Module 2 Graded Assignment 2 Editorial

1. Question:

You have an algorithm where the total number of operations grows in direct proportion to n multiplied by n. Which term best describes this growth?

a) Linear time
b) Quadratic time
c) Constant time
d) Exponential time

Correct Answer: b)
Explanation: Quadratic time means the operations increase roughly by n multiplied by n.

2. Question:

Among the following scenarios, which one typically requires more operations than a single pass (O(n)) but fewer operations than an n-squared ($O(n^2)$) approach?

a) Random access in an array
b) Repeatedly cutting the problem size in half (like an efficient sorting method)
c) Reading every single element once
d) A method that tries every possible combination

Correct Answer: b)
Explanation: Approaches that repeatedly cut the problem size (for example, certain sorting methods) typically grow faster than linear but slower than quadratic time.

3. Question:

Consider two loops, one inside the other, both running from 1 to n. How many total loop iterations happen?

```
for i in range(n):
    for j in range(n):
        # Some basic operation
```

a) Roughly n iterations
b) Roughly n multiplied by n iterations
c) Twice as many as n
d) Only 1 iteration overall

Correct Answer: b)
Explanation: Each of the n iterations in the outer loop runs an inner loop of n iterations, resulting in n multiplied by n total iterations.

4. Question:

During bubble sort, each pass moves what element to its correct position in the worst case?

a) The smallest element
b) The largest element among those not placed yet
c) The middle element
d) The element with the highest index

Correct Answer: b)
Explanation: In bubble sort, each pass "bubbles up" the largest unsorted element to the end of the list.

5. Question:

Which principle does a stack follow for adding and removing elements?

a) First In, First Out
b) Last In, Last Out
c) Largest In, First Out
d) Last In, First Out

Correct Answer: d)
Explanation: A stack follows the last in, first out principle, often called LIFO for short.

6. Question:

In Python, which list method is most commonly used to remove the top item of a stack?

```
my_stack = []
# Code to push items onto the stack
# Now remove the top item:
item = my_stack.____()
```

a) insert
b) remove
c) pop
d) clear

Correct Answer: c)
Explanation: The pop method removes the last item added to the list, making it work like a stack.

7. Question:

You want to create a queue by using two stacks, named stack_in and stack_out. How should you remove an element from this queue?

a) Pop directly from stack_in at all times
b) If stack_out is empty, move all elements from stack_in to stack_out, then pop from stack_out
c) Always move elements one by one back and forth for each removal
d) Keep adding to stack_out and never remove from stack_in

Correct Answer: b)
Explanation: To dequeue, transfer all items from stack_in to stack_out only when stack_out is empty, then pop from stack_out. This sequence ensures first in, first out behavior.

8. Question:

In a simple singly linked list, what does each node typically store?

a) A reference to the next node and a reference to the previous node
b) A reference to the next node and the data
c) Only the data
d) A reference to itself and the data

Correct Answer: b)
Explanation: A singly linked list node usually keeps data plus a reference to the next node in the list.

9. Question:

When creating a queue using a linked list, where do you usually remove elements to maintain first in, first out behavior?

a) Remove from the tail of the list
b) Remove from the head of the list
c) Remove from a random position
d) Remove from all positions at once

Correct Answer: b)
Explanation: Items are typically inserted at the tail of the linked list and removed from the head to keep the first in, first out order.

10. Question:

Which important requirement must be satisfied before you can use the standard method that repeatedly cuts your search problem in half each step?

a) The collection must be sorted
b) The collection must be very large
c) The collection must have no repeated elements
d) The collection must be a linked list

Correct Answer: a)
Explanation: The technique that splits the problem space in half, often known as a binary-like search, only works on sorted collections.

Question : Make Leaderboard

```
def bubbleSortOnName(names, scores):
    N = len(names)
    for i in range(N-1):
        for j in range(N-i-1):
            if names[j] > names[j+1]:
                names[j], names[j+1] = names[j+1], names[j]
                scores[j], scores[j+1] = scores[j+1], scores[j]

def bubbleSortOnScore(names, scores):
    N = len(names)
    for i in range(N-1):
        for j in range(N-i-1):
            if scores[j] < scores[j+1]:
                names[j], names[j+1] = names[j+1], names[j]
                scores[j], scores[j+1] = scores[j+1], scores[j]

def printLeaderBoard(names, scores):
    N = len(names)
    rank = 1
    for i in range(N):
        print(rank, names[i])
        if i != N-1 and scores[i] > scores[i+1]:
            rank = i + 2

def solve(names, scores):
    bubbleSortOnName(names, scores)
    bubbleSortOnScore(names, scores)
    printLeaderBoard(names, scores)

def inp():
    N = int(input())
    names = []
    scores = []
    for i in range(N):
        name, score = input().split()
        names.append(name)
        scores.append(int(score))
    solve(names, scores)

inp()
```

Explanation of Each Part

1. `bubbleSortOnName(names, scores)`

```
def bubbleSortOnName(names, scores):
    N = len(names)
    for i in range(N-1):
        for j in range(N-i-1):
            if names[j] > names[j+1]:
                names[j], names[j+1] = names[j+1], names[j]
                scores[j], scores[j+1] = scores[j+1], scores[j]
```

- This function sorts the `names` list in **ascending** alphabetical order using a **bubble sort** approach.
- We perform `(N-1)` passes, and in each pass, we compare adjacent elements `names[j]` and `names[j+1]`.
- If `names[j]` is lexicographically **greater** than `names[j+1]`, we swap both `names` and their corresponding `scores`.
- By the end, `names` will be sorted alphabetically, and `scores` will be reordered accordingly so that each student's score remains aligned with their name.

2. `bubbleSortOnScore(names, scores)`

```
def bubbleSortOnScore(names, scores):
    N = len(names)
    for i in range(N-1):
        for j in range(N-i-1):
            if scores[j] < scores[j+1]:
                names[j], names[j+1] = names[j+1], names[j]
                scores[j], scores[j+1] = scores[j+1], scores[j]
```

- This function sorts the `scores` in **descending** order using another bubble sort pass.
- We again perform `(N-1)` passes, but this time we check if `scores[j] < scores[j+1]`.
- If so, we swap both the scores and the corresponding names to keep them aligned.
- After completion, the list will be ordered such that the highest score is at the front, moving down to the lowest score.

3. `printLeaderBoard(names, scores)`

```
def printLeaderBoard(names, scores):
    N = len(names)
    rank = 1
    for i in range(N):
        print(rank, names[i])
        if i != N-1 and scores[i] > scores[i+1]:
            rank = i + 2
```

- After sorting, this function prints out each student's **rank** and **name**.
- We start with `rank = 1`.
- For each student in the list:
  - We print the current `rank` and the student's `name`.
  - Then we check if the current student's score is greater than the **next** student's score:
    - If `scores[i] > scores[i+1]`, it means the next student has a strictly lower score, so the next student should get a new rank (i.e., `i+2`).
    - Otherwise, if the scores are the same, the next student shares the same rank.
- This logic effectively handles the case where multiple students have the same score (they share the same rank, and the next student with a lower score gets the next rank).

4. `solve(names, scores)`

```
def solve(names, scores):
    bubbleSortOnName(names, scores)
    bubbleSortOnScore(names, scores)
    printLeaderBoard(names, scores)
```

- The `solve` function orchestrates the process:
  i. Sort the list by name (alphabetically).
  ii. Sort by score (descending).
  iii. Print the leaderboard with the correct rank ordering.

5. `inp()` and Main Execution

```
def inp():
    N = int(input())
    names = []
    scores = []
    for i in range(N):
        name, score = input().split()
        names.append(name)
        scores.append(int(score))
    solve(names, scores)

inp()
```

- The `inp()` function:
  - Reads an integer `N` from input (number of students).
  - Initializes empty lists `names` and `scores`.
  - Repeats `N` times:
    - Reads a line, splits it into `name` (string) and `score` (integer).
    - Appends `name` to `names` and `score` to `scores`.
  - Calls the `solve(names, scores)` function to perform the sorting and ranking.
- Finally, we call `inp()` to run the entire process.

How It All Fits Together

1. The user inputs the number of students `N`, followed by `N` lines of name-score pairs.
2. `bubbleSortOnName` arranges students alphabetically, keeping each name-score pair intact.
3. `bubbleSortOnScore` then sorts these pairs by scores in descending order.
4. `printLeaderBoard` prints the rank and name for each student, adjusting the rank properly when scores differ.
5. As a result, you get a final sorted leaderboard with the highest scoring student(s) at the top, ties handled gracefully, and ranks assigned appropriately.

Question : Doctor's Appointment

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

c = 0
for i in range(len(lis) - 1):
    for j in range(len(rand)):
        c += 1
        if lis[i] == rand[j]:
            rand.pop(j)
            break

print(c)
```

Explanation

1. **Reading Inputs**

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

   - `N` is the number of people in the line (or some list of identifiers).
   - `lis` is a list of `N` integers (e.g., the original arrangement of people).
   - `rand` is another list of `N` integers (e.g., a random arrangement or sequence).

2. **Initializing a Counter**

   ```
   c = 0
   ```

   - `c` will keep track of the total number of increments or “moves” made.

3. **Outer Loop**

   ```
   for i in range(len(lis) - 1):
       ...
   ```

   - This loop iterates from `i = 0` to `i = len(lis) - 2`.

- Notably, it does **not** iterate over the very last index of `lis` (`len(lis) - 1`).

4. **Inner Loop**

```
for j in range(len(rand)):
    c += 1
    if lis[i] == rand[j]:
        rand.pop(j)
        break
```

- For each `lis[i]` , we iterate through `rand` from the beginning.
- Each time we check an element in `rand` , we increment `c` by 1.
- If we find a match (`lis[i] == rand[j]`), we remove that element from `rand` (`rand.pop(j)`) and then `break` out of the inner loop.

5. **Print the Counter**

```
print(c)
```

- After processing all elements (except the last one in `lis`), we print the total count `c` .

How It Works Overall

- The code effectively compares each element of `lis` (except the last one) against elements in `rand` .
- **For each element in `lis` ,** it loops through `rand` until it finds a match, incrementing `c` each time it checks an element in `rand` .
- Once it finds a match, it removes that matched element from `rand` and stops checking further for that `lis[i]` .
- The final value of `c` represents the total number of comparisons (or “moves”) made across all iterations.

Question : Solve Age of empires

```
n = int(input()) * 2
arr = list(map(int, input().split()))
arr.sort()
sp = 0
for i in range(n):
    if i % 2 == 0:
        sp += arr[i]
print(sp)
```

Explanation

1. **Reading Input**

   ```
   n = int(input()) * 2
   arr = list(map(int, input().split()))
   ```

   - The first input gives N, which represents the number of **associations** (each association is a pair of workers). Since each association has 2 workers, the total number of workers is $2N$.
   - We then read the next line containing $2N$ integers, which represent the **building speeds** of the workers, and store them in the list `arr`.

2. **Sorting the Speeds**

   ```
   arr.sort()
   ```

   - We sort the list `arr` in **ascending order**. Sorting helps us pair workers in a way that maximizes the sum of the minimum speeds in each pair.

3. **Summing Up Every Second Element**

   ```
   sp = 0
   for i in range(n):
       if i % 2 == 0:
           sp += arr[i]
   ```

   - After sorting, we iterate through the list from `i = 0` to `i = n - 1` (remember `n` here is actually `2N` from the problem statement).
   - We check `if i % 2 == 0`; that means we take every **even-indexed** element in the sorted list.

- We add these elements to `sp`.

Why does this work?

- When we pair the sorted speeds like this:
  - Pair $(arr[0], arr[1])$, the minimum is `arr[0]`.
  - Pair $(arr[2], arr[3])$, the minimum is `arr[2]`.
  - Pair $(arr[4], arr[5])$, the minimum is `arr[4]`, and so on.
- By summing the elements at **even indices**, we are effectively summing all the "minimum" values of each pair, which is exactly the maximum possible sum of minima if you pair them optimally in ascending order.

4. **Printing the Result**

```
print(sp)
```

- Finally, we print the total sum `sp`, which represents the **maximum possible sum** of the minimum building speeds of the N pairs.

How It Solves the Problem

- **Goal**: Maximize the sum of the **minimum** speeds among pairs of workers.
- **Method**: Sort the speeds, then pair them in ascending order so that each minimum is as large as possible (pair the 1st smallest with the 2nd smallest, 3rd smallest with the 4th smallest, etc.).
- **Result**: The sum of these minimum values (taken at even indices in the sorted list) is printed.

Question : Palindrome List

```
'''
Node {
        int data,
    Node* next
}
'''
'''
This function should return true/false if linked list is palindrome/not palindrome
'''
def isPalindrome(head):
    if not head or not head.next:
        return True

    slow = head
    fast = head

    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

    prev = None
    curr = slow
    while curr:
        next_node = curr.next
        curr.next = prev
        prev = curr
        curr = next_node

    first_half = head
    second_half = prev

    while second_half:
        if first_half.data != second_half.data:
            return False
        first_half = first_half.next
        second_half = second_half.next

    return True
```

Explanation

1. **Base Cases**

```
if not head or not head.next:
    return True
```

   - If the list is empty (`head` is `None`) or has only one node (`head.next` is `None`), it is trivially a palindrome, so we return `True` .

2. **Finding the Middle of the List**

```
slow = head
fast = head
while fast and fast.next:
    slow = slow.next
    fast = fast.next.next
```

   - We use two pointers, `slow` and `fast` .
   - `slow` moves one step at a time, `fast` moves two steps at a time.
   - By the time `fast` reaches the end (or just before it), `slow` will be at the **middle** of the linked list.

3. **Reversing the Second Half**

```
prev = None
curr = slow
while curr:
    next_node = curr.next
    curr.next = prev
    prev = curr
    curr = next_node
```

   - Starting from the middle (`slow`), we reverse the linked list from this point to the end.
   - `prev` becomes the new head of the reversed second half by the time the loop finishes.

4. **Comparing Both Halves**

```
first_half = head
second_half = prev
while second_half:
    if first_half.data != second_half.data:
        return False
    first_half = first_half.next
    second_half = second_half.next
```

- We set `first_half` to the original head of the list and `second_half` to the head of the reversed second half.
- We then traverse both halves in tandem:
  - If at any point the data in `first_half` does not match the data in `second_half`, we know the list is not a palindrome, and we return `False`.
- If we finish this loop without finding a mismatch, it means the list **is** a palindrome.

5. **Return True if All Match**

```
return True
```

- If all corresponding nodes in the first and second halves match, the function returns `True`, indicating the linked list is a palindrome.

How It Works Overall

- We find the middle of the list using fast and slow pointers.
- We reverse the second half of the list in-place.
- We compare the first half and the reversed second half node by node.
- If all nodes match, the list is a palindrome; otherwise, it is not.

Question : Delete Y after X

```
# Definition of Linked List Node
#
# class Node:
#       def __init__(self, data):
#               self.data = data
#               self.next = None

# Complete the function below
def deleteYAfterX(head, N, X, Y):
    # write code here
    curr = head
    while curr:
        # Step 1: Move forward X-1 times
        for _ in range(X - 1):
            if not curr:
                return head
            curr = curr.next

        # Step 2: From the current position, skip (delete) the next Y nodes
        temp = curr.next
        for _ in range(Y):
            if not temp:
                break
            temp = temp.next

        # Step 3: Connect the current node to the node after the Y skipped nodes
        if curr:
            curr.next = temp
            curr = temp

    return head
```

Explanation

1. **Parameters**
    - **head** : The head of the linked list.

- `N` : Total number of nodes in the linked list (not directly used in the logic here, but given as part of the problem statement).
- `X` : Number of nodes to **retain** in each cycle before deletion.
- `Y` : Number of nodes to **delete** (skip) in each cycle.

2. **Traversal Using `curr`**

```
curr = head
while curr:
    ...
```

- We use `curr` to move through the linked list.
- The loop continues as long as `curr` is not `None` .

3. **Retain `X` Nodes**

```
for _ in range(X - 1):
    if not curr:
        return head
    curr = curr.next
```

- We move forward `X - 1` steps from the current node to **retain** these nodes.
- If we run out of nodes before completing `X - 1` steps (i.e., `curr` becomes `None`), we simply return `head` since there are no more nodes to process.

4. **Delete `Y` Nodes**

```
temp = curr.next
for _ in range(Y):
    if not temp:
        break
    temp = temp.next
```

- After moving `X - 1` steps, we take note of `curr.next` in a temporary pointer `temp` .
- We move `temp` forward by `Y` steps, effectively skipping those `Y` nodes.
- If `temp` becomes `None` at any point, it means we have fewer than `Y` nodes left to delete, so we stop the loop.

5. **Reconnect the List**

```
if curr:
    curr.next = temp
    curr = temp
```

- We link `curr.next` to `temp` , which points to the node after the skipped `Y` nodes.

- Then we move `curr` to `temp` so we can continue the process for the next cycle (if any nodes remain).

6. **Return the Modified Head**

```
return head
```

- Once the `while curr` loop finishes, we have retained `X` nodes and deleted `Y` nodes repeatedly throughout the list.
- We return the (potentially modified) linked list starting at `head`.

How It Works Overall

1. You skip `X-1` nodes to keep them.
2. Then, starting right after the `X` th node, you delete the next `Y` nodes.
3. You link the `X` th node to the node after those `Y` deletions.
4. You continue this process until you reach the end of the list or run out of nodes.

As a result, the final list retains groups of `X` nodes and removes `Y` nodes in between, repeated throughout the list.

Question : Can be sorted ?

```python
def solve(a, b):
    first = -1
    last = -1
    for i in range(len(a)):
        if a[i] != b[i]:
            if first == -1:
                first = i
            last = max(last, i)

    if first == last:
        return True

    for i in range(first, last):
        if a[i] < a[i + 1]:
            return False

    return True

def main():
    # Get input
    n = int(input())
    a = list(map(int, input().split()))
    b = a.copy()  # Create a copy of list a
    b.sort()      # Sort list b

    # Print result
    if solve(a, b):
        print("YES")
    else:
        print("NO")

if __name__ == "__main__":
    main()
```

Explanation

1. **Copy and Sort the Array**

```
a = list(map(int, input().split()))
b = a.copy()
b.sort()
```

- We read the original array `a` from input.
- We make a copy `b` of `a` and sort it in ascending order.

2. **Identify the First and Last Indices Where `a` Differs From `b`**

```
first = -1
last = -1
for i in range(len(a)):
    if a[i] != b[i]:
        if first == -1:
            first = i
        last = max(last, i)
```

- We iterate through the arrays to find the first position (`first`) and the last position (`last`) at which `a` and `b` differ.
- If the entire array `a` is already the same as `b` , `first` and `last` will remain `-1` until a difference is found.

3. **Check if the Array is Already Sorted**

```
if first == last:
    return True
```

- If `first == last` , it means there were no differences or only a single differing index. In either case, the array can be considered effectively sorted, so we return `True` .

4. **Check if the Subarray is Strictly Descending**

```
for i in range(first, last):
    if a[i] < a[i + 1]:
        return False
```

- We then check the subarray `a[first]` through `a[last]` . For the array to be fixable by **one reversal**, this subarray must be strictly descending.
- If we find any place where `a[i] < a[i + 1]` , the subarray is **not** strictly descending, so reversing it would not produce a sorted array. We return `False` .

5. **Return True if the Subarray is Descending**

```
return True
```

- If the subarray from `first` to `last` is strictly descending, reversing it will align `a` with `b`, making the entire array sorted. Hence, we return `True`.

6. **Main Logic**

```
if solve(a, b):
    print("YES")
else:
    print("NO")
```

- If `solve(a, b)` determines the array can be made sorted by reversing one subarray (or is already sorted), we print `"YES"`. Otherwise, we print `"NO"`.

How It Works Overall

- We compare the original array `a` with its sorted version `b`.
- We pinpoint the first and last indices where they differ.
- If that differing portion of `a` is **strictly descending**, reversing it will match the corresponding sorted segment in `b`.
- If it is not strictly descending, a single reversal can't fix the array.
- This simple check efficiently determines if the array can be made sorted by reversing at most one subarray.

Question : Subarrays Problem

```
n = int(input())
arr = list(map(int, input().split()))
res = 0
s = 0

for i in range(n - 1, -1, -1):
    if arr[i] % 2 == 1:
        s = (n - i - 1) - s
    else:
        s = s + 1
    res += s

print(res)
```

What the Code Does

This code counts the **number of subarrays whose sum is even**. The variable `res` accumulates the total count of such subarrays.

Key Variables

- `n` : The number of elements in the array.
- `arr` : The list of integers.
- `res` : The final count of subarrays with an even sum.
- `s` : A running count (updated from right to left) that helps keep track of subarray parity information.

Step-by-Step Explanation

1. **Reading Input**

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

   - We read the integer `n` , then read `n` integers into the list `arr` .
2. **Initialize Counters**

```
res = 0
s = 0
```

- `res` will store the final count of subarrays with even sum.
- `s` is a helper counter that gets updated in each iteration.

3. **Iterate from Right to Left**

```
for i in range(n - 1, -1, -1):
    ...
```

- We loop over the indices of the array from `n-1` down to `0`.

4. **Update `s` Based on the Current Element's Parity**

```
if arr[i] % 2 == 1:
    s = (n - i - 1) - s
else:
    s = s + 1
```

- If the current element `arr[i]` is **odd**, we update `s` to `(n - i - 1) - s`.
  - This effectively **flips** how many subarrays starting at `i` will have even sum versus odd sum, because an odd element toggles the parity of each subarray that includes it.
  - `n - i - 1` represents the number of ways to choose endpoints for subarrays from index `i` to the end, minus 1 for the index itself. Subarrays that were previously even become odd, and vice versa.
- If the current element is **even**, we simply do `s = s + 1`.
  - An even element does **not** flip subarray parity; instead, it keeps the parity counts consistent, effectively increasing the count of even-sum subarrays by one for the new single-element subarray `[arr[i]]`.

5. **Accumulate into `res`**

```
res += s
```

- After updating `s`, we add `s` to `res`.
- `s` now represents how many new even-sum subarrays **begin at or include** the position `i`.

6. **Print the Result**

```
print(res)
```

- Finally, we print the total count of even-sum subarrays.

Why It Works

- **Parity Flipping**: An **odd** number flips the parity (even/odd nature) of any subarray that includes it. Hence, when we encounter an odd element, the count of even-sum subarrays starting at that position is recalculated as `(n - i - 1) - s`.
- **Even Elements**: An **even** number does not flip parity; it preserves the counts of even-sum subarrays and adds exactly one new even subarray (the single element itself if it's even).
- **Right-to-Left** Iteration**: By going from the end to the beginning, we efficiently keep track of how each new element (moving leftward) affects the subarray sums.

In the end, `res` holds the **total number** of subarrays of `arr` whose sum is even.

Question : Remove to Sort

```
n = int(input())
arr = list(map(int, input().split()))
arr2 = []
min = -100000
for i in range(n):
    if arr[i] >= min:
        arr2.append(arr[i])
        min = arr[i]
for i in range(len(arr2)):
    print(arr2[i], end=" ")
```

Explanation

1. **Reading the Input**

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

   - The program reads an integer `n` (the number of elements in the array).
   - It then reads `n` integers into a list called `arr`.
2. **Initializing Variables**

```
arr2 = []
min = -100000
```

- `arr2` will store the elements that maintain the sorted (non-decreasing) sequence.
- We set `min` to a very small value, `-100000`, so that the first element of `arr` can be compared and possibly appended to `arr2`.

3. **Building the Sorted Sequence**

```
for i in range(n):
    if arr[i] >= min:
        arr2.append(arr[i])
        min = arr[i]
```

- We iterate through the array `arr`.
- For each element `arr[i]`, we check if it is **greater than or equal** to `min`.
  - If it is, we append it to `arr2` and update `min` to this new value.
  - If it isn't, we skip it, effectively "removing" it from the sequence.
- This ensures that `arr2` remains **non-decreasing** (sorted) as we move through `arr`.

4. **Printing the Result**

```
for i in range(len(arr2)):
    print(arr2[i], end=" ")
```

- Finally, we print out the elements in `arr2` in a single line, separated by spaces.

How It Works Overall

- The code keeps track of the largest value (`min`) added so far to the new list `arr2`.
- As it iterates through `arr`, it only adds elements that are **at least** as large as the last added element.
- This process removes any element that would break the sorted (non-decreasing) order, resulting in a final array `arr2` that is sorted in non-decreasing order.

Question : Push, Pop and Top

```
stack = []
ops = int(input())
for _ in range(ops):
    x = list(map(int, input().split()))
    if x[0] == 1:
        stack.append(x[1])
    elif x[0] == 2:
        if len(stack) > 0:
            stack.pop()
    else:
        if len(stack) > 0:
            print(stack[-1])
        else:
            print('Empty!')
```

Explanation

1. **Initialize the Stack**

   ```
   stack = []
   ```

   - We use a Python list `stack` to implement the stack functionality.
2. **Number of Operations**

   ```
   ops = int(input())
   for _ in range(ops):
       x = list(map(int, input().split()))
       ...
   ```

   - We read an integer `ops` which represents how many operations we will perform on the stack.
   - For each operation, we read a line of input, split it by spaces, convert each part to an integer, and store it in list `x` .
3. **Performing Operations**
   - **Operation 1**: `x[0] == 1`

```
if x[0] == 1:
    stack.append(x[1])
```

  - This means **push** the integer `x[1]` onto the top of the stack.
- **Operation 2**: `x[0] == 2`

```
elif x[0] == 2:
    if len(stack) > 0:
        stack.pop()
```

  - This means **pop** the top element from the stack (remove the last pushed element), but only if the stack is not empty.
- **Operation 3**: `x[0] == 3` (the `else` part)

```
else:
    if len(stack) > 0:
        print(stack[-1])
    else:
        print('Empty!')
```

  - This means **print** the top element of the stack (the last item in the list).
  - If the stack is empty, print `"Empty!"`.

How It Works Overall

- We use a list called `stack` to simulate standard stack operations:
  - **Push**: `stack.append(...)`
  - **Pop**: `stack.pop()`
  - **Top**: `stack[-1]`
- For each of the `ops` operations, we check the first number in the input:
  - `1`: Push the next integer onto the stack.
  - `2`: Pop the top element if the stack is not empty.
  - `3`: Print the top element if it exists; otherwise, print `"Empty!"`.
- This straightforward approach fulfills the required stack functionality in a concise manner.

Question : Discover number

```
def binary_search(array, k, n):
    left = 0
    right = n - 1
    while left <= right:

        mid = left + (right - left) // 2

        if array[mid] == k:
            return 'YES'

        elif k < array[mid]:
            right = mid - 1

        else:
            left = mid + 1
    return 'NO'

n, q = map(int, input().split())
array = list(map(int, input().split()))
array.sort()
while q:
    print(binary_search(array, int(input()), n))
    q -= 1
```

Explanation

1. **Function Definition:** `binary_search(array, k, n)`

    ```
    def binary_search(array, k, n):
        left = 0
        right = n - 1
        ...
    ```

    - This function implements the **binary search** algorithm to check if the integer `k` exists in the sorted list `array`.
    - `n` is the length of the list.

- We use two pointers, `left` (start of the list) and `right` (end of the list).

2. **Binary Search Loop**

```
while left <= right:
    mid = left + (right - left) // 2

    if array[mid] == k:
        return 'YES'
    elif k < array[mid]:
        right = mid - 1
    else:
        left = mid + 1
return 'NO'
```

- **`while left <= right`** : We repeat until our search space is invalid (i.e., `left` goes beyond `right`).
- **`mid = left + (right - left) // 2`** : We calculate the midpoint to avoid potential overflow (though in Python it's less of an issue).
- We compare `array[mid]` with the target `k` :
  - If `array[mid]` is **equal** to `k` , we return `"YES"` immediately.
  - If `k` is **less** than `array[mid]` , we move our `right` pointer leftward to `mid - 1` .
  - Otherwise, `k` is **greater** than `array[mid]` , so we move `left` pointer to `mid + 1` .
- If we exit the loop without finding `k` , we return `"NO"` .

3. **Reading Inputs**

```
n, q = map(int, input().split())
array = list(map(int, input().split()))
array.sort()
```

- We first read two integers:
  - `n` : the size of the array.
  - `q` : the number of queries.
- Next, we read the list `array` of length `n` from input.
- We **sort** the array so we can use binary search on it.

4. **Processing Queries**

```
while q:
    print(binary_search(array, int(input()), n))
    q -= 1
```

- We run a loop `q` times.

- For each query, we read an integer from input and call `binary_search(array, int(input()), n)`.
- We print the result of the binary search (`"YES"` or `"NO"`).
- We decrement `q` until all queries are processed.

How It Works Overall

1. We have an array of `n` integers.
2. We sort this array to ensure it's in ascending order, which is a prerequisite for binary search.
3. For each of the `q` queries:
   - We read an integer `k`.
   - We use binary search to see if `k` exists in the array.
   - If it does, we print `"YES"`, otherwise we print `"NO"`.