SAPIENZA
UNIVERSITÀ DI ROMA

DEPARTMENT OF COMPUTER SCIENCE

The Battle of sexes

A Java Simulation implementing Game Theory

PROGRAMMING 2

Professor:
Pietro Cenciarelli

Students:
Pellegrino Lorenzo
Lin Can
Russolillo Simone
Wang Filippo

Academic Year 2021/2022

Contents

1 Introduction

This project is inspired by Chapter 9 of The Selfish Gene, a famous book by Richard Dawkins that builds upon the *Adaptation and Natural Selection* theory as a way of expressing the **gene-centered** view of evolution.
The chapter describes "the battle of the sexes", where a model is provided of a population featuring two male types, the faithful (F) and the philanderers (P), and two female types, the coy (C) and the fast (S).
The latter are the players of dawkins' *Game Theory* based study in which we have a mathematical model of conflict and cooperation between rational decision-makers

a : the evolutionary benefit for having a baby

b : the cost of parenting a baby

c : the cost of courtship

Each type of Individual has different strategy of survival:
– **faithful men:** they are willing to engage in a long courtship and participate in rearing their children;
– **philanderers**: reckless men, they don't waste time in courting women: if not immediately accepted, they move away and try somewhere else; moreover, if accepted, they mate and then leave anyway, ignoring the destiny of their children;
– **coy women**: they accept a partner only after a long courtship;
– **fast women**: if they feel so, they don't mind copulating with whoever they like, even if just met.

	F	P
C	$(a - b/2 - c, a - b/2 - c)$	$(0, 0)$
S	$(a - b/2, a - b/2)$	$(a - b, a)$

Table 1: Evolutionary rules set in the payoff matrix

The goal of our software is to perform non-deterministic simulations that are parametric to the a, b, c values taking into account the behavior of male and female types and returning the percentages of the four categories of people at the

end of the simulation, analyzing the system's equilibrium state. We opted employ multi-threading to create the non-deterministic program, as we'll see..

2 Design

The basic idea behind the whole project is pretty straightforward: we have a class-defined population, composed by single different individuals. Each of them spends its entire life trough a cyclical repetition of two periods of time, that define and articulate our population's individual life and behaviour:

- **Night time**, when people are lively and go out meeting others inside a "nightclub" (who knows what might happen), and
- **Day time**, in which people are inactive and we, the programmers, gather useful informations on all the population's values.

More specifically, during Night time, all females enter the club and wait for a male companion; of course, every one of them has her own tastes in man depending on her type. Then, every male enters the place and finds a girl, they exit and they give birth to a new individual who will share one of their parents' type. During Day time instead, people will rest or raise their kids while we take a census of every type of individual to calculate the overall male-female ratio and plot a graph.

2.1 Towards equilibrium

This process goes on as long as we like but eventually will reach a stall point when it achieves equilibrium between the four types. This happens when each type is at its maximum evolutionary potential and it's indeed a peculiar event: one may think that some of the types are at an evolutionary disadvantage compared to others and are destined to be outlasted, the faithful man wastes time courting the girls and stays with the kids while the philanderer doesn't even bother, the fast may be left alone raising her offspring... this is because we cannot see things on the global level, in fact every tipe has its weaknesses and strenghts, the philanderer depends exclusively on the fast just like the coy relies on the faithful, the fast instead can choose between both males. These subtle and close bonds let us achieve equilibrium in a perfect **Circle of Life** just like in *"The Lion King"*, where the antelope feeds the lion and the latter will become the grass that feeds the former.

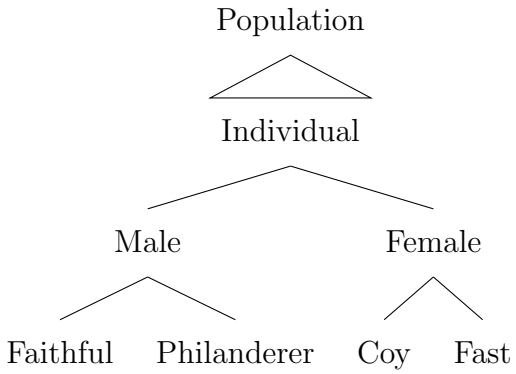
2.2 Usage of multi-threading

One of the most thrilling aspects in our program is that every single individual mantains their own independency and can live simultaneously with others , but how is that? Computer users take it for granted that their systems can do more than one thing at a time, but this is not true, computers use a clever strategy called concurrency to pursue different tasks at the same time, exactly as Java does with threads, a lightweight sub-process that allows a program to work more effectively. It is a small independent unit having a separate execution path. Thanks to their concurrency capabilities, we chose to use a multithreading algorithm to implement our code, each thread is an individual and they interact with each other independently thus causing each run a slightly different result. This is also because threads are non-deterministic, which means that during execution the different small tasks can always return different outputs or can succeed each other in random order.

2.3 Lifetime of a thread

Let's go into more detail, the life of an individual goes as follows: they are born with a fixed amount of initial points, so the population gets incremented, they are inactive until maturity, therefore they enter the club while it's Night time and eventually copulate. Depending on their type the males will enter the nightclub multiple times per night, in order to find a damsel and reproduce with her. The individual's points increase by a different amount of bonus points, that is the evolutionary payoffs (according to the table) of their last mating, this resolves in someone who had a lot of kids having more points than someone else who had less. At the end of every day, however, their points are decremented by a fixed value that represent the cost of life, if the points become negative they die. The result of this choice is that an individual who deals a major evolutionary benefit to its own type by giving more offspring will live longer. Once the desired iterations are done executing and our simulation is over, an anonymous thread class called Bodyguard comes into action, kicking everyone out of the club, while shouting "The club is CLOSED!!!".
An interesting remark worth noticing is that even if we didn't use the day-night analogy, by substituting it with "evolutionary generations", the program would work the same and still make sense: kids are raised and grow up until they are able to mate, then they will partecipate in the creation of new generations yet to come.

3 Implementation



3.1 A new life is born

First and foremost the main thread creates the individuals of the four different types by invoking a static method of the Population class: *createIndividuals*(). This method **sequentially** calls the constructor of the specified type hence instantiating and starting the individual. The constructor is responsible of two operations:
1. initialising the type field of the individual with an integer number from 0 to 3 dependnig on the type (a sort of **"DNA"**)
2. increasing the related index of the **synchronized** int array numberIndividuals. Doing so the exact number of living individuals is known at any given time and with no loss of data.

3.2 Waiting for the party to get started

Since the threads are created one after the other someone could think the type of the older individuals has an advantage on the newborns in the game of life but that is not the case. When coming to synchronize such a huge amount of threads without losing parallelism, we thought of a very effective way of communication between them: a **shared access to boolean flags**. The "night" boolean flag is an attribute of the Time class simulating the alternance of night and day just like in reality. The variable can either be:

1. false → daylight
2. true → night

In the Time class "night" is initialised to false meaning the threads have to wait for the night to come before going to the club. The flag is changed to true only when all the individuals have been created hence they all access the party at the same

time.
Adopting the above method is extremely efficient but can cause **memory consistency errors**. To avoid any we declared the "night" attribute to be **volatile**: threads access the variable directly through main memory and not through cache.

3.3 The night club

The club is the structure where individuals meet and eventually end up in a relationship. It has been implemented by extending the **generic class** $ArrayList <>$ and by choosing the Female class as its parameter. The result is a class of our own called myList adding two more methods:
1. push: intended for females to enetr the club
2. pop: intended for males to interact with the chosen woman
To avoid any $outOfBoundsException$ and $lossofdata$, these two methods are synchronized on the stack so that only one thread at a time can operate on it.

3.4 The monotonous life of individuals

The run method of threads is structured into two while loops.
The outer loop gets the threads to run until the boolean untilEquilibriumReached is changed to false from the main.
On the other hand the inner loop represents the nightlife of the $individuals$: females enter the club (they push themselves into the stack) and wait for a male to interact with them (pop them from the stack and invoke the giveBirth method). Finally each night the *individual* points of a thead are checked and if less then zero an exception is raised (meaning the time has come for that thread to terminate execution) wherease each day a life cost is subtracted from those same points.

3.5 The giveBirth method

This synchronized method is defined in the Female class and **overridden** both by the fast and coy women. It is executed by the males and is given as input the girl who has just been popped by the stack. The **dynamic method dispatch** (meaning methods are linked at runtime) enables the males to know which version of the method (coy or fast) to execute even though the Female super class is given as input (the type isn't specified). This method is responsible of:
1. creating a new thread depending on the parents

2. updating the individual points of the parents with the payoffs dependent on a, b, c
3. let the parents sleep to simulate the cost of courtship and parenting the child
4. waiking up the girl (both the giveBirth method and the Female run method are synchronised on the female).

3.6 No more fun: the Bodyguard

When equilibrium is reached the program must end meaning all individuals must terminate. For this reason the bodyguard has been created.
It is an object belonging to an **anonymous class**. It gets instantiated by invoking the thread constructor taking as input a Runnable. the Runnable interface is a **functional interface** that we have implemented on the fly using a **lambda expression**. The duty of the bodyguard is to wake up all males and to interrupt all females. This leads to the correct termination of every thread.

4 Results and evaluations

The outputs of the program are 2 charts:
- Population Trend shows the growth of each specific time in population, the X axis represents the number of iteration while the Y axis represents the number of people of the specific type of Individual;
- Population Ratio shows the ratio between Faithfull-Philanderer and Coy-Fast, the X axis represents the number of iterations while the Y axis represents the ratio of these 2 pairs.
The sum of the ratio between males is 100,and so is the sum of the ratio between the females. Ratio is calculated throught the formula (number of type/sum of the sex of the type)*100

We consider the population to be stable when the ratio of each type of Individual is growing asymptotically.

The ratio between the numbers of males and females may not be around 50%, this is caused by the quick growing of Philanderers resulting in a rapid decrease of Fast.

Normal Run

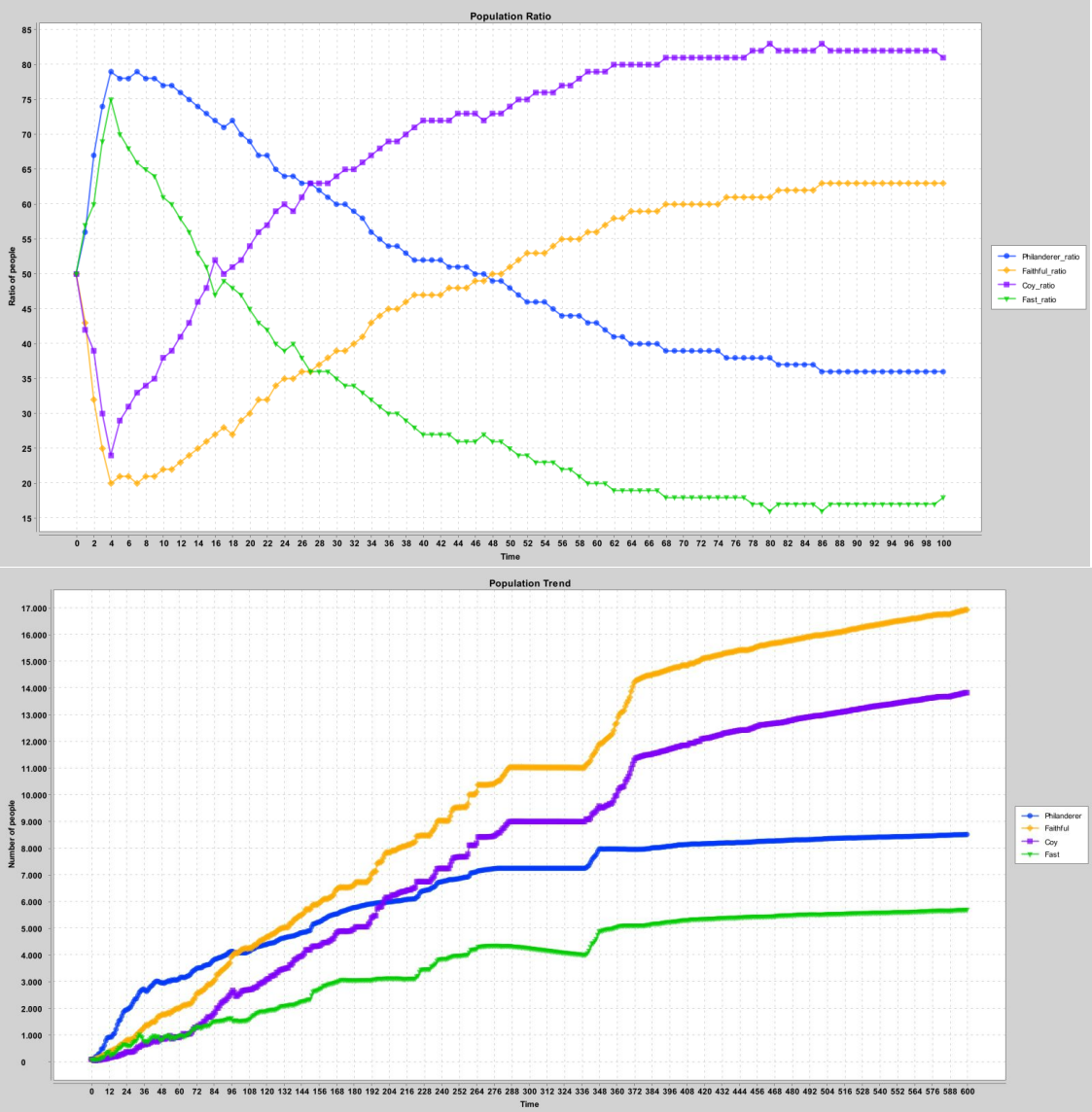


Figure 1: phil 10 - faith 10 - fast 10 - coy 10
a = 15 b = 20 c = 3

This is the most basic run with the same ratio between the different type of people, as we can see from the image at the very start the number of Philanderers grows very fast compared to Coy and Faithful since they don't engage in parenting the child. The success of Philanderers will also be their failure, as they grow the number of Fast will start decreasing since they will struggle in pareting the child. The pair Coy and Faithfull will have an advantage again, growing until reaching the stable situation.

Different run 1

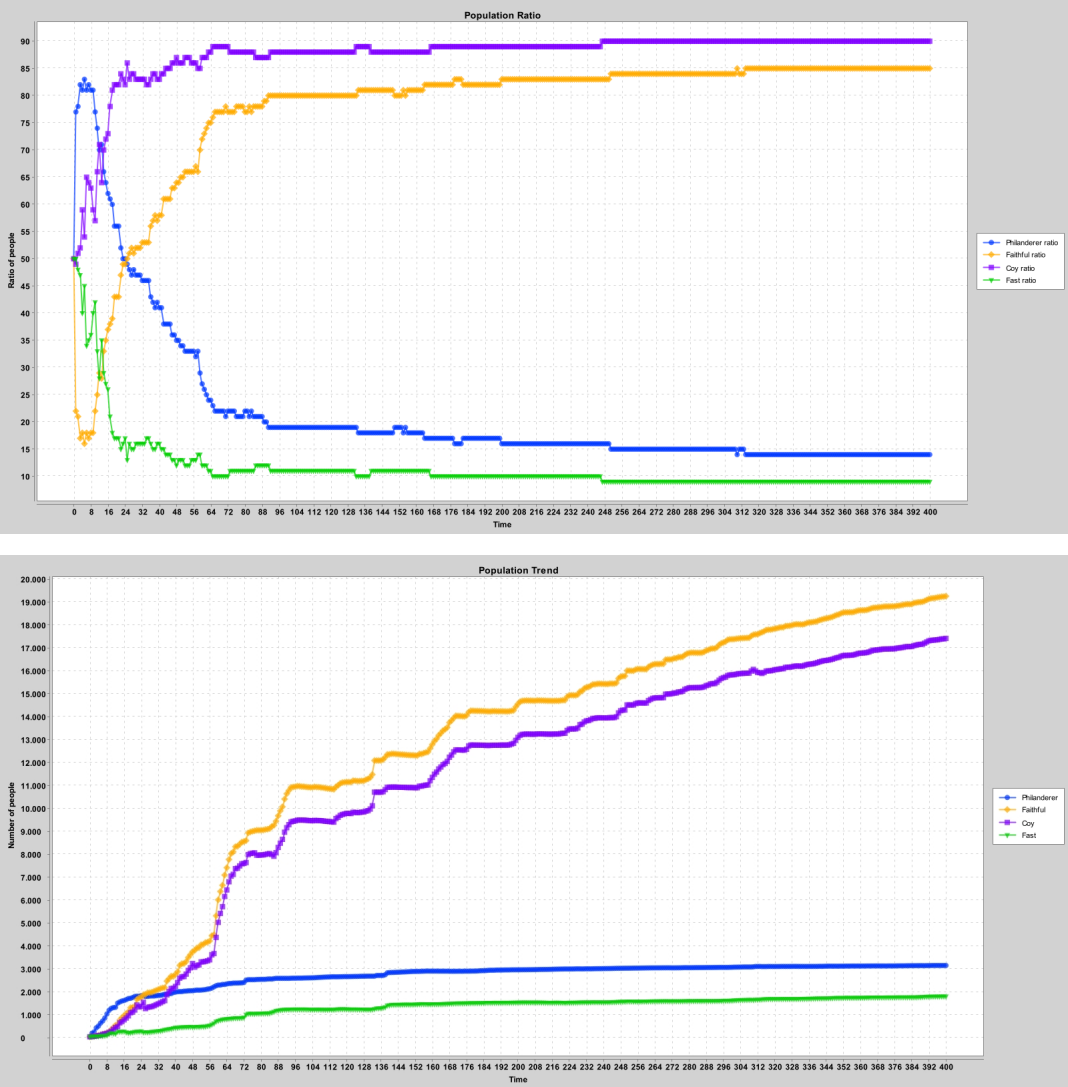


Figure 2: phil 100 - faith 100 - fast 100 - coy 100
a = 15 b = 20 c = 1

This run is very similar to the basic run with the exception of the value C being lowered to 1, as we can see from the image, the results is close to the normal run, the only difference being the ratio of Faithfull and Coy being higher.

Different run 2

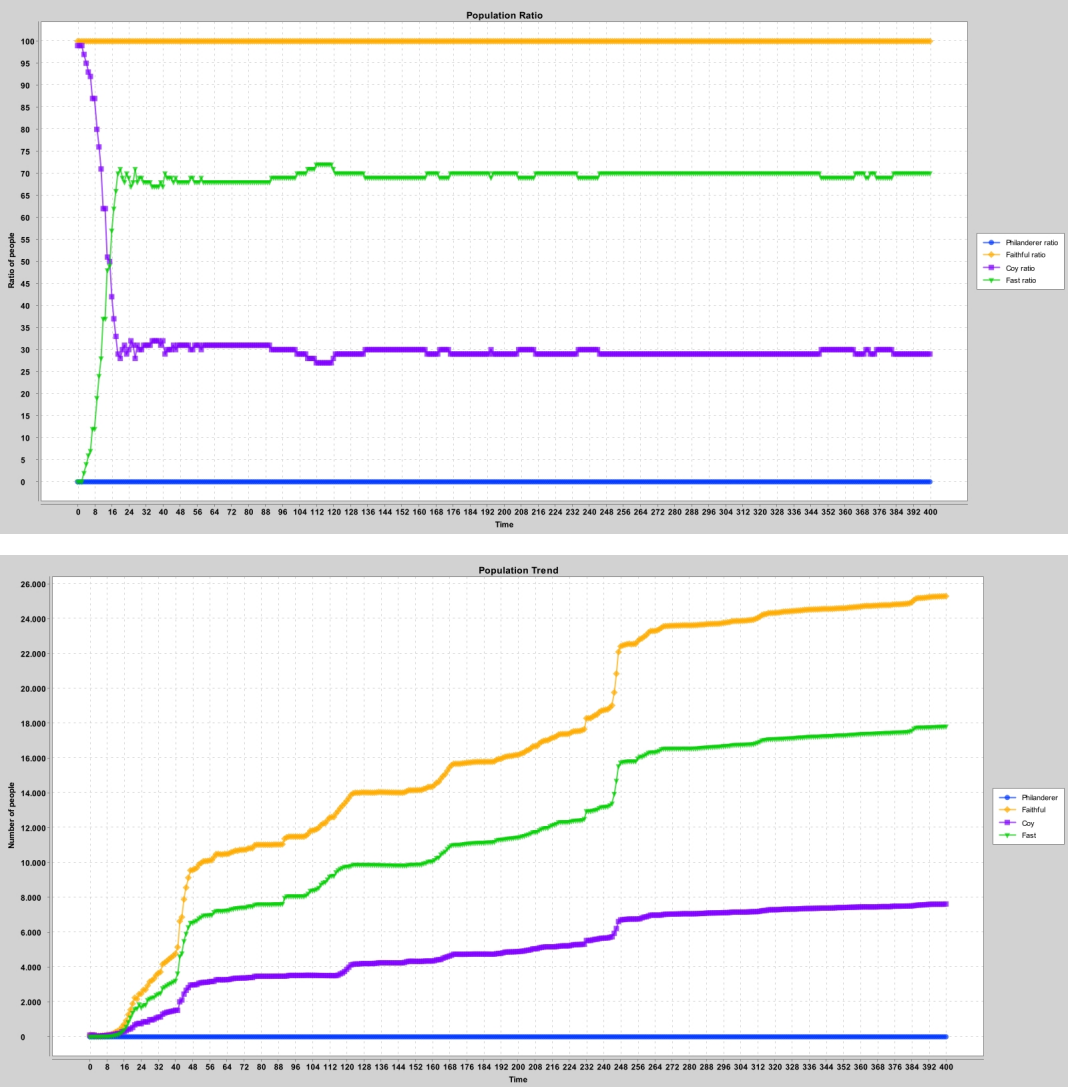


Figure 3: phil 0 - faith 100 - fast 1 - coy 100
a = 15 b = 20 c = 3

In this case since there are no Philanderer, a lot of Faithfull and Coy but only one Fast. Despite the numerical disanvantage of Fast, they grow very rapidly in confront of Coy, until the population stabilizes at 70% Fast and 30% Coy.

Different run 3

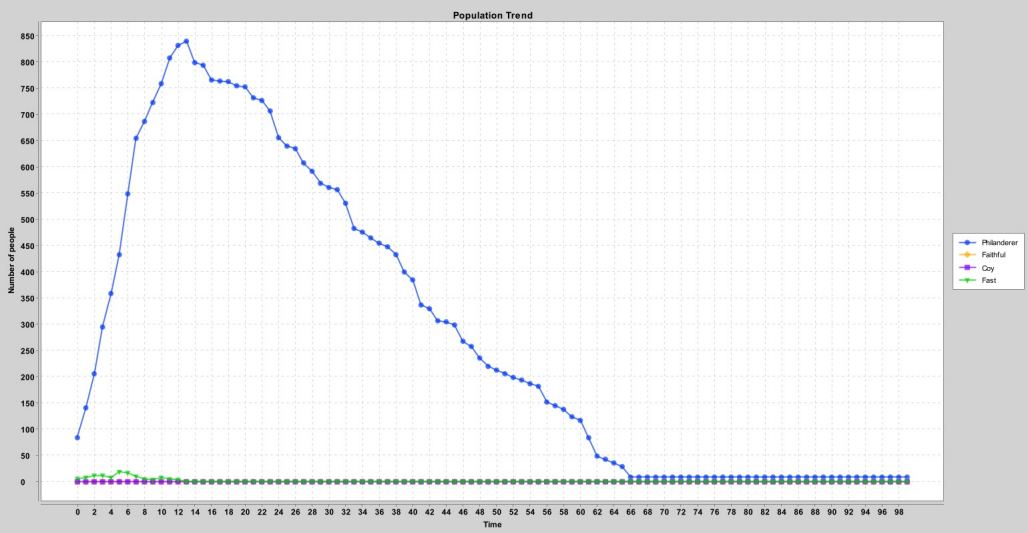


Figure 4: phil 100 - faith 0 - fast 100 - coy 0
a = 15 b = 20 c = 3

We tried to do a run with only Philanderers and Fast, initially there is a rapid grow of Philanderers since there are only Fast, but then with the exinction of all women, they decrease as well until the whole population is wiped off.

5 Conclusions

The primary goal of simulations is to reveal the underlying mechanisms that determine a system's behavior. More effectively, simulation may be used to predict a system's future behavior and figure out what you can do to alter it. That is, simulation may be used to forecast how the system will grow and adapt to its environment, allowing you to identify any essential adjustments that will help the system operate as you desire, in our case we've put our conceptual knowledge of Java and programming into practice.

This group project allowed us to gain communication and organization skills that helped us understanding how to work in teams and also made us take the most advantage of software that help teams to build a common project from remote(e.g. Github.com, overleaf.com, discord.com...)

One of the most effective methods to learn about how the world works is to work in groups. The experiences gained through group work help us become better individuals who can contribute effectively to society.