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 - Temple Challange

In a hidden jungle temple, you discover a sacred altar, guarded by ancient spirits. To unlock the treasure hidden within, you must solve a puzzle using a stack of numbered stones.

The temple provides you with a stream of stones numbered from 1 to n. Your goal is to arrange these stones on the altar to match a specific sacred sequence (the target array). You can perform two actions: Push and Pop

You must push or pop stones until the stack matches the sacred sequence exactly. If the stack matches the sequence at any point, you stop, and the treasure is yours.

Your task is to determine the exact sequence of push and pop operations needed to arrange the stones and unlock the temple's treasure.

Constraints

- 1 <= target.length <= 100
- 1 <= n <= 100
- 1 <= target[i] <= n
- target is strictly increasing.

Example 1

```
Input: target = [1,3], n = 3
Output: ["Push","Push","Pop","Push"]
Explanation:
Initially the stack s is empty. The last element is the top of the stack.
- Read 1 from the stream and push it to the stack. s = [1].
- Read 2 from the stream and push it to the stack. s = [1,2].
- Pop the integer on the top of the stack. s = [1].
- Read 3 from the stream and push it to the stack. s = [1,3].
```

Example 2

```
Input: target = [1,2,3], n = 3
Output: ["Push","Push","Push"]
Explanation:
Initially the stack s is empty. The last element is the top of the stack.
- Read 1 from the stream and push it to the stack. s = [1].
- Read 2 from the stream and push it to the stack. s = [1,2].
- Read 3 from the stream and push it to the stack. s = [1,2,3].
```

Example 3

```
Input: target = [1,2], n = 4
Output: ["Push","Push"]
Explanation:
Initially the stack s is empty. The last element is the top of the stack.
- Read 1 from the stream and push it to the stack. s = [1].
- Read 2 from the stream and push it to the stack. s = [1,2].
- Since the stack (from the bottom to the top) is equal to target, we stop
the stack operations.
The answers that read integer 3 from the stream are not accepted.
```

Function

```
def treasure_seq(target : List, n : int)) -> List:
    # Your code goes here.
```

Question 2 - Conveyor Belt

Imagine you're working at a high-tech factory where products are placed on a conveyor belt. Each product arrives one by one and is lined up to be processed. However, the factory needs to be highly efficient, so the speed of adding products to the belt and removing them for processing is critical. Your job is to implement the conveyor belt system as a Queue. The conveyor belt will have the following properties: 1. Products are added to the end of the belt (this is called enqueue). 2. Products are removed from the front of the belt for processing (this is called dequeue). 3. The factory manager has strict requirements:

a. Adding a product to the conveyor belt (enqueue) must take O(1) time.
b. Removing a product from the front (dequeue) must also take O(1) time. You decide to design the conveyor belt using a linked list to meet these requirements.

Constraints

- Enqueue and Dequeue operations must exeeute in O(1) time.

Example 1:

```
Input: [["enqueue", 1], ["enqueue", 2], ["enqueue", 3], ["dequeue"],
["enqueue", 4], ["dequeue"],  ["dequeue"], ["enqueue", 6]]
```

```
Output:
Explanation:
Enqueue 1 to the queue. ([1])
Enqueue 2 to the queue. ([1,2])
Enqueue 3 to the queue. ([1,2,3])
Dequeue                 ([2,3])
Enqueue 4 to the queue. ([2,3,4])
Dequeue                 ([3,4])
Dequeue                 ([4])
Enqueue 6 to the queue. ([4,6])
```

Example 2:

```
Input: [["enqueue", 1],["dequeue"], ["dequeue"], ["enqueue", 6],["dequeue"]]

Output:
Explanation:
Enqueue 1 to the queue. ([1])
Dequeue                 ([])
Dequeue                 ([])
Enqueue 6 to the queue  ([6])
Dequeue                 ([])
```

Function

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

class Queue_LL:
    def __init__(self):
        self.head = None
        self.tail = None


    def enqueue(self, val:int)->None:
        pass


    def dequeue(self)->int:
        pass

    # Convert the Queue into an Array and return it
    def QueueToList(self)->list[int]:
        pass
```

Question 3 - Mission Briefing for a Substitute Soul Reaper

As a Soul Reaper in training, you are tasked with solving spiritual arithmetic using a mystical array of incantations known as **Reverse Polish Notation** (RPN). These incantations represent the spiritual energy flow during a battle and must be evaluated correctly to harness the full power of your Zanpakutō.

You are given an array of **spiritual tokens** that represents an arithmetic expression in this mystical notation. Your mission is to evaluate the expression and return the final spiritual power as an integer.

Key Details:

- The valid spiritual operators are:
  - '+' : Releases spiritual pressure to combine two energies.
  - '-' : Subtracts the opposing force from your spiritual energy.
  - '*' : Amplifies your spiritual power by multiplying energies.
  - '/' : Divides and balances spiritual energy, always truncating towards zero.
- Each operand may be a pure spiritual power (integer) or another expression of energy.
- The division between two energies will always truncate toward zero, no need to worry about dividing by zero (your Reiatsu will prevent that!).
- The input will always represent a valid spiritual arithmetic expression in Reverse Polish Notation.

As a Soul Reaper, ensure all your calculations stay within the bounds of a **32-bit spiritual integer** .

Return the final spiritual power after evaluating the entire expression. Only then will you be able to continue your journey towards becoming a powerful Soul Reaper!

Constraints

```
- 1 <= tokens.length <= 10^4
- tokens[i] is either an operator: "+", "-", "*", or "/", or an integer in the range
[-200, 200].
```

Example 1:

```
Input: incantations = ["2","1","+","3","*"]
Output: 9
Explanation:
((2 + 1) * 3) = 9
The input tokens = ["2", "1", "+", "3", "*"] is evaluated in Reverse Polish Notation:
- Start with the numbers 2 and 1.
- Encounter +: Perform 2 + 1 = 3.
- Now the stack has 3 and 3.
- Encounter *: Multiply 3 * 3 = 9.
Thus, the final result of the expression is 9.
```

Example 2:

```
Input: incantations = ["4","13","5","/","+"]
```

```
Output:6
Explanation:
(4 + (13 / 5)) = 6
The input tokens = ["4", "13", "5", "/", "+"] is evaluated in Reverse Polish Notation:

- Start with the numbers 4, 13, and 5.
- Encounter / (division): Perform 13 / 5 = 2 (integer division truncates toward zero).
- Now the stack has 4 and 2.
- Encounter +: Add 4 + 2 = 6.
```

Example 3:

```
Input: incantations = ["10","6","9","3","+","-11","*","/","*","17","+","5","+"]
Output: 22
Explanation:
((10 * (6 / ((9 + 3) * -11))) + 17) + 5
= ((10 * (6 / (12 * -11))) + 17) + 5
= ((10 * (6 / -132)) + 17) + 5
= ((10 * 0) + 17) + 5
= (0 + 17) + 5
= 17 + 5
= 22

- Start with 9 and 3.
- Encounter +: Perform 9 + 3 = 12.
- Next, multiply: 12 * -11 = -132.
- Now divide: 6 / -132 = 0 (integer division truncates toward zero).
- Multiply: 10 * 0 = 0.
- Add 17: 0 + 17 = 17.
- Finally, add 5: 17 + 5 = 22.
```

Function

```python
def spiritualArithmetic(incantation: list[str]) -> int:
    return finalAns
```

Question 4 - Largest Continuous Block Area

Imagine you have a sequence of positive integers arranged in a line. Each integer represents the height of a stack of blocks, and each stack is exactly one unit wide. Your task is to determine the largest area you can cover by selecting one or more consecutive stacks and forming a rectangle whose height is equal to the smallest stack in that group.

- You are given an array blocks where blocks[i] represents the height of the i-th stack.
- The width of each stack is 1 unit.

Calculate the maximum possible area of a rectangle that can be formed by choosing a contiguous segment of the stacks and using the shortest stack in that segment as the height of the rectangle.

Example-1

```
Input: blocks = [1, 3, 2, 5, 4]
Output: 8
Explanation: Several configurations are possible, but the best is formed by
either the last two blocks (5, 4), with the smallest height being 4, and so
the area is 4 * 2 = 8, or by using blocks two to five, with the smallest
height being 2, and so the area is 2 * 4 = 8.
```

Example-2

```
Input: blocks = [3, 4, 5, 3, 5]
Output: 15
Explanation: The largest rectangle is formed using all the five blocks, where
the smallest height is 3, covering five blocks, and resulting in an area of 3 * 5 = 15
```

Constraints

- 1 <= blocks.length <= 10^3
- 0 <= blocks[i] <= 10^4

Function Signature

```python
def largestArea(blocks: list[int]) -> int:
    pass
```

Question 5 - The Line Up Shuffle

Imagine you're the organizer of a special talent show where several performers are waiting backstage. Each performer has a unique number that represents the order they need to reveal themselves on stage. However, the backstage area is small, and there's only enough space for them to line up in a single file, one behind the other.

Your job is to arrange the performers backstage in such a way that when they step onto the stage, they reveal themselves in ascending order of their numbers, but there's a catch:

1. The first performer in line will immediately step out and reveal their number.
2. After that, if any performers remain backstage, the next one in line will be moved to the very back of the queue without revealing their number.
3. The process continues, with the first in line always stepping out to reveal themselves, and the second being sent to the back, until everyone has had their turn.

Given `nums` you need to decide how to arrange the performers backstage so that when they follow this pattern, they reveal their numbers in order, starting from the lowest and ending with the highest i.e. in ascending order.

Constraints

1. `1 <= nums.length <= 1000`
2. `1 <= nums[i] <= 10<sup>6</sup>`
3. All the values of `nums` are unique

Example:

Input: `nums = [17, 13, 11, 2, 3, 5, 7]`
Output: [2, 13, 3, 11, 5, 17, 7]
Explanation:
We get the deck in the order [17,13,11,2,3,5,7] (this order does not matter), and reorder it.
After reordering, the deck starts as [2,13,3,11,5,17,7], where 2 is the top of the deck.
We reveal 2, and move 13 to the bottom. The deck is now [3,11,5,17,7,13].
We reveal 3, and move 11 to the bottom. The deck is now [5,17,7,13,11].
We reveal 5, and move 17 to the bottom. The deck is now [7,13,11,17].
We reveal 7, and move 13 to the bottom. The deck is now [11,17,13].
We reveal 11, and move 17 to the bottom. The deck is now [13,17].
We reveal 13, and move 17 to the bottom. The deck is now [17].
We reveal 17.
Since all the cards revealed are in increasing order, the answer is correct.

Input: `nums = [1000, 1]`
Output: [1, 1000]

Function

```
def arrangePerformers(nums):
    return res
```

Question 6 - Walter White's Quest

After parting ways with Walter White, Jessie Pinkman has decided to delve deeper into the world of chemistry on his own. He has stacks of chemical formulas that he wants to analyze, counting how many atoms of each element are present in each formula. He needs to write a program to calculate the number of each type of atom in these formulas. The formulas can contain nested groups of atoms enclosed in parentheses, followed by a multiplier indicating how many times the group should be counted. Your task is to help Jessie by writing a function 'atom_counter(formula: str) -> str' , that takes a chemical formula as a string and returns a string representing the count of each atom, sorted alphabetically by atom name.

Constraints

- 1 <= formula.length <= 1000
- formula consists of English letters, digits, '(', and ')'.
- formula is always valid.

Hint: Use a stack to handle the nested structure of groups within parentheses. As you parse the formula, use the stack to keep track of counts, and multiply counts as necessary when closing groups.

Example 1:

Input:H2O

Output: H2O

Explanation:The count of elements are {'H': 2, 'O': 1}

Example 2:

Input:Mg(OH)2

Output: H2MgO2

Explanation:The count of elements are {'H': 2, 'Mg': 1, 'O': 2}

Function

```
# Function description goes here
def atom_counter(formula: str) -> str:
    #Your code goes here.
```

Question 7 - Managing Customer Flow at a Supermarket Checkout Counter

You are managing a shared checkout counter at a supermarket. There are `n` customers numbered from `0` to `n - 1`, each arriving at the checkout counter at different times. Each customer either wants to pay for their items or return an item.

You are given a non-decreasing integer array `arrival` of size `n`, where `arrival[i]` represents the arrival time of the `i`th customer at the checkout counter. You are also given an array `action` of size `n`, where `action[i]` is `0` if customer `i` wants to pay for their items (checkout), and `1` if customer `i` wants to return an item.

If two or more customers are present at the counter at the same time, they follow these rules when using the checkout counter:

1. If the counter was not used in the previous second, then the customer who wants to return an item goes first.
2. If the counter was used in the previous second for checking out, the next customer who wants to checkout goes first.
3. If the counter was used in the previous second for returning, the next customer who wants to return an item goes first.
4. If multiple customers want to perform the same action, the one with the smallest index goes first.

Return an array `result` of size `n` where `result[i]` is the second at which the `i`th customer completes their checkout or return process.

Note that:

- Only one customer can use the checkout counter at any second.
- A customer may arrive at the counter and wait without performing their action to follow the mentioned rules.

Constraints

- n == arrival.length == action.length

- 1 <= n <= 10^5
- 0 <= arrival[i] <= n
- arrival is sorted in non-decreasing order.
- action[i] is either 0 or 1.

Example 1:

```
Input: arrival = [0,1,1,2,4], action = [0,1,0,0,1]
Output: [0,3,1,2,4]
Explaination: At each second we have the following:
- At t = 0: Person 0 is the only one who wants to pay, so
they just pay at the
checkout counter and leave.
- At t = 1: Person 1 wants to return an item, and person 2 wants
to checkout.
Since the counter was used the previous second for checking out,
person 2 pays and leaves.
- At t = 2: Person 1 still wants to return the item,
and person 3 wants to checkout. Since the counter
was used the previous second for checking out,
person 3 pays and leaves.
- At t = 3: Person 1 is the only one at the counter wanting
to return the item, so they just return the item at the
checkout counter and leave.
- At t = 4: Person 4 is the only one who wants to return
an item, so they just return the item at the counter and leave.
```

Example 2:

```
Input: arrival = [0,0,0], action = [1,0,1]
Output: [0,2,1]
Explaination: At each second we have the following:
- At t = 0: Person 1 wants to checkout while persons 0 and
2 want to return
their items. Since the counter was not used in the previous
second, the persons
who want to return get to use the counter first. Since person 0
has a smaller index, they return first.
- At t = 1: Person 1 wants to checkout, and person 2 wants to
return. Since the
counter was used in the previous second for returning, person 2
returns their product and leaves.
- At t = 2: Person 1 is the only one present at the counter
wanting to checkout,
so they just checkout at the counter and leave.
```

Function

```
def checkout_counter_times(arrival, action):
    # Function implementation here
```

Question-8 - Quarry Pie II

John, after designing a storage management system, takes a good night sleep with satisfaction, after a very long time. The next morning, he is assigned an opening shift at IMU's Quarry Pie. Initially, it's a slow day, but he learns that a major event is scheduled for the afternoon, which will bring a large crowd. To accommodate the rush, a new pizza stall, Quarry Pie II, is set up at the opposite end of the hall by the Assistant Managers, enabling customers to join either end. Once someone joins the line, they cannot switch the positions due to supervision.

John had promised free pizza slices to his friends, and they texted him when they joined the line. Since he cannot frequently check his phone during his shift, he needs to rely on the stall's registers to verify that his friends receive their slices.

So, John decides to design a new data structure, QuarryPieLine, to manage this. The data structure will allow the tracking of customers to join the line from the front, back, or middle, and also allow John to remove customers, once they receive the pizza or leave the line due to a long waiting time, and also check if his friends are still in line (this is kept secret). Additionally, Jack, John's coworker and friend, challenges him to implement this data structure using only stacks.

John who loves challenges, accepts it. Now he has three hours before the crowd arrives at noon. Can you help John design the QuarryPieLine data structure using only the stack data structure?

Task: Create a class quarryPieLine which has the methods, joinInFront, joinInMiddle, joinInBack, removeFromFront, removeFromMiddle, removeFromBack, whoIsFront, whoIsMiddle, whoIsBack

joinInFront, joinInMiddle, joinInBack - Exactly one Person can join at a time in the front, middle, back removeFromFront, removeFromBack, removeFromMiddle - Exactly one Person is removed respectively whoIsFront, whoIsBack, whoIsMiddle - returns the person id present at the position respectively (returns -1 if the quarryPieLine is empty)

Note: Use minimal number of stacks and No inbuilt data structures are to be used. Anything created should be from scratch (apart from stack).
Create an optimal time complex solution for this. If there are n people in the line currently, joining in middle means:
n is even, join between n/2 and n/2 + 1,
n is odd, join between floor(n/2) and floor(n/2) + 1
Same works for removing from the middle.
Person's id is a counter which tells you when the customer has exactly joined the line

For simplicity of the test cases all these methods are numbered from 1 to 9 starting from joinInFront, joinInMiddle, joinInBack, removeFromFront, removeFromMiddle, removeFromBack, whoIsFront, whoIsMiddle, whoIsBack respectively.

A maximum of 10000 calls will be made including all the methods

Example:

```
Input: [1, 2, 2, 3, 5, 6, 7, 8, 9]
```

```
Output: [2, 2, 1]
Explanation:
Let maintain the person id with counter (initialized to 0)
1. The first operation (1) is joinInFront, counter = 1
(incremented to 1 as we are joining the first person in line). Line = [1]
2. The second operation (2) is joinInMiddle, counter = 2
( second person is joining the line). Line = [2, 1] (as mentioned in the note)
3. The third operation (2) is joinInMiddle, counter = 3
(third person is joining the line). Line = [2, 3, 1]
4. The fourth operation (3) is joinInBack, counter = 4
(fourth person is joining the line). Line = [2, 3, 1, 4]
5. The fifth operation (5) is removeFromMiddle, No one is joining
(hence counter remains same). Line = [2, 1, 4]
6. The sixth operation (6) is removeFromBack, No one is joining
(hence counter remains same). Line = [2, 1]
7. The seventh operation (7) is whoIsFront, the resulting array is [2].
 Line = [2, 1] (appended 2 to array since 2 is at the front of line)
8. The eight operation (8) is whoIsMiddle, the resulting array is [2, 2].
Line = [2, 1] (appended 2 to array since 2 is at the middle of line)
9. The nineth operation (9) is whoIsBack, the resulting array is [2, 2, 1].
Line = [2, 1] (appended 1 to array since 1 is at the back of line)
```

Function

```python
class quarryPieLine:

    def __init__(self):
        pass

    def joinInFront(self):
        pass

    def joinInMiddle(self):
        pass

    def joinInBack(self):
        pass

    def removeFromFront(self):
        pass

    def removeFromMiddle(self):
        pass

    def removeFromBack(self):
        pass

    def whoIsFront(self) -> int:
```

```
        pass

    def whoIsMiddle(self) -> int:
        pass

    def whoIsBack(self) -> int:
        pass
```

Question 9 - Amortized Dictionary

Using Python Dictionary, implement Amortized Dictionary as a Python class, where the dictionary keys represent the levels and values are the elements of the corresponding level. At each level i, there can be either 2^i elements or no elements at all. Any level that contains elements, has them in a sorted order. Implement the below methods for this class: - Insert method inserts an element into the amortoized dictionary. - Search method searches for the element, and returns the level if the element exists in the amortized dictionary, else returns -1. - Print method returns a list consisting of list of elements at each level. - All the above methods should be implemented as per the time complexity discussed in class. No in built sort functions should be used.

Example:

```
ad = amor_dict([23, 12 ,24, 42])
print(ad.print())
# [[], [], [12, 23, 24, 42]]
ad.insert(11)
print(ad.print())
# [[11], [], [12, 23, 24, 42]]
ad.insert(74)
print(ad.print())
# [[], [11, 74], [12, 23, 24, 42]]
print(ad.search(74))
# 1
print(ad.search(77))
# -1
```

```
ad = amor_dict([1, 5, 2, 7, 8, 4, 3])
print(ad.print())
# [[3], [4, 8], [1, 2, 5, 7]]
print(ad.search(1))
# 2
ad.insert(11)
print(ad.print())
# [[], [], [], [1, 2, 3, 4, 5, 7, 8, 11]]
print(ad.search(1))
# 3
```

Function

```
class amor_dict():
  def __init__(self, num_list = []):
    # your code here
    pass
  def insert(self, num):
    # your code here
    pass
  def search(self, num):
    # your code here
    pass

  # Returns a list of all levels under a list
  # ex [[x],[t,y]...]
  def print(self):
    # your code here
    pass
```

Question 10 - SkipList

Charlie and Dana are brainstorming how to implement an efficient lookup dictionary. Dana suggests using skip lists for the task. Assist Charlie and Dana in creating a lookup dictionary based on skip lists.

Constraints

- Implement the Skip List using the provided class signature. Ensure the class includes the methods `insert` and `search`.
- `insert` and `search` methods should be implemented in `O(logn)` time complexity.

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
```

Class Signature

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

```python
class SkipList:

    # Any variables for intialization
    def __init__(self):
        pass

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

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