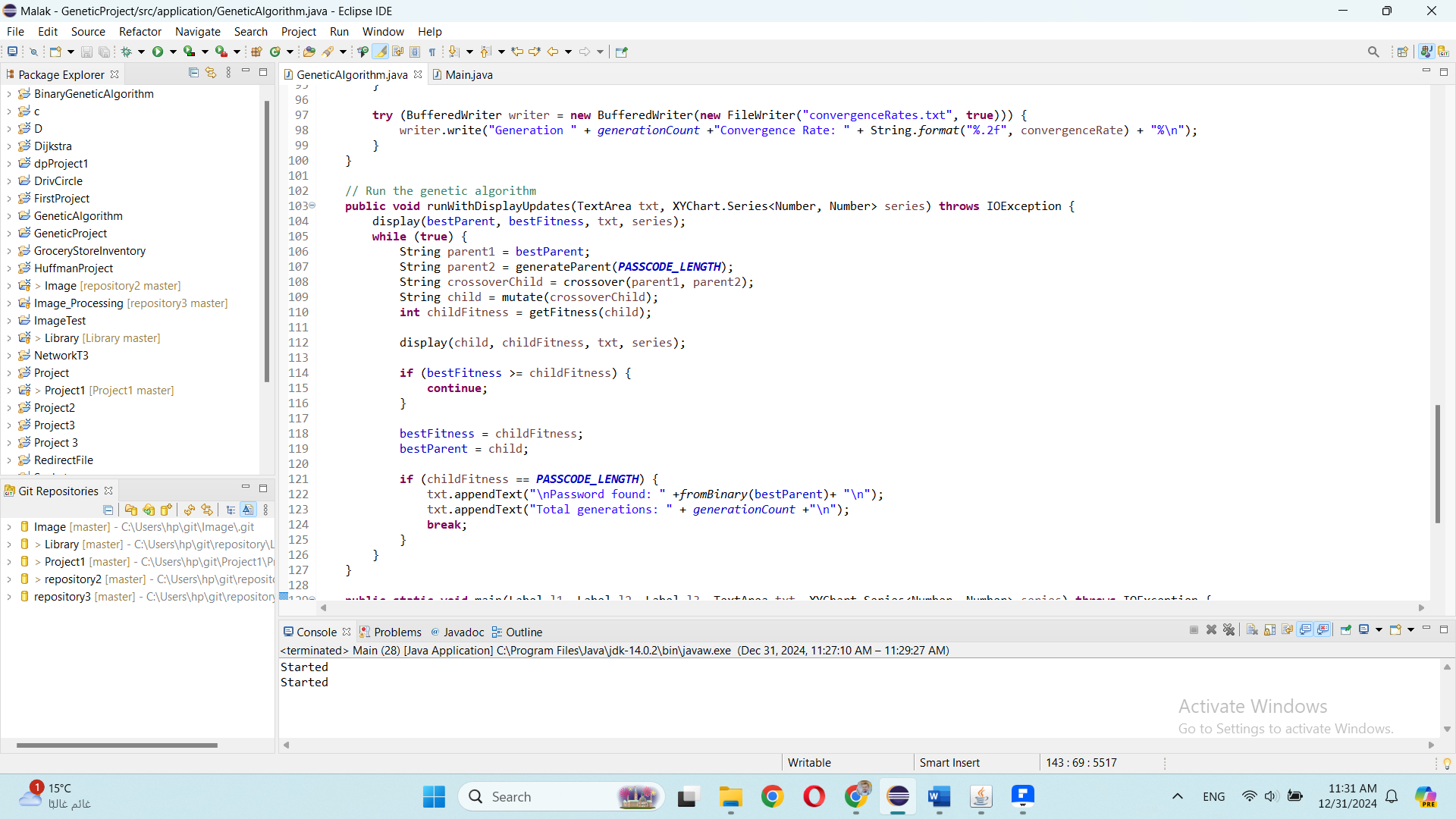
****

The generate parent method takes a parameter length (32) and generates a parent of the type string builder using random numbers 0 and 1, and the method return the generated number.

The get fitness method takes a parameter guess (the string number we need to calculate its fitness), at first the fitness will be zero (mean no matching), then in for loop we take each bit of guess parameter and passcode to check if they are matching or not, if they are matching the fitness will increment one.

The mutate method takes a parameter parent (the number we need to mutate it), we convert the string to array to get index and in the for loop for each character we take random number from 0 to 1 and check if it is less than 0.05 (mutation rate), if it is replace the character in the index if it is a 0, we replaced with 1 if not 0 we replaced to 0 and return the new number as a string.

The crossover method takes a parameter parent 1 and parent 2 (the numbers we need to cross it), our method of determining the number position is based on a random index, then we crossed the parents and return the crossover child (the first bit of it to the index from parent 1 and the other bits from parent 2).



In the with display updates method, we generate two random parents using generate parent method (the best parent variable also generated randomly), then do the crossover using crossover method to give a new child. After that, do a mutate using mutate method in the new child and calculate the fitness using get fitness method. Next check if the fitness equals or larger than the passcode length the algorithm will continue, if not we changed the best fitness and the best parent to the last child, also check if the fitness equals passcode length, if it is displaying the solution.

**Parameter tuning analysis**

We performed an analysis of the mutation rate to evaluate its impact on the convergence rate.

1. Mutation Rate Values:  
   Mutation rates tested: 0.02, 0.05 and 0.1.

 Low **Mutation Rate (0.02)**: Slower convergence, as diversity is insufficient to escape

local optima.

 Moderate **Mutation Rate (0.05)**: Optimal balance between exploration and

exploitation, leading to fast convergence.

 High **Mutation Rate (0.1)**: Slower convergence and unstable performance due to excessive randomness disrupting good solutions.

1. Metric for Analysis:
   * Convergence Rate: The percentage of correct bits in the best candidate solution across generations.
   * Recorded at each generation and plotted for different mutation rates.

**References:**

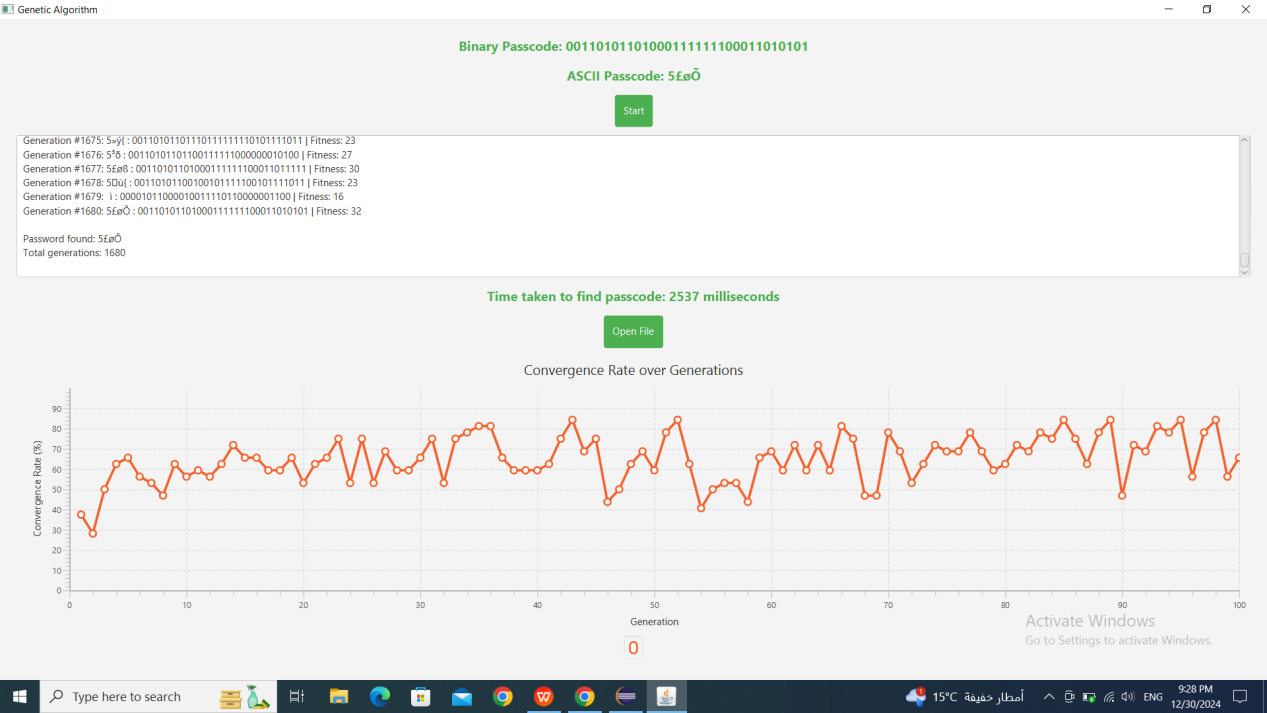
We explored various sources on the internet to understand how this algorithm works, including the material we have at hand and a website on google.

Website google:

<https://gist.github.com/Audhil/c970bb25729206b60c8a10c055f684b0>

Material we have: <https://www.jarrar.info/courses/AI/Jarrar.LectureNotes.Ch4.LocalSearch.pdf>

**Screenshots for the run:**



**Results and discussion:**

The number of generations and the time required to find the passcode using a genetic algorithm are influenced by factors such as population size, mutation and crossover rates, the fitness function, and the inherent randomness of the algorithm. Typically, it may take from a few hundred to several thousand generations to converge, depending on these variables. The time taken can vary from a few seconds to a minute, based on the algorithm's configuration, the complexity of the problem. The algorithm's performance may vary because of the random elements in each run. In our program, it took 2537 milliseconds and resulted in a total of 1680 generations to implement the algorithm on a random passcode “00110101101000111111100011010101”