Editorial : Stack Assignment

Question : Reduce String

```
def solve(s):
    stk = []
    for char in s:
        if len(stk) == 0:
            stk.append(char)
        elif char == stk[-1]:
            stk.pop()
        else:
            stk.append(char)
    if len(stk):
        print("".join(stk))
    else:
        print("Empty String")

def inp():
    s = input()
    solve(s)

inp()
```

Explanation

1. **Function Definition:** `solve(s)`
    - The function takes a string `s` as input.
2. **Using a Stack (** `stk` **)**

    ```
    stk = []
    ```

    - We use a list `stk` as a stack to build the resulting string.
3. **Iterating Through the Characters**

```
for char in s:
    if len(stk) == 0:
        stk.append(char)
    elif char == stk[-1]:
        stk.pop()
    else:
        stk.append(char)
```

- **Case 1**: If `stk` is empty, we push the current character onto `stk`.
- **Case 2**: If the current character `char` matches the top of the stack (`stk[-1]`), it means we found a pair of adjacent identical characters. We pop the top character from the stack to remove the matching pair.
- **Case 3**: If `char` does not match `stk[-1]`, we push `char` onto the stack.

> **Purpose**:
> This effectively removes pairs of consecutive identical characters. Each time we find two identical letters in a row, we eliminate them.

4. **Checking the Result**

```
if len(stk):
    print("".join(stk))
else:
    print("Empty String")
```

- After processing all characters:
  - If the stack is **not empty**, we join all characters in `stk` into a string and print it. This is the final reduced string.
  - If the stack is **empty**, it means all characters were removed in pairs, so we print `"Empty String"`.

5. **Input Function `inp()`**

```
def inp():
    s = input()
    solve(s)
```

- Reads a string `s` from standard input.
- Calls `solve(s)` to perform the reduction logic.

6. **Calling `inp()`**

```
inp()
```

- Starts the entire process by reading input and printing the final reduced result.

How It Works Overall

- Each time you add a character to the stack, you check if it cancels out the top of the stack (i.e., if they are the same).
- If they match, you remove the top character instead of adding the new one, effectively removing both characters from consideration.
- In the end, the stack contains only those characters that **could not** be paired off.
- If the stack is empty, it means every character eventually found a matching pair, leaving no remaining characters.

Question : Push, Pop and Top

```
class Stack:
    def __init__(self):
        self.arr = []

    def push(self, item):
        self.arr.append(item)

    def pop(self):
        if len(self.arr) == 0:
            return "Underflow!"
        return self.arr.pop()

    def top(self):
        if len(self.arr) == 0:
            return "Empty!"
        return self.arr[-1]


def solve():
    N = int(input())
    stk = Stack()
    for i in range(N):
        query = input().split()
        if query[0] == "1":
            # Push operation
```

```
            stk.push(query[1])
        elif query[0] == "2":
            # Pop operation
            stk.pop()
        else:
            # Top operation
            ans = stk.top()
            print(ans)

solve()
```

Explanation

1. **Stack Class Definition**

```
class Stack:
    def __init__(self):
        self.arr = []
```

   - We define a `Stack` class that internally uses a Python list `self.arr` to store elements.

2. **Push Method**

```
def push(self, item):
    self.arr.append(item)
```

   - Appends `item` to the end of the list `self.arr`.
   - In stack terminology, this places `item` on top of the stack.

3. **Pop Method**

```
def pop(self):
    if len(self.arr) == 0:
        return "Underflow!"
    return self.arr.pop()
```

   - If `self.arr` is empty, it returns `"Underflow!"` to indicate there's nothing to pop.
   - Otherwise, it removes and returns the last element in `self.arr` (the top of the stack).

4. **Top Method**

```
def top(self):
    if len(self.arr) == 0:
        return "Empty!"
    return self.arr[-1]
```

- If `self.arr` is empty, returns `"Empty!"`.
- Otherwise, returns the last element in `self.arr` (the top element), without removing it.

5. **`solve()` Function**

```
def solve():
    N = int(input())
    stk = Stack()
    for i in range(N):
        query = input().split()
        if query[0] == "1":
            stk.push(query[1])
        elif query[0] == "2":
            stk.pop()
        else:
            ans = stk.top()
            print(ans)
```

- Reads an integer `N`, which specifies the number of operations.
- Creates a `Stack` instance `stk`.
- Loops `N` times, each time reading a query.
  - If the query starts with `"1"`, it's followed by a value to **push** onto the stack.
  - If the query starts with `"2"`, we **pop** from the stack (discard the returned value, unless it's `"Underflow!"` which is not explicitly printed here).
  - Otherwise (query starts with `"3"`), we call `top()` and **print** the result.

How It Works Overall

- The code implements a simple stack with push, pop, and top operations:
  i. **Push** (`1 X`): Places the element `X` on top of the stack.
  ii. **Pop** (`2`): Removes the top element of the stack (if any).
  iii. **Top** (`3`): Prints the top element of the stack, or `"Empty!"` if the stack is empty.
- If a **pop** is attempted on an empty stack, the `Stack` class returns `"Underflow!"`, but the code does not print this message—it simply ignores it.
- This code handles `N` queries in total, printing the top element each time query `3` is encountered.

Question : Again a classical problem

```
def solve(s):
    dic = {
        ')': '(',
        ']': '[',
        '}': '{'
    }
    stk = []
    for char in s:
        if char == "(" or char == "[" or char == "{":
            stk.append(char)
        else:
            if len(stk):
                # Notice this condition checks the same thing thrice; it's redundant but kept as
                if dic[char] == stk[-1] or dic[char] == stk[-1] or dic[char] == stk[-1]:
                    stk.pop()
            else:
                return "not balanced"
    if len(stk):
        return "not balanced"
    return "balanced"


def inp():
    N = int(input())
    for i in range(N):
        s = input()
        ans = solve(s)
        print(ans)

inp()
```

Explanation

1. **Dictionary of Matching Brackets**

    ```
    dic = {
        ')': '(',
        ']': '[',
        '}': '{'
    }
    ```

- This dictionary maps each **closing** bracket to its corresponding **opening** bracket.

2. **Stack Initialization**

```
stk = []
```

   - We use a list called `stk` as a stack to keep track of opening brackets.

3. **Iterating Through Each Character**

```
for char in s:
    if char == "(" or char == "[" or char == "{":
        stk.append(char)
    else:
        ...
```

   - For each character in the string `s`, we check if it is an **opening** bracket (`(`, `[`, `{`). If it is, we push it onto the stack.

4. **Handling Closing Brackets**

```
else:
    if len(stk):
        if dic[char] == stk[-1] or dic[char] == stk[-1] or dic[char] == stk[-1]:
            stk.pop()
    else:
        return "not balanced"
```

   - If the character is a **closing** bracket:
     - We first check if the stack is **not empty** (i.e., `len(stk) != 0`).
     - Then we compare the top of the stack (`stk[-1]`) with the expected matching bracket (`dic[char]`).
     - If they match, we pop from the stack (removing the matched opening bracket).
     - If they do **not** match (or the stack was empty), we return `"not balanced"`.

   Note: The condition `if dic[char] == stk[-1] or dic[char] == stk[-1] or dic[char] == stk[-1]:` is effectively the same check repeated three times. It is redundant, but we are keeping it exactly as in the provided code.

5. **Final Stack Check**

```
if len(stk):
    return "not balanced"
return "balanced"
```

- After processing all characters, if the stack is **empty**, it means every opening bracket had a matching closing bracket in the correct order, so the string is `"balanced"`.
- If the stack is **not empty**, it means there are unmatched opening brackets left, so the string is `"not balanced"`.

6. `inp()` **Function**

```
def inp():
    N = int(input())
    for i in range(N):
        s = input()
        ans = solve(s)
        print(ans)
```

- Reads the number of test cases `N`.
- For each test case, reads a string `s`, calls `solve(s)`, and prints the result (`"balanced"` or `"not balanced"`).

7. **Main Call**

```
inp()
```

- This line starts the process by calling `inp()`, which handles input and output for multiple test cases.

How It Works Overall:

1. For each test case (string of brackets), the program uses a stack to match every closing bracket with its corresponding opening bracket.
2. If at any point there is a mismatch or the stack is empty when expecting a match, the string is considered `"not balanced"`.
3. After processing the entire string, if the stack is empty, the string is `"balanced"`; otherwise, `"not balanced"`.

Question : Smaller neighbour element

```
N = int(input())
arr = list(map(int, input().split()))

stack = []
ans = []
for i in range(N):
    while len(stack) != 0 and stack[-1] >= arr[i]:
        stack.pop()
    if len(stack):
        ans.append(stack[-1])
    else:
        ans.append(-1)
    stack.append(arr[i])

print(*ans)
```

Explanation

1. **Reading Input**

```
N = int(input())
arr = list(map(int, input().split()))
```

   - We read an integer `N`, the number of elements in the array.
   - We then read `N` integers from the next line into the list `arr`.

2. **Initializing Data Structures**

```
stack = []
ans = []
```

   - `stack` will be used to keep track of the elements we've seen, in a manner that helps find the previous smaller element.
   - `ans` will hold the final answers, one for each element in `arr`.

3. **Iterating Through Each Element**

```
for i in range(N):
    while len(stack) != 0 and stack[-1] >= arr[i]:
```

```
        stack.pop()
    if len(stack):
        ans.append(stack[-1])
    else:
        ans.append(-1)
    stack.append(arr[i])
```

- We process each element `arr[i]` one by one from left to right.
- **Inner While Loop**:

  ```
  while len(stack) != 0 and stack[-1] >= arr[i]:
      stack.pop()
  ```

  - While the stack is not empty, and the top of the stack (`stack[-1]`) is **greater than or equal** to the current element `arr[i]`, we pop from the stack.
  - This ensures that any element larger than or equal to `arr[i]` is not useful as a "smaller neighbor," so we remove it.
- **Check the Top of the Stack**:

  ```
  if len(stack):
      ans.append(stack[-1])
  else:
      ans.append(-1)
  ```

  - After removing all elements that are not smaller than `arr[i]`, if the stack is still not empty, its top element is the **nearest smaller element** on the left.
  - If the stack is empty, it means no smaller element exists on the left, so we append `-1`.
- **Push Current Element**:

  ```
  stack.append(arr[i])
  ```

  - We then push the current element onto the stack, making it available to be a "previous smaller" for future elements.

4. **Printing the Result**

```
print(*ans)
```

- Finally, we print all elements in `ans` separated by spaces.
- Each element in `ans` corresponds to the nearest smaller neighbor of the respective element in `arr`. If no such neighbor exists, it's `-1`.

How It Works Overall

- The code uses a **monotonic stack** approach to find the nearest smaller element to the left for each position in the array.
- By popping from the stack all elements greater than or equal to the current element, the top of the stack (if it exists) will be the **closest** smaller element.
- If the stack is empty, we know there is no smaller element on the left.
- This method runs in **O(N)** time because each element is pushed and popped at most once.

Question : Array Split

```python
def count_valid_splits(Arr, n):
    # Calculate the total sum of the array
    total_sum = sum(Arr)
    # Initialize the sum of the left part
    left_sum = 0
    # Initialize the count of valid splits
    count = 0

    # Iterate up to n-1 (i.e., the last index we can split before the end)
    for i in range(n - 1):
        # Add the current element to the left sum
        left_sum += Arr[i]
        # Check if the left sum is at least half the total sum
        if 2 * left_sum >= total_sum:
            count += 1

    return count

# Read the number of test cases
T = int(input())
# Process each test case
for _ in range(T):
    # Read the size of the array
    n = int(input())
    # Read the array elements
    Arr = list(map(int, input().split()))
    # Compute and print the number of valid splits
    print(count_valid_splits(Arr, n))
```

Explanation of the Code

1. **Function Definition:** `count_valid_splits(Arr, n)`
   - **Parameters**:
     - `Arr`: The list of integers (the array).
     - `n`: The size of the array (`len(Arr)`).
   - **Goal**: Determine how many ways we can split the array into two parts—left and right—such that the sum of the left part is at least as large as the sum of the right part.
2. **Total Sum of the Array**

```
total_sum = sum(Arr)
```

   - We compute the sum of all elements in `Arr`. Let's call this `total_sum`.
3. **Initializing Variables**

```
left_sum = 0
count = 0
```

   - `left_sum` will track the cumulative sum of the left part as we iterate through the array.
   - `count` will track how many valid split points we find.
4. **Iterating Over Potential Split Points**

```
for i in range(n - 1):
    left_sum += Arr[i]
    if 2 * left_sum >= total_sum:
        count += 1
```

   - We loop from `i = 0` to `i = n-2` (inclusive). Each `i` represents a potential split point between `Arr[:i+1]` (left) and `Arr[i+1:]` (right).
   - After adding `Arr[i]` to `left_sum`, we check if: $[2 \times \text{left_sum} ; \geq ; \text{total_sum}]$
     - This condition is equivalent to: $[\text{left_sum} ; \geq ; \frac{\text{total_sum}}{2}]$
     - In other words, **the sum of the left part** is at least **half the total sum**, which also means **the left sum** is **at least** the sum of the right part.
   - If this condition is satisfied, we increment `count` by 1.

5. **Returning the Count**

```
return count
```

   - After examining all possible split points, `count` holds the total number of valid splits. We return that value.

6. **Main Program Logic**

```
T = int(input())
for _ in range(T):
    n = int(input())
    Arr = list(map(int, input().split()))
    print(count_valid_splits(Arr, n))
```

   - We read the number of test cases `T`.
   - For each test case:
     - Read `n` (the size of the array).
     - Read the array `Arr`.
     - Call `count_valid_splits(Arr, n)` to compute the number of valid splits.
     - Print the result.

How It Works Overall

- For each test case, the code calculates the total sum of the array.
- It then iterates through each possible split point (from the start up to the second-to-last element) to see if the **left sum** is at least as large as the **right sum**.
- Mathematically, checking (\text{left_sum} \ge \frac{\text{total_sum}}{2}) ensures that the left portion's sum is greater than or equal to the right portion's sum.
- The final output is the count of all such valid split points for each test case.