Assignment 02

Instructions

1. Write your functions according to the signature provided below in a python(.py) file and submit them to the autograder for evaluation.
2. The autograder has a limited number of attempts. Hence, test it thoroughly before submitting it for evaluation.
3. Save your python file as text(.txt) file and submit it to the canvas.
4. If the assignment has theoretical questions, upload those answers as a PDF file on the canvas.
5. Submission on the canvas is mandatory and the canvas submitted text file should be the same as that of the final submission of the autograder. If not, marks will be deducted.
6. No submission on Canvas or no submission on autograder then marks will be deducted.
7. Access the auto grader at https://c200.luddy.indiana.edu.

Question 1

You're enjoying your summer, and you visited New York City to explore. While exploring, you stumbled across the "death street," a very small street. Only one person can pass at a time on this death street, unless they are moving in the same direction. When two people travel in opposite directions, they will both die because at some time they will face each other and won't be able to move further. You are watching everyone while standing on the sidewalk. Everyone is rushing since that's just how New York City is, so instead of abiding by the single person or multiple person but all in same direction on the street guideline, individuals started walking in both directions and some of them were even standing still.

On the sidewalk, you are standing and scanning this traffic in front of you. The person is either moving right (denoted by 1), moving left (denoted by -1), or standing in place (denoted by 0). Also, every person is carrying some amount of money. Each person has 2 attributes, first, direction and second, the amount. At the end of the street is a police officer who, with the right amount of money, can use magic to save people's lives.

For each interaction, when people are travelling in opposite direction and they meet, they both have to submit all the money they are carrying to the police officer. If the moving person interacts with stationary person, then, only the person who is moving have to submit all the money and stationary person won't give any money. Once the interaction happens, both the people involved in the interaction die and stay at the same place and act as a single stationary person.

The policeman looks at you and asks whether you can estimate how much money he will receive, when everyone leaves the street or dies due to interaction.

Examples

Example 1:

```
Input: people = [[1,3] , [-1,10], [1,4] , [0,7] , [-1, 12] , [-1,6]]
Output: 35
```

Explanation: First and second person will meet and submit their money and will act as a stationary person now - Current amount = 3 + 10 = 13.Third person who is moving will meet 4th one who is stationary and will submit the money and now both of them act as a single stationary person now

- Current amount = 13 + 4 = 17. 5th and 6th person will meet the 4th person who is stationary - Current amount = 17 + 12 + 6 = 35

Example 2:

```
Input: people = [[1,3] , [1,10], [1,4] , [1,7] , [1, 12] , [1,6]]
Output: 0
```

Explanation: All people are travelling in same direction, and no one will meet.

Function:

```
def amountPoliceGets(people):
  # Return the amount of money the police officer can collect
  return 0
```

Question 2

Alice and Bob are discussing the implementation of an efficient lookup dictionary. Alice came up with an idea of usage of skip lists. Help Alice and Bob implement a lookup dictionary using skip list.

Following are the instructions for SkipList:

Constraints

- Class should have the methods insert and search.
- `insert` method inserts the number into the skiplist.
- `search` method checks the presence of a target in the skiplist and returns `True` if it is present and returns `False` otherwise.
- All the methods should be implemented in `O(logn)` time complexity.

Note: Class Node is already written for you and will be useful in creating your skip list.

Example:

```
sl = SkipList()
sl.insert(1) # None
sl.insert(2) # None
sl.insert(3) # None
print(sl.search(4)) # False
sl.insert(4) # None
print(sl.search(4)) # True
print(sl.search(1)) # True
```

Function

```
class Node:
  def __init__(self, val):
      self.val = val
      self.next = None
      self.down = None
```

```python
class SkipList:
    def __init__(self):
        # Any variables for intialization

    def search(self, target: int) -> bool:
        # Complete this function
        # Returns True if the element is present in skip list else False
        return False

    def insert(self, num: int) -> None:
        # Complete this function
        # Inserts the element into the skip list
        return None
```

Question 3

The theatre company "The Grand Playhouse" has hired you as a seasoned software engineer to tackle a critical issue with their live production server. They've encountered severe performance bottlenecks with their current Queue class, leading to delays in handling audience requests. The problem lies in the existing Queue class, which exhibits either sluggish `enque()` or `deque()` operations. To ensure a seamless theater experience, you've been entrusted with the formidable task of creating a new Queue class that can perform both `enque()` and `deque()` operations in constant time, O(1).

Constraints:

- The queue will always be of size DEFAULT_SIZE
- The enque() function return a boolean value, True on successful insertion and False on unsuccessful insertion
- The deque() function will return the next value in the queue, if the queue is empty it will return -1
- Ensure that both `enque()` and `deque()` operation maintains a time complexity of `O(1)`.

Examples

Example 1:

```
Input:
queue = Queue()
queue.enque(1)
queue.enque(2)
queue.enque(3)

Output:
[True, True, True]
```

Example 2:

```
Input:
```

```
queue = Queue()
queue.deque()
queue.enque(1)
queue.enque(2)
queue.enque(3)
queue.deque()
queue.deque()
queue.deque()
queue.deque()

Output: [-1, True, True, True, 1, 2, 3, -1]
```

Instructions for Submission:

Please ensure that your code submission includes the following three functions within the `Queue` class: `__init__()`, `enque()`, and `deque()`. You are allowed to use additional functions if necessary for the implementation, but inbuilt libraries should be prohibited.

Function

```
class Queue:

DEFAULT_SIZE = 10

# Initialization step
def __init__(self):
  pass

# Implement the enque() function to insert values into the queue.
# It should return True on successful insertion and False on unsuccessful insertion.
def enque(self, value):
  pass

# Implement the deque() function to retrieve values from the queue.
# It should return the next value in the queue; if the queue is empty, return -1.
def deque(self):
  pass
```

Question 4

There are several distinct colored trays piled(vertically) on a counter in a restaurant. The manager wants the trays arranged in a specific order for aesthetic purposes. The counter space available to you is really small, and it is constrained that you can at a maximum form one intermediate pile before creating the final pile in the specified color order. The catch? Once you move a tray out of the initial pile, you cannot move it back to the initial pile; and once you move a tray into the final pile, you cannot move it back to the intermediate pile. You need to let the manager know if the required order of trays is possible to make or not.

Constraints

- Expected time complexity: O(n)

Examples

Example 1

```
Input:
Initial =  ['Red', 'Blue', 'Green', 'Yellow', 'Orange']
Final = ['Yellow', 'Orange', 'Green', 'Blue', 'Red']

Output: True
```

Explanation: Move Red, followed by Blue and Green, to the final pile. Move Yellow to the intermediate pile, Orange to the final pile, and lastly Yellow to the final pile. Now the initial and intermediate piles and empty, and the final pile is in the required order.

Example 2

```
Input:
Initial = ['Red', 'Blue', 'Green', 'Yellow', 'Orange']
Final = ['Yellow', 'Green', 'Orange', 'Red', 'Blue']

Output: False
```

Explanation: Red is top plate in the initial pile. Move Red to the intermediate pile, Blue to the final pile, and Red to the final pile. Now move Green followed by Yellow to the intermediate pile, and Orange to the final pile. We find that Yellow cannot come before Green if these two are transferred to the final pile directly. Since we cannot transfer any trays into the initial pile (or out of the final pile), we are left with no valid moves and hence cannot obtain the required order.

Function

```python
def isItPossible(initial: list[str], final: list[str]) -> bool:
    # Your code here
    return ans
```