MapCoder: Multi-Agent Code Generation for Competitive Problem Solving

---

An example containing problem from HumanEval Dataset (k=5, t=5)

Prompt for Self-retrieval Agent:

Given a problem, provide relevant problems then identify the algorithm behind it and also explain the tutorial of the algorithm.

We will refer this as **original problem** from the next time

# Problem:

```
def sum_squares(lst):
    """You are given a list of numbers.
    You need to return the sum of squared numbers in the given list,
    round each element in the list to the upper int(Ceiling) first.
    Examples:
    For lst = [1,2,3] the output should be 14
    For lst = [1,4,9] the output should be 98
    For lst = [1,3,5,7] the output should be 84
    For lst = [1.4,4.2,0] the output should be 29
    For lst = [-2.4,1,1] the output should be 6
    """
```

# Exemplars:

Recall five (05) relevant and distinct problems (different from problem mentioned above). For each problem,
1. describe it
2. generate Python3 code step by step to solve that problem
3. finally generate a planning to solve that problem

# Algorithm:

Important:

Your response must follow the following xml format-

<root>

<problem>

# Recall five (05) relevant and distinct problems (different from problem mentioned above). Write each problem in the following format.

<description> # Describe the problem. **</description>**

<code> # Let's think step by step to solve this problem in Python3 programming language. **</code>**

<planning> # Planning to solve this problem. **</planning>**

</problem>

# similarly add more problems here...

<algorithm>

# Identify the algorithm (Brute-force, Dynamic Programming, Divide-and-conquer, Greedy, Backtracking, Recursive, Binary search, and so on) that needs to be used to solve the original problem.

# Write a useful tutorial about the above mentioned algorithms. Provide a high level generic tutorial for solving this types of problem. Do not generate code.

</algorithm>

</root>

Response from Self-retrieval Agent:

<root>
<problem>
<description> 1. Problem: Find the sum of all even numbers in a list. **</description>**
<code>

```
def sum_even(lst):
    total = 0
    for num in lst:
        if num % 2 == 0:
            total += num
    return total
```

</code>
<planning>

To solve this problem, we can iterate through the list and check if each number is even. If it is, we add it to a running total. Finally, we return the total sum of all even numbers in the list.

</planning>
</problem>

<problem>
<description> 2. Problem: Find the product of all numbers in a list. **</description>**
<code>

```
def product_of_numbers(lst):
    product = 1
    for num in lst:
        product *= num
    return product
```

</code>
<planning>

To solve this problem, we can iterate through the list and multiply each number with a running product. Finally, we return the total product of all numbers in the list.

</planning>
</problem>

Three more plans are generated in similar way…

We will refer this as **Retrieved algorithm** from the next time

<algorithm>

The algorithm behind the original problem of summing squared numbers in a list is a simple iteration through the list and performing a calculation on each element. This can be solved using a brute-force algorithm.
Tutorial:
Brute-force algorithm involves iterating through the input and checking all possible solutions to find the best one. In the case of summing squared numbers in a list, we iterate through the list, square each number, round it to the upper integer, and then sum them up. This algorithm is straightforward and easy to implement, but it may not be the most efficient for large inputs.

</ algorithm>
</root>

Prompt for Planning Agent (Plan Generation):

Given a competitive programming problem generate a concrete planning to solve the problem.

# Problem: 1. Problem: Find the sum of all even numbers in a list.
# Planning: To solve this problem, we can iterate through the list and check if each number is even. If it is, we add it to a running total. Finally, we return the total sum of all even numbers in the list.
## Relevant Algorithm to solve the next problem: *{Retrieved algorithm}*
## Problem to be solved: *{Original problem}*
## Sample Test cases:

```
assert sum_squares([1,2,3])==14
assert sum_squares([1,4,9])==98
assert sum_squares([1,3,5,7])==84
assert sum_squares([1.4,4.2,0])==29
assert sum_squares([-2.4,1,1])==6
```

We will refer this as **Sample IO** from the next time

## Planning:

Important: You should give only the planning to solve the problem. Do not add extra explanation or words.

Response from Planning Agent (Plan Generation):

To solve this problem, we can iterate through the list, round each number to the upper integer, square it, and then add it to a running total. Finally, we return the total sum of all squared numbers in the list.

Prompt for Planning Agent (Confidence Generation):

Given a competitive programming problem and a plan to solve the problem in Python3, tell whether the plan is correct to solve this problem.

# Problem: *{Original problem}*
# Planning:
To solve this problem, we can iterate through the list, round each number to the upper integer, square it, and then add it to a running total. Finally, we return the total sum of all squared numbers in the list.

Important: Your response must follow the following xml format-
<root>
<explanation>
Discuss whether the given competitive programming problem is solvable by using the above mentioned planning.
</explanation>
<confidence>
Confidence score regarding the solvability of the problem. Must be an integer between 0 and 100.
</confidence>
</root>

Response from Planning Agent (Confidence Generation):

```
<root>
    <explanation>
        The given competitive programming problem is solvable by using the above mentioned planning. The
        plan correctly outlines the steps to iterate through the list, round each number to the upper integer,
        square it, and then add it to a running total. Finally, returning the total sum of all squared numbers in
        the list.
    </explanation>
    <confidence> 100 </confidence>
</root>
```

Similarly generate planning and their confidence using other examples….

Decision: As First Planning gets 100 as confidence score Coding Agent will work with first Planning.

Prompt for Coding Agent:

Given a competitive programming problem generate Python3 code to solve the problem.

## Relevant Algorithm to solve the next problem: *{Retrieved algorithm}*

## Problem to be solved: *{Original problem}*

## Planning:
To solve this problem, we can iterate through the list, round each number to the upper integer, square it, and then add it to a running total. Finally, we return the total sum of all squared numbers in the list.

## Sample Test cases: *{Sample I/O}*

## Let's think step by step.

Important:
## Your response must contain only the Python3 code to solve this problem. Do not add extra explanation or words.

Response from Coding Agent:

```
def sum_squares(lst):
    total = 0
    for num in lst:
        total += (int(num) ** 2)
    return total
```

Decision: *Testing using Sample I/O will fail because this is wrong implementation. So, our approach will go to Debugging Agent.*

Prompt for Debugging Agent: (1st iteration)

Given a competitive programming problem you have generated Python3 code to solve the problem. But the generated code can not pass sample test cases. Improve your code to solve the problem correctly.

## Relevant Algorithm to solve the next problem: *{Retrieved algorithm}*
## Problem to be solved: *{Original problem}*
## Planning: {Current Planning}
## Code: {Current Implementation}
## Test Report:
passed in test case: assert sum_squares([1,2,3]) == 14
passed in test case: assert sum_squares([1,4,9]) == 98
passed in test case: assert sum_squares([1,3,5,7]) == 84
failed in test case: assert sum_squares([1.4,4.2,0]) == 29
passed in test case: assert sum_squares([-2.4,1,1]) == 6
## Modified Planning:
## Let's think step by step to modify Python3 Code for solving this problem.

Important:
## Your response must contain the modified planning and then the Python3 code inside ``` block to solve this problem.

Response from Debugging Agent: (1st iteration)

## Modified Planning:
To modify the code, we need to round each number to the upper integer before squaring it. We can use the math.ceil() function to achieve this. After rounding each number, we can then square it and add it to the total sum.

## Code:

~~~
```Python3
import math

def sum_squares(lst):
    total = 0
    for num in lst:
        total += (math.ceil(num) ** 2)
    return total
```
~~~

Decision: This time all Sample I/O passed. We return this implementation as the Final Code for final evaluation. Finally, we evaluated this code with private test case and our implementation passed the evaluation.

An example containing problem from CodeContest Dataset (k=3, t=5)

We will refer this as **original problem** from the next time

Prompt for Self-retrieval Agent:

Given a problem, provide relevant problems then identify the algorithm behind it and also explain the tutorial of the algorithm.

# Problem:

```
Luntik has decided to try singing. He has a one-minute songs, b two-minute songs
and c three-minute songs. He wants to distribute all songs into two concerts such
that every song should be included to exactly one concert.
He wants to make the absolute difference of durations of the concerts as small as
possible. The duration of the concert is the sum of durations of all songs in that
concert.
Please help Luntik and find the minimal possible difference in minutes between the
concerts durations.

Input
The first line contains a single integer t (1 ≤ t ≤ 1000) — the number of test
cases.
Each test case consists of one line containing three integers a, b, c (1 ≤ a, b, c
≤ 10^9) — the number of one-minute, two-minute and three-minute songs.
Output
For each test case print the minimal possible difference in minutes between the
concerts durations.

Example
Input
4
1 1 1
2 1 3
5 5 5
1 1 2
Output
0
1
0
1

Note
In the first test case, Luntik can include a one-minute song and a two-minute song
into the first concert, and a three-minute song into the second concert. Then the
difference will be equal to 0.
In the second test case, Luntik can include two one-minute songs and a two-minute
song and a three-minute song into the first concert, and two three-minute songs
into the second concert. The duration of the first concert will be 1 + 1 + 2 + 3 =
7, the duration of the second concert will be 6. The difference of them is |7-6| =
1.    """
-------
Important Note: You must follow the input output format. Input must be taken from
standard input and output must be given to standard output. The code will be
tested against multiple test cases and all the test cases must be passed.
```

# Exemplars:
Recall three (03) relevant and distinct problems (different from problem mentioned above). For each problem,
1. describe it
2. generate Python3 code step by step to solve that problem
3. finally generate a planning to solve that problem

# Algorithm:

Important:
Your response must follow the following xml format-

<root>
<problem>
# Recall three (03) relevant and distinct problems (different from problem mentioned above). Write each problem in the following format.
<description> # Describe the problem. **</description>**
<code> # Let's think step by step to solve this problem in Python3 programming language. **</code>**
<planning> # Planning to solve this problem. **</planning>**
</problem>

# similarly add more problems here...

<algorithm>
# Identify the algorithm (Brute-force, Dynamic Programming, Divide-and-conquer, Greedy, Backtracking, Recursive, Binary search, and so on) that needs to be used to solve the original problem.
# Write a useful tutorial about the above mentioned algorithms. Provide a high level generic tutorial for solving this types of problem. Do not generate code.
</algorithm>
</root>

Response from Self-retrieval Agent:

<root>
<problem>
<description>
Problem 1:
You are given an array of integers, find the maximum sum of a contiguous subarray within the array.
</description>
<code>

```
def maxSubArray(nums):
    max_sum = nums[0]
    current_sum = nums[0]

    for i in range(1, len(nums)):
        current_sum = max(nums[i], current_sum + nums[i])
        max_sum = max(max_sum, current_sum)

    return max_sum
```

</code>
<planning>
- Initialize two variables max_sum and current_sum to the first element of the array.

- Iterate through the array and at each index, update the current_sum to be the maximum of the current element and the sum of the current element and the previous current_sum.
- Update max_sum to be the maximum of max_sum and current_sum.
- Return max_sum at the end.

</planning>

</problem>

<problem>

<description>

Problem 2:

You are given a list of tasks with their start and end times. Find the minimum number of meeting rooms required to schedule all the tasks.

</description>

<code>

```
import heapq

def minMeetingRooms(intervals):
    if not intervals:
        return 0

    intervals.sort(key=lambda x: x[0])

    rooms = []
    heapq.heappush(rooms, intervals[0][1])

    for i in intervals[1:]:
        if rooms[0] <= i[0]:
            heapq.heappop(rooms)
        heapq.heappush(rooms, i[1])

    return len(rooms)
```

</code>

<planning>

- Sort the intervals based on their start times.
- Create a min heap to store the end times of the meetings.
- Push the end time of the first meeting into the heap.
- Iterate through the intervals, if the start time of the current interval is greater than or equal to the end time at the top of the heap, pop the end time from the heap. Push the end time of the current interval into the heap.
- Return the length of the heap at the end.

</planning>

</problem>

<problem>

<description>

Problem 3:

You are given a list of intervals representing the start and end times of meetings. Find the maximum number of overlapping intervals.

</description>

<code>

```
def maxOverlap(intervals):
    if not intervals:
        return 0

    start_times = sorted([i[0] for i in intervals])
    end_times = sorted([i[1] for i in intervals])

    max_overlap = 0
    current_overlap = 0
    start_ptr = 0
    end_ptr = 0

    while start_ptr < len(intervals):
        if start_times[start_ptr] < end_times[end_ptr]:
            current_overlap += 1
            max_overlap = max(max_overlap, current_overlap)
            start_ptr += 1
        else:
            current_overlap -= 1
            end_ptr += 1

    return max_overlap
```

</code>

<planning>

- Sort the start times and end times of the intervals separately.
- Initialize max_overlap and current_overlap to 0, and start and end pointers to 0.
- Iterate through the start times, if the current start time is less than the current end time, increment current_overlap and update max_overlap. Increment the start pointer.
- If the current start time is greater than or equal to the current end time, decrement current_overlap and increment the end pointer.
- Return max_overlap at the end.

</planning>

</problem>

We will refer this as **Retrieved algorithm** from the next time

<algorithm>

The algorithm that can be used to solve the original problem of distributing songs into two concerts with minimal possible difference in durations is the Greedy algorithm. The Greedy algorithm makes the locally optimal choice at each stage with the hope of finding a global optimum.

To solve the original problem, we can use the following steps:
1. Sort the songs by their durations in non-increasing order.
2. Start with two empty concerts.
3. Iterate through the sorted songs and at each stage, assign the current song to the concert with the smaller total duration.
4. Calculate the absolute difference in durations between the two concerts at the end.

This approach ensures that we are making the locally optimal choice at each stage by assigning the current song to the concert with the smaller total duration, and it leads to the minimal possible difference in durations between the concerts.

</ algorithm>

</root>

Prompt for First Planning Agent (Plan Generation):

Given a competitive programming problem generate a concrete planning to solve the problem.

# Problem:
Problem 1:
You are given an array of integers, find the maximum sum of a contiguous subarray within the array.

# Planning:
- Initialize two variables max_sum and current_sum to the first element of the array.
- Iterate through the array and at each index, update the current_sum to be the maximum of the current element and the sum of the current element and the previous current_sum.
- Update max_sum to be the maximum of max_sum and current_sum.
- Return max_sum at the end.

## Relevant Algorithm to solve the next problem: *{Retrieved algorithm}*
## Problem to be solved: *{Original problem}*
## Sample Test cases:

```
Input:
4
1 1 1
2 1 3
5 5 5
1 1 2
Expected output:
0 1 0 1
```

We will refer this as **Sample IO** from the next time

## Planning:

Important: You should give only the planning to solve the problem. Do not add extra explanation or words.

Response from First Planning Agent (Plan Generation):

- Sort the songs by their durations in non-increasing order.
- Start with two empty concerts.
- Iterate through the sorted songs and at each stage, assign the current song to the concert with the smaller total duration.
- Calculate the absolute difference in durations between the two concerts at the end.

Prompt for First Planning Agent (Confidence Generation):

Given a competitive programming problem and a plan to solve the problem in Python3, tell whether the plan is correct to solve this problem.

# Problem: *{Original problem}*
# Planning:
- Sort the songs by their durations in non-increasing order.
- Start with two empty concerts.
- Iterate through the sorted songs and at each stage, assign the current song to the concert with the smaller total duration.

- Calculate the absolute difference in durations between the two concerts at the end.

Important: Your response must follow the following xml format-
<root>
    <explanation>
        Discuss whether the given competitive programming problem is solvable by using the above mentioned planning.
    </explanation>
    <confidence>
        Confidence score regarding the solvability of the problem. Must be an integer between 0 and 100.
    </confidence>
</root>

Response from First Planning Agent (Confidence Generation):

<root>
    <explanation>
        The given competitive programming problem is solvable by using the above mentioned planning. The plan involves sorting the songs by duration and then assigning them to two concerts in a way that minimizes the absolute difference in durations between the concerts.
    </explanation>
    <confidence> 100 **</confidence>**
</root>

Prompt for Second Planning Agent (Plan Generation):

Given a competitive programming problem generate a concrete planning to solve the problem.

# Problem:
Problem 2:
You are given a list of tasks with their start and end times. Find the minimum number of meeting rooms required to schedule all the tasks.

# Planning:
- Sort the intervals based on their start times.
- Create a min heap to store the end times of the meetings.
- Push the end time of the first meeting into the heap.
- Iterate through the intervals, if the start time of the current interval is greater than or equal to the end time at the top of the heap, pop the end time from the heap. Push the end time of the current interval into the heap.
- Return the length of the heap at the end.

## Relevant Algorithm to solve the next problem: *{Retrieved algorithm}*
## Problem to be solved: *{Original problem}*
## Sample Test cases: {Sample I/O}
## Planning:

Important: You should give only the planning to solve the problem. Do not add extra explanation or words.

Response from Second Planning Agent (Plan Generation):

- Sort the songs by their durations in non-increasing order.
- Start with two empty concerts.

- Iterate through the sorted songs and at each stage, assign the current song to the concert with the smaller total duration.
- Calculate the absolute difference in durations between the two concerts at the end.

Prompt for Second Planning Agent (Confidence Generation):

Given a competitive programming problem and a plan to solve the problem in Python3, tell whether the plan is correct to solve this problem.

# Problem: *{Original problem}*
# Planning:
- Sort the songs by their durations in non-increasing order.
- Start with two empty concerts.
- Iterate through the sorted songs and at each stage, assign the current song to the concert with the smaller total duration.
- Calculate the absolute difference in durations between the two concerts at the end.

Important: Your response must follow the following xml format-
<root>
    <explanation>
        Discuss whether the given competitive programming problem is solvable by using the above mentioned planning.
    </explanation>
    <confidence>
        Confidence score regarding the solvability of the problem. Must be an integer between 0 and 100.
    </confidence>
</root>

Response from Second Planning Agent (Confidence Generation):

<root>
    <explanation>
        The given competitive programming problem is solvable by using the above mentioned planning. The plan involves sorting the songs by duration, then assigning the songs to two concerts in a way that minimizes the absolute difference in durations between the concerts.
    </explanation>
    <confidence> 100 **</confidence>**
</root>

Similarly generate the planning and the confidence of last example….

Decision: As First Planning gets 100 as confidence score Coding Agent will work with first Planning.

Prompt for Coding Agent:

Given a competitive programming problem generate Python3 code to solve the problem.
## Relevant Algorithm to solve the next problem: *{Retrieved algorithm}*

## Problem to be solved: *{Original problem}*
## Planning:
- Sort the songs by their durations in non-increasing order.
- Start with two empty concerts.
- Iterate through the sorted songs and at each stage, assign the current song to the concert with the smaller total duration.
- Calculate the absolute difference in durations between the two concerts at the end.

## Sample Test cases: *{Sample I/O}*
## Let's think step by step.

Important:
## Note: Strictly follow the input and output format. The input should be taken from Standard input and output should be given to standard output. If you are writing a function then after the function definition take input using `input()` function then call the function with specified parameters and finally print the output of the function. Do not add extra print statement otherwise it will failed the test cases.
## Your response must contain only the Python3 code to solve this problem. Do not add extra explanation or words.

Response from Coding Agent:

```
def min_difference_in_concerts(t, test_cases):
    for case in test_cases:
        a, b, c = case
        songs = [(1, a), (2, b), (3, c)]
        songs.sort(key=lambda x: x[0], reverse=True)
        concert1 = 0
        concert2 = 0
        for song in songs:
            if concert1 <= concert2:
                concert1 += song[0] * song[1]
            else:
                concert2 += song[0] * song[1]
        print(abs(concert1 - concert2))

t = 4
test_cases = [(1, 1, 1), (2, 1, 3), (5, 5, 5), (1, 1, 2)]
min_difference_in_concerts(t, test_cases)
```

***Decision:** Testing using Sample I/O will fail because of incorrect implementation. The main problem with this code is that it does not take input from the console. It writes a function and just calls it with fixed value. So, our approach will go to the Debugging Agent.*

Prompt for Debugging Agent: (1st iteration)

Given a competitive programming problem you have generated Python3 code to solve the problem. But the generated code can not pass sample test cases. Improve your code to solve the problem correctly.

## Relevant Algorithm to solve the next problem: *{Retrieved algorithm}*
## Problem to be solved: *{Original problem}*

## Planning:

- Sort the songs by their durations in non-increasing order.
- Start with two empty concerts.
- Iterate through the sorted songs and at each stage, assign the current song to the concert with the smaller total duration.
- Calculate the absolute difference in durations between the two concerts at the end.

## Code:

```
def min_difference_in_concerts(t, test_cases):
    for case in test_cases:
        a, b, c = case
        songs = [(1, a), (2, b), (3, c)]
        songs.sort(key=lambda x: x[0], reverse=True)
        concert1 = 0
        concert2 = 0
        for song in songs:
            if concert1 <= concert2:
                concert1 += song[0] * song[1]
            else:
                concert2 += song[0] * song[1]
        print(abs(concert1 - concert2))

t = 4
test_cases = [(1, 1, 1), (2, 1, 3), (5, 5, 5), (1, 1, 2)]
min_difference_in_concerts(t, test_cases)
```

## Test Report:

## Tested passed:

No test cases passed.

## Tests failed:

Input:
4
1 1 1
2 1 3
5 5 5
1 1 2
Expected Output:
0 1 0 1

Your Output:
0 5 0 3

## Modified Planning:

## Let's think step by step to modify Python3 Code for solving this problem.

Important:

## Note: Strictly follow the input and output format. The input should be taken from Standard input and output should be given to standard output. If you are writing a function then after the function definition take input using `input()` function then call the function with specified parameters and finally print the output of the function. Do not add extra print statement otherwise it will failed the test cases.

## Your response must contain the modified planning and then the Python3 code inside ``` block to solve this problem.

Response from Debugging Agent: (1st iteration)

## Modified Planning:

1. Read the integer t from input to determine the number of test cases.
2. Create a function min_difference_in_concerts(t, test_cases) that takes t and test_cases as input parameters.
3. For each test case in test_cases, do the following:
   - Read the integers a, b, and c from input.
   - Calculate the total duration of each type of song (1-minute, 2-minute, 3-minute).
   - Calculate the total duration of all songs.
   - Calculate the minimum possible difference in minutes between the concerts durations.
   - Print the minimum possible difference in minutes between the concerts durations.
4. Call the function min_difference_in_concerts with t and test_cases as input parameters.

## Code:

```Python3
def min_difference_in_concerts(t, test_cases):
    for case in test_cases:
        a, b, c = case
        total_duration = a + 2*b + 3*c
        if total_duration % 2 == 0:
            print(total_duration // 2)
        else:
            print((total_duration - 1) // 2)

t = int(input())
test_cases = []
for _ in range(t):
    a, b, c = map(int, input().split())
    test_cases.append((a, b, c))

min_difference_in_concerts(t, test_cases)
```

***Decision:** This time, it takes input from the console and gives output to the console. It correct that part. But again, it failed because of incorrect implementation. So, the Debugging Agent tries to improve the code again.*

Prompt for Debugging Agent: (2nd iteration)

Given a competitive programming problem you have generated Python3 code to solve the problem. But the generated code can not pass sample test cases. Improve your code to solve the problem correctly.

## Relevant Algorithm to solve the next problem: *{Retrieved algorithm}*
## Problem to be solved: *{Original problem}*
## Planning:

1. Read the integer t from input to determine the number of test cases.
2. Create a function min_difference_in_concerts(t, test_cases) that takes t and test_cases as input parameters.
3. For each test case in test_cases, do the following:
   - Read the integers a, b, and c from input.
   - Calculate the total duration of each type of song (1-minute, 2-minute, 3-minute).
   - Calculate the total duration of all songs.
   - Calculate the minimum possible difference in minutes between the concerts durations.

- Print the minimum possible difference in minutes between the concerts durations.

4. Call the function min_difference_in_concerts with t and test_cases as input parameters.

## Code:

```
def min_difference_in_concerts(t, test_cases):
    for case in test_cases:
        a, b, c = case
        total_duration = a + 2*b + 3*c
        if total_duration % 2 == 0:
            print(total_duration // 2)
        else:
            print((total_duration - 1) // 2)

t = int(input())
test_cases = []
for _ in range(t):
    a, b, c = map(int, input().split())
    test_cases.append((a, b, c))

min_difference_in_concerts(t, test_cases)
```

## Test Report:
## Tested passed:
No test cases passed.

## Tests failed:
Input:
4
1 1 1
2 1 3
5 5 5
1 1 2
Expected Output:
0 1 0 1

Your Output:
3 6 15 4

## Modified Planning:
## Let's think step by step to modify Python3 Code for solving this problem.

Important:
## Note: Strictly follow the input and output format. The input should be taken from Standard input and output should be given to standard output. If you are writing a function then after the function definition take input using `input()` function then call the function with specified parameters and finally print the output of the function. Do not add extra print statement otherwise it will failed the test cases.

## Your response must contain the modified planning and then the Python3 code inside ``` block to solve this problem.

Response from Debugging Agent: (2nd iteration)

## Modified Planning:
1. Read the integer t from input to determine the number of test cases.
2. Create a function min_difference_in_concerts(t, test_cases) that takes t and test_cases as input parameters.
3. For each test case in test_cases, do the following:
   - Read the integers a, b, and c from input.
   - Calculate the total duration of each type of song (1-minute, 2-minute, 3-minute).
   - Calculate the total duration of all songs.
   - Calculate the minimum possible difference in minutes between the concerts durations.
   - Append the minimum possible difference to a list.
4. Return the list of minimum possible differences.

## Code:

~~~
```Python3
def min_difference_in_concerts(t, test_cases):
    result = []
    for case in test_cases:
        a, b, c = case
        total_duration = a + 2*b + 3*c
        if total_duration % 2 == 0:
            result.append(total_duration // 2)
        else:
            result.append((total_duration + 1) // 2)
    return result

t = int(input())
test_cases = []
for _ in range(t):
    a, b, c = map(int, input().split())
    test_cases.append((a, b, c))

output = min_difference_in_concerts(t, test_cases)
for res in output:
    print(res)
```
~~~

Decision: *Again failed because of incorrect implementation. So, the Debugging Agent tries to improve the code again.*

Prompt for Debugging Agent: (3rd iteration)

Given a competitive programming problem you have generated Python3 code to solve the problem. But the generated code can not pass sample test cases. Improve your code to solve the problem correctly.

## Relevant Algorithm to solve the next problem: *{Retrieved algorithm}*
## Problem to be solved: *{Original problem}*

## Planning:

1. Read the integer t from input to determine the number of test cases.
2. Create a function min_difference_in_concerts(t, test_cases) that takes t and test_cases as input parameters.
3. For each test case in test_cases, do the following:
    - Read the integers a, b, and c from input.
    - Calculate the total duration of each type of song (1-minute, 2-minute, 3-minute).
    - Calculate the total duration of all songs.
    - Calculate the minimum possible difference in minutes between the concerts durations.
    - Append the minimum possible difference to a list.
4. Return the list of minimum possible differences.

## Code:

```
def min_difference_in_concerts(t, test_cases):
    result = []
    for case in test_cases:
        a, b, c = case
        total_duration = a + 2*b + 3*c
        if total_duration % 2 == 0:
            result.append(total_duration // 2)
        else:
            result.append((total_duration + 1) // 2)
    return result

t = int(input())
test_cases = []
for _ in range(t):
    a, b, c = map(int, input().split())
    test_cases.append((a, b, c))

output = min_difference_in_concerts(t, test_cases)
for res in output:
    print(res)
```

## Test Report:

## Tested passed:

No test cases passed.

## Tests failed:

Input:

```
4
1 1 1
2 1 3
5 5 5
1 1 2
```

Expected Output:

```
0 1 0 1
```

Your Output:

```
3 7 15 5
```

## Modified Planning:

## Let's think step by step to modify Python3 Code for solving this problem.

Important:

## Note: Strictly follow the input and output format. The input should be taken from Standard input and output should be given to standard output. If you are writing a function then after the function definition take input using `input()` function then call the function with specified parameters and finally print the output of the function. Do not add extra print statement otherwise it will failed the test cases.

## Your response must contain the modified planning and then the Python3 code inside ``` block to solve this problem.

Response from Debugging Agent: (3rd iteration)

## Modified Planning:

1. Read the integer t from input to determine the number of test cases.
2. Create a function min_difference_in_concerts(t, test_cases) that takes t and test_cases as input parameters.
3. For each test case in test_cases, do the following:
   - Read the integers a, b, and c from input.
   - Calculate the total duration of each type of song (1-minute, 2-minute, 3-minute).
   - Calculate the total duration of all songs.
   - Calculate the minimum possible difference in minutes between the concerts durations.
   - Append the minimum possible difference to a list.
4. Return the list of minimum possible differences.

## Code:

~~~
```Python3
def min_difference_in_concerts(t, test_cases):
    result = []
    for case in test_cases:
        a, b, c = case
        total_duration = a + 2*b + 3*c
        if total_duration % 2 == 0:
            result.append(0)
        else:
            result.append(1)
    return result

t = int(input())
test_cases = []
for _ in range(t):
    a, b, c = map(int, input().split())
    test_cases.append((a, b, c))

output = min_difference_in_concerts(t, test_cases)
for res in output:
    print(res)
```
~~~

Decision: This time all Sample I/O passed. We return this implementation as the Final Code for final evaluation. Finally, we evaluated this code with a private test case, and our implementation passed the evaluation. This way Debugging Agents debug the code like a human.