Q1

It is better to use a mix of the two programming paradigms together, for the whole software implementation.

Firstly, for large scale and complicated programming, using OOP can divide design into classes so that it can promote reusability of the code and can promote data abstraction and encapsulation (which organize code for better usability and extensibility). In this project, there are different kinds of cells and enemies, and they naturally come in the concept of classes and have the inheritance relationship, so using OOP to treat the system is suitable.

Meanwhile, for parts of the project that are algorithmic or simple, procedural programming can provide more efficient and straightforward ways to implement the system.

In conclusion, it is better to use a mix of the two programming paradigms together, for the whole software implementation.

Q2

(When I asked Dr Lam about this question, he says that we can write many possible problems, and as long as we get the 2 most critical errors it will be OK)

1. In the class definition of *Neutrophils*, the method *is_enemy* is not implemented. But it since it is the implementation of *WhiteBloodCell*, it should override all methods of *WhiteBloodCell*.
2. The *Enemy* interface has implemented an attribute *has_been_seen_before* and *antigens*. However, in popular programming languages like Java, the attribute of interfaces can only be constant, which is inconsistent with the design. One possible solution is to change *Enemy* from interface to abstract class.
3. The relationship between *Enemy* and *Antigen* could better be aggregation instead of being composition. Since in the real case, antigen could exist even though an enemy is deleted, and also different enemies can all have one kind of antigen, it would be more proper and efficient to design the antigen to exist a bit independently from the enemy, making the relationship aggregation instead of composition.
4. The relationship between *Bcell*, *Tcell* and *HelperTCell* isn't correct. In the description it is said that "B cells can make use of such information [list of floating point numbers] to create antibody to attack the enemies, either on their own or by working together with some T cells". This means that it should be not the case that only the *HelperTCell* has the attribute *enemy_to_characteristics* and method *get_enemy_characteristics*. On the contrary, they should be defined

in class *BCell* and *TCell* (and gets inherited by *HelperTCell*). (Or alternatively, just define in class *Lymphocytese* and gets inherited by *NKCell*, *BCell*, *TCell* and *HelperTCell*.)

5. The interface of *Enemy* and *Antigen* have no implementations. However, the implementations themselves are just high-level abstracts and cannot be used directly.
6. It is better to implement some of the interfaces to be private since they should not be changed by the outside. For example the *has_been_seen_before* in *Enemy*, and *is_activated* in *TCell*.

Q3

1. Encapsulation
    1. Private attribute *enemy_to_characteristics* in *HelperTCell and* Private attribute *enemy* in *Antibody* keeps implementation details of the classes hidden and encapsulated.
    2. All public methods (e.g., *is_enemy*, *kill*) expose only necessary functionality, and the implementation details are hidden within classes and the outside user only needs to provide the specific arguments to so the functionality can be successfully used.
    3. Also, all classes (eg. *NKCell*, *BCell*, *TCell*) encapsulates the details inside the classes.
2. Abstraction / information hiding
    1. In the implementation, all interfaces (*WhiteBloodCell, Enemy, Antigen*) and abstract classes (*Lymphocytes*) define common behavior for of its subclasses and (some abstract classes can) provide common implementation for its subclasses. In this way, the design can better divide the responsibilities and make the implementation more modularized. Also, the implementation of interfaces and abstract classes can hide the implementation details and let them be implemented in subclasses.
3. Inheritance
    1. In the implementation, sub classes (*Lymphocytes, Neutrophils*, *NKCell*, *BCell*, *TCell, HelperTCell*) are inherited from their parent classes or implemented from interfaces. In this way, common data structures are grouped and common function interfaces and behaviors are extracted in parent classes, making the implementation more clear and manageable, also enhances the code reusability.
4. Polymorphism
    1. In the implementation, *Lymphocytes, Neutrophils* are inherited from *WhiteBloodCell*; *NKCell*, *BCell* and *TCell* are inherited from *Lymphocytes; HelperTCell* is inherited from *TCell*. In these sub classes, Same method names

(*is_enemy*, *kill*) are used across different classes with different implementations. This polymorphism allow modification of class behaviours without altering common function interfaces, enabling localized code changes.

Q4

For strong association, a class is directly a member of another class. For example the relationship between *Antibody* and *Enemy*, the *Enemy* is directly an attribute of *Antibody* (but not initialized by the constructor of *Antibody* but created by the method *set_enemy*). When *Antibody* is deleted, *Enemy* is also deleted.

For weak association, a class just reference another class but does not control its lifetime. For example, for *WhiteBloodCell* and *Enemy*, while W*hiteBloodCell* reference *Enemy* in its method, it doesn't create or store *Enemy*. They are more independent, their lifecycle doesn't affect each other.

Q5

I think theoretically, we can store a list of floating point numbers. Since from the set up, each enemy has one to one correspondence to a set of antigens and a set of antigents has one to one correspondence to a list of floating point numbers, then by comparing the floating point numbers, we can make sure if the two kinds of enemies are the same. However, this has the below advantages and disadvantages:

Disadvantages:

1. Since floating point numbers are not precise, the comparison of floating point numbers may cause problems. If we set the threshold of difference allowed in comparison too small, then we may miss identify one kinds of enemy into two, we set the threshold of difference allowed in comparison too big, then we may miss identify different kinds of enemy into one kind.
2. Since converting enemies to its characteristics needs the method *enemy_to_characteristics* and its correctly only implemented in the *WhiteBloodCell*'s subclasses, we may need to define the method *enemy_to_characteristics* somewhere else and use it. This may cause further inconvenience. Also, since there is no *characteristics_to_enemy* method, the *Antibody* class cannot convert characteristics back to enemy, and may lead to potential loss of information.

Adventages:

1. Storing floating point numbers require less memory and less computer operations, leading to lower spatial and temporal complexity.
2. In this way *Antibody* is decoupled from *Enemy*, improving the modularity of the

code

Q6

Yes.

Since the relationship between *Antigen* and *Enemy* is composition relationship, *Antigen* objects are part of the Enemy object and cannot exist independently of the Enemy.

(But in case of special cases, it is safer to say that: if *Enemy* is destroyed, *Antigen* won't exist, but if *Antigen* destroyed, *Enemy* can still exist.)

Q7

Yes.

Since sub class *HelperTCell* have other resources that are not implemented to be freed by the parent class *TCell*, the parent class destruction method won't implement the code for freeing those additional resources. So even if parent class destruction methods are called, the additional resources of *HelperTCell* won't be freed. So it is important to override the destructor to add the implementations of freeing additional resources of *HelperTCell*.

Q8

Written in white_blood_cells.py.

Q9