**Description of the problems**

**Karatsuba.py:**

In this first problem we performed different multiplication processes.

I divided this problem into parts.

1. In the first part, the requirement is to multiply two integers and observe time complexity with increase of digits. I solved this part problem in such a way that firstly take two digits from the user and then convert them into integers and pass to the function then in a function after performing different operations return the result in string form and then display it on the console.
2. In the second part, the requirement is to take input as string then perform different operations according to the requirement and display it on the console. In this part, the only difference is that we take input as string otherwise it is same as the part 1.
3. In the third part, we take input and then perform multiplication processes and visualize it in the standard multiplication form. In this example, I use multiple print statements to visualize it according to the requirement of the manual.
4. In the fourth part, I use recursive function to perform the multiplication. This is also the requirement that we have to show the time consumed by the function. This function is tricky and some built-in functions are not working for this.
5. In the fifth part we perform the Karatsuba’s recursion function. This function is similar to the fourth part but the prime difference in the division process. This technique is very helpful to decrease the time complexity.
6. In the sixth part, the requirement is we have performed calculation of Karatsuba for base 10, but we have to use it for base 2 and base 16. The requirement is that we perform calculation using base 2 or base 16 and display the answer in the same format as the input is given.

This example is associated to the multiplication and tricky problems occur which were really helpful for logic building and other tasks similar to these problems.

**Friends.py:**

I divide this problem into two parts and then perform the calculation, which I mentioned below.

**Part (a): Algorithm with O(n2) Time Complexity**

**Algorithm Description:**

The straightforward algorithm compares each pair of users and checks if they overlap in time. We iterate through all pairs of users, and for each pair, we check if their time intervals overlap. If they do, we count the pair.

In the actual code, we have to check that at the same time which two friends are present in the class.

As the requirement is to provide the pseudo code of the problem so the pseudo code of the problem is given as:

friendSlower(Input):

Initialize an empty list to store the pairs

For each user i in Input:

For each user j after i in Input:

If user i and user j overlap in time:

Add the pair (i, j) to the list

Return the list of pairs

**Part (b): Algorithm with O(n log(n)) Time Complexity**

**Algorithm Description:**

The optimized algorithm sorts the events (enter and leave times) in ascending order of time. We then traverse the sorted events and keep track of the number of users present at each event. Whenever a user enters, we increment the count, and when they leave, we decrement the count. The number of pairs at any event is given by the count of users multiplied by the count of users minus one, divided by 2.

In the code we change our algorithm in such form that it takes less time than the previous algorithm.

As the requirement is to provide the pseudo so I have provided the pseudo code as following.

friendsFaster(Input):

Initialize an empty list to store the pairs

Initialize an empty list to store the events (start or end times)

For each user (start, end) in Input:

Add the event (start, 'start') to the list of events

Add the event (end, 'end') to the list of events

Sort the list of events by time

Initialize a variable 'count' to keep track of the number of users present

For each event (time, type) in the sorted events:

If type is 'start':

Increment the count of users

Else:

Decrement the count of users

If count is greater than or equal to 2:

Add pairs of users using the count (countC2) to the list of pairs

Return the list of pairs

In this we can perform our algorithm in less time.

**Toads.py:**

I have divided this problem into five parts which I mentioned below.

**Part (a): Straightforward Algorithm**

**Algorithm Description:**

The straightforward algorithm compares each pair of toads and checks if they both identify each other as trustworthy. We iterate through all pairs of toads, and for each pair, we check if both toads trust each other. If they do, we add both toads to the list of trustworthy toads.

The pseudo code of straight forward algorithm is given below:

getPopulation(n):

Initialize an empty list to store the toads

For i = 1 to n:

Create a new Toad object with a random is\_trustworthy value

Add the Toad object to the list

Return the list of toads

TruthFulToadsA(population):

Initialize an empty list to store the indices of trustworthy toads

For each pair of toads (i, j) in population:

If both toads i and j trust each other:

Add the indices i and j to the list

Return the list of indices of trustworthy toads

Next, we'll move on to part (b) and design a procedure to reduce the problem to the same problem with less than half the size using n/2 toad-to-toad comparisons.

**Part (b): Reduce the Problem Size**

**Procedure Description:**

Given a population of toads, we can pair them up and compare their trustworthiness. We can then choose the trustworthy toads based on the comparison results and form a new population with less than half the size.

**Procedure Steps:**

1. Pair up the toads and compare their trustworthiness.
2. If both toads in a pair are trustworthy, consider them trustworthy.
3. If one of the toads in a pair is trustworthy, randomly choose whether to consider them trustworthy.
4. Form a new population with the trustworthy toads.

The pseudo code of problem is given below.

TruthFulToadsB(population):

Initialize an empty list to store the indices of trustworthy toads

Initialize a new list to store the reduced population

n = length of population

For i = 1 to n (incrementing by 2):

If both toads i and i+1 are trustworthy:

Add both indices i and i+1 to the list of trustworthy toads

Else:

Randomly choose one of the toads and add its index to the list

Return the list of indices of trustworthy toads

Next, we'll move on to part (c) and extend the argument for odd n.

**Part (c): Extend the Argument for Odd n**

To extend the argument for odd n, we can use the same logic as for even n but handle the last toad separately.

**Procedure Steps:**

1. Pair up the toads and compare their trustworthiness, similar to even n.
2. If both toads in a pair are trustworthy, consider them trustworthy.
3. If one of the toads in a pair is trustworthy, randomly choose whether to consider them trustworthy.
4. If there's an odd toad remaining, randomly choose whether to consider it trustworthy.

The pseudo code of problem is:

TruthFulToadsC(population):

Initialize an empty list to store the indices of trustworthy toads

Initialize a new list to store the reduced population

n = length of population

For i = 1 to n-1 (incrementing by 2):

If both toads i and i+1 are trustworthy:

Add both indices i and i+1 to the list of trustworthy toads

Else:

Randomly choose one of the toads and add its index to the list

# Handle the last toad for odd n

If n is odd:

Randomly choose whether to consider the last toad trustworthy

Return the list of indices of trustworthy toads

**Part (d): Recursive Algorithm to Find a Trustworthy Toad**

We'll design a recursive algorithm that divides the population into smaller groups, compares the toads in each group, and recursively calls itself on the groups with more than half trustworthy toads.

**Algorithm Description:**

1. If the population size is 1, return the only toad in the population as the trustworthy toad.
2. Reduce the population size using the procedure from part (b) or (c).
3. Recur on the reduced population until a trustworthy toad is found.

The pseudo code is:

FindTrustworthyToad(population):

If the population size is 1:

Return the only toad in the population as the trustworthy toad

# Reduce the population size using the procedure from part (b) or (c)

trustworthy\_toads, reduced\_population = TruthFulToadsB(population) # For even n

# trustworthy\_toads, reduced\_population = TruthFulToadsC(population) # For odd n

# Recur on the reduced population until a trustworthy toad is found

return FindTrustworthyToad(reduced\_population)

**Part (e): Formal Proof of Correctness using Induction**

**Inductive Hypothesis:**

The recursive algorithm FindTrustworthyToad(population) correctly finds a trustworthy toad for any population of toads.

**Base Case:**

For a population of size 1, the algorithm returns the only toad as the trustworthy toad. This is correct.

**Inductive Step:**

Assume the algorithm is correct for all populations of size less than k (where k > 1).

Now consider a population of size k. The algorithm reduces the population and calls itself on the reduced population. By the inductive hypothesis, the algorithm correctly finds a trustworthy toad for the reduced population.

The algorithm always reduces the population size using the procedure from part (b) or (c) such that the new population size is less than or equal to k/2. By the inductive hypothesis, the algorithm will correctly find a trustworthy toad for the new population.

Hence, by induction, the algorithm is correct for all population sizes.