Problem 01: Binary search extension

Suppose an array of length `n` sorted in ascending order is **rotated** between `1` and `n` times. For example, the array `nums = [0,1,2,4,5,6,7]` might become:

- `[4,5,6,7,0,1,2]` if it was rotated `4` times.
- `[0,1,2,4,5,6,7]` if it was rotated `7` times.

Notice that **rotating** an array `[a[0], a[1], a[2], ..., a[n-1]]` 1 time results in the array `[a[n-1], a[0], a[1], a[2], ..., a[n-2]]`.

Given the sorted rotated array `nums` of **unique** elements, return *the minimum element of this array.*

You must write an algorithm that runs in `O(log n) time.`

Example 1:

```
Input: nums = [3,4,5,1,2]
Output: 1
Explanation: The original array was [1,2,3,4,5] rotated 3 times.
```

Example 2:

```
Input: nums = [4,5,6,7,0,1,2]
Output: 0
Explanation: The original array was [0,1,2,4,5,6,7] and it was rotated 4 times.
```

Example 3:

```
Input: nums = [11,13,15,17]
Output: 11
Explanation: The original array was [11,13,15,17] and it was rotated 4 times.
```

Problem 02: Binary search extension

There is an integer array `nums` sorted in ascending order (with **distinct** values).

Prior to being passed to your function, `nums` is **possibly rotated** at an unknown pivot index `k` (`1 <= k < nums.length`) such that the resulting array is `[nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]]` (**0-indexed**). For example, `[0,1,2,4,5,6,7]` might be rotated at pivot index `3` and become `[4,5,6,7,0,1,2]`.

Given the array `nums` **after** the possible rotation and an integer `target`, return *the index of* `target` *if it is in* `nums`, *or* `-1` *if it is not in* `nums`.

You must write an algorithm with `O(log n)` runtime complexity.

Example 1:

```
Input: nums = [4,5,6,7,0,1,2], target = 0
Output: 4
```

Example 2:

```
Input: nums = [4,5,6,7,0,1,2], target = 3
Output: -1
```

Example 3:

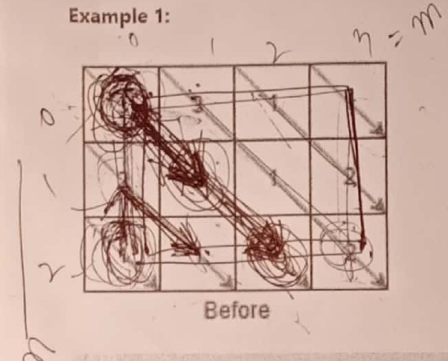
```
Input: nums = [1], target = 0
Output: -1
```

Problem 3: Sorting algorithm extension

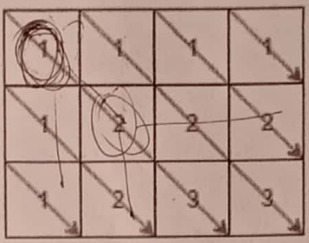
A **matrix diagonal** is a diagonal line of cells starting from some cell in either the topmost row or leftmost column and going in the bottom-right direction until reaching the matrix's end. For example, the **matrix diagonal** starting from `mat[2][0]`, where `mat` is a `6 x 3` matrix, includes cells `mat[2][0]`, `mat[3][1]`, and `mat[4][2]`.

Given an `m x n` matrix `mat` of integers, sort each **matrix diagonal** in ascending order and return *the resulting matrix.*

Example 1:



Before



After

```
Input: mat = [[3,3,1,1],[2,2,1,2],[1,1,1,2]]
Output: [[1,1,1,1],[1,2,2,2],[1,2,3,3]]
```

Example 2:

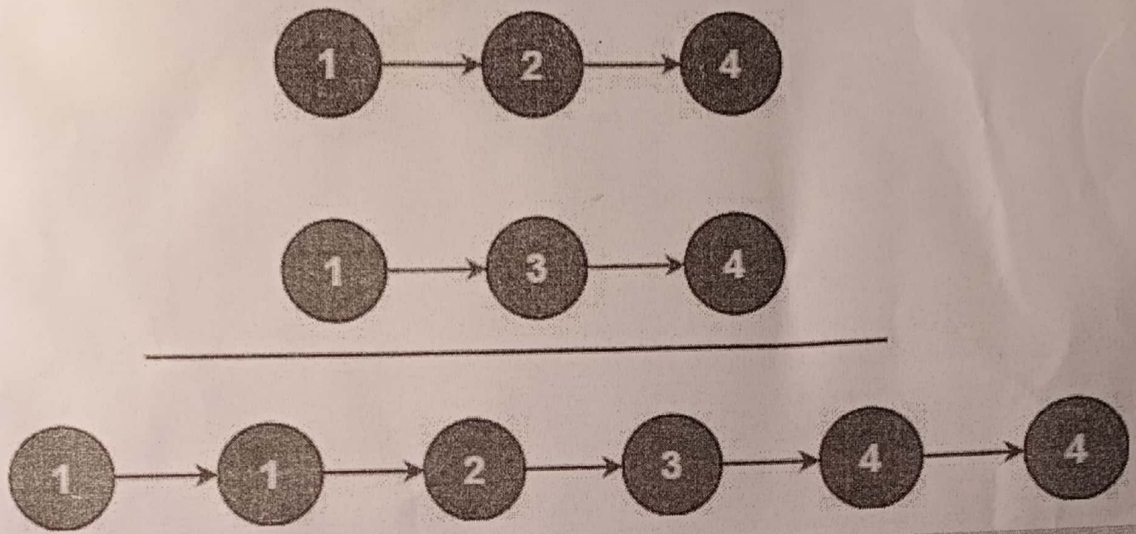
```
Input: mat = [[11,25,66,1,69,7],[23,55,17,45,15,52],[75,31,36,44,58,8],[22,27,33,25,68,4],
[84,28,14,11,5,50]]
Output: [[5,17,4,1,52,7],[11,11,25,45,8,69],[14,23,25,44,58,15],[22,27,31,36,50,66],
[84,28,75,33,55,68]]
```

You are given the heads of two sorted linked lists `list1` and `list2`.

Merge the two lists in a one **sorted** list. The list should be made by splicing together the nodes of the first two lists.

Return *the head of the merged linked list.*

Example 1:



```
Input: list1 = [1,2,4], list2 = [1,3,4]
Output: [1,1,2,3,4,4]
```

Example 2:

```
Input: list1 = [], list2 = []
Output: []
```

You are given an array of `k` linked-lists `lists`, each linked-list is sorted in ascending order.

Merge all the linked-lists into one sorted linked-list and return it.

Example 1:

```
Input: lists = [[1,4,5],[1,3,4],[2,6]]
Output: [1,1,2,3,4,4,5,6]
Explanation: The linked-lists are:
[
  1->4->5,
  1->3->4,
  2->6
]
merging them into one sorted list:
1->1->2->3->4->4->5->6
```

Example 2:

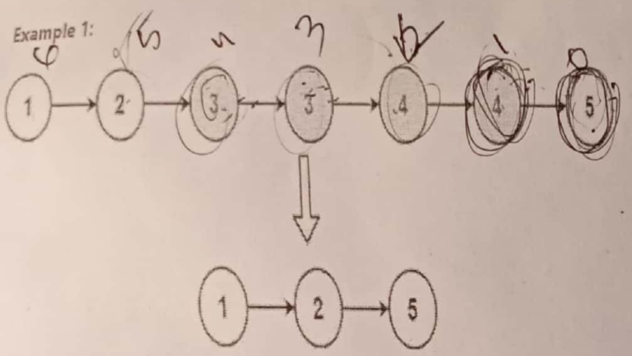
```
Input: lists = []
Output: []
```

Example 3:

```
Input: lists = [[]]
Output: []
```

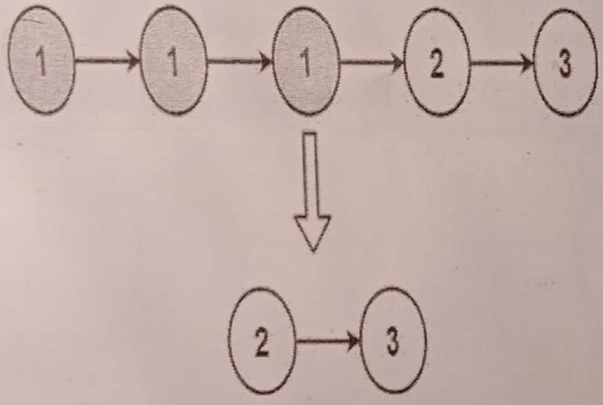
Given the `head` of a sorted linked list, *delete all nodes that have duplicate numbers, leaving only distinct numbers from the original list*. Return the linked list **sorted** as well.

Example 1:

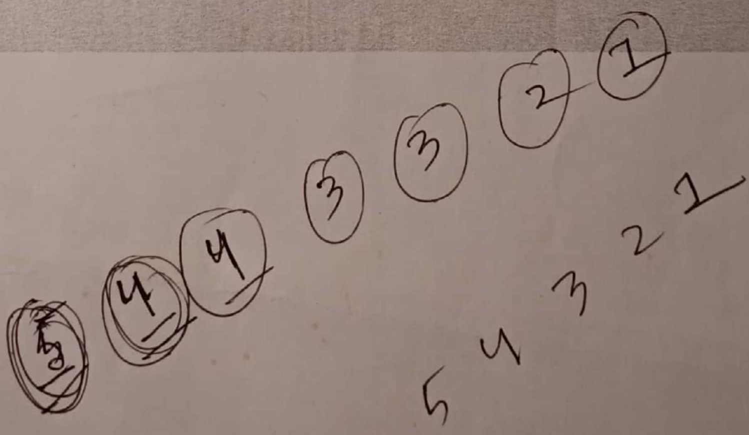


```
Input: head = [1,2,3,3,4,4,5]
Output: [1,2,5]
```

Example 2:



```
Input: head = [1,1,1,2,3]
Output: [2,3]
```



A.) [illegible] [illegible]

You have an infinite number of stacks arranged in a row and numbered (left to right) from `0`, each of the stacks has the same maximum capacity.

Implement the `DinnerPlates` class:

- `DinnerPlates(int capacity)` Initializes the object with the maximum capacity of the stacks `capacity`.
- `void push(int val)` Pushes the given integer `val` into the leftmost stack with a size less than `capacity`.
- `int pop()` Returns the value at the top of the rightmost non-empty stack and removes it from that stack, and returns `-1` if all the stacks are empty.
- `int popAtStack(int index)` Returns the value at the top of the stack with the given index `index` and removes it from that stack or returns `-1` if the stack with that given index is empty.

```
Input
["DinnerPlates", "push", "push", "push", "push", "push", "popAtStack", "push", "push", "popAtStack", "popAtStack", "pop",
"pop", "pop", "pop", "pop"]
[[2], [1], [2], [3], [4], [5], [0], [20], [21], [0], [2], [], [], [], [], []]
Output
[null, null, null, null, null, null, 2, null, null, 20, 21, 5, 4, 3, 1, -1]
```

Cmpl.

```
Explanation:
DinnerPlates D = DinnerPlates(2);  // Initialize with capacity = 2
D.push(1);
D.push(2);
D.push(3);
D.push(4);
D.push(5);         // The stacks are now:  2  4
                                           1  3  5
                                           ﹈ ﹈ ﹈
D.popAtStack(0);   // Returns 2.  The stacks are now:     4
                                                       1  3  5
                                                       ﹈ ﹈ ﹈
D.push(20);        // The stacks are now: 20  4
                                           1  3  5
                                           ﹈ ﹈ ﹈
D.push(21);        // The stacks are now: 20  4 21
                                           1  3  5
                                           ﹈ ﹈ ﹈
D.popAtStack(0);   // Returns 20.  The stacks are now:     4 21
                                                        1  3  5
                                                        ﹈ ﹈ ﹈
D.popAtStack(2);   // Returns 21.  The stacks are now:     4
                                                        1  3  5
                                                        ﹈ ﹈ ﹈
D.pop()            // Returns 5.  The stacks are now:      4
                                                        1  3
                                                        ﹈ ﹈
D.pop()            // Returns 4.  The stacks are now:   1  3
                                                        ﹈ ﹈
D.pop()            // Returns 3.  The stacks are now:   1
                                                        ﹈
D.pop()            // Returns 1.  There are no stacks.
D.pop()            // Returns -1.  There are still no stacks.
```

c = '4'   c = '3'

d = c

2. Given two integer arrays pushed and popped each with distinct values, return true if this could have been the result of a sequence of push and pop operations on an initially empty stack, or false otherwise.

Example 1:

1 2 3 5

5 3 2 1

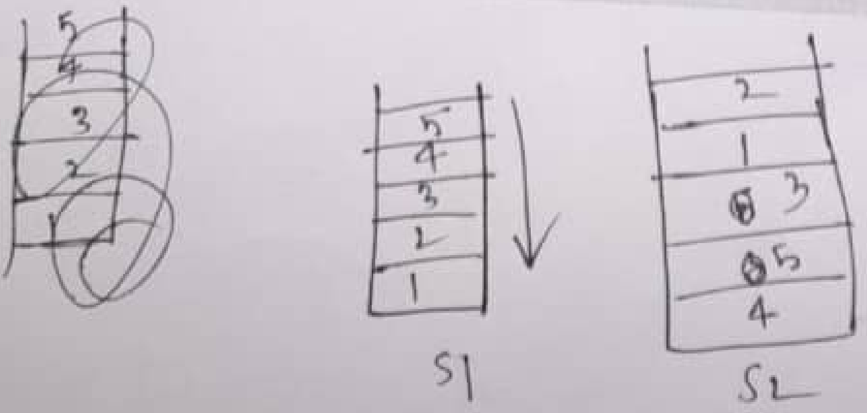
```
Input: pushed = [1,2,3,4,5], popped = [4,5,3,2,1]
Output: true
Explanation: We might do the following sequence:
push(1), push(2), push(3), push(4),
pop() -> 4,
push(5),
pop() -> 5, pop() -> 3, pop() -> 2, pop() -> 1
```

Example 2:

1 2 3 4 5

2 1 5 3 4

```
Input: pushed = [1,2,3,4,5], popped = [4,3,5,1,2]
Output: false
Explanation: 1 cannot be popped before 2.
```



1 2 3 4 5

Scanned with CamScanner

ou are given an array of strings `tokens` that represents an arithmetic expression in a Reverse Polish Notation.

↓
post

Evaluate the expression. Return *an integer that represents the value of the expression*.

Note that:

- The valid operators are `'+'`, `'-'`, `'*'`, and `'/'`.
- Each operand may be an integer or another expression.
- The division between two integers always **truncates toward zero**.
- There will not be any division by zero.
- The input represents a valid arithmetic expression in a reverse polish notation.
- The answer and all the intermediate calculations can be represented in a **32-bit** integer.

post $\longrightarrow$ infix

2 1 + 3 *

2,1 (+)

Example 1:

```
Input: tokens = ["2","1","+","3","*"]
Output: 9
Explanation: ((2 + 1) * 3) = 9
```

Example 2:

```
Input: tokens = ["4","13","5","/","+"]
Output: 6
Explanation: (4 + (13 / 5)) = 6
```

(-11)

Example 3:

```
Input: tokens = ["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
```

5

Problem 01

Implement a double-ended queue (deque) using a circular array with a fixed size. Your implementation should support the operations of addFront, addRear, removeFront, removeRear, and isEmpty.

Sample input

```
deque = MyCircularDeque(5)
deque.addFront(3)
deque.addRear(4)
deque.addRear(5)
deque.removeRear()
deque.addFront(2)
deque.removeFront()
print(deque.isEmpty())
```

Sample output: False

Problem 02:

Given a queue of strings, write a function to reverse the order of the strings in the queue without using extra space.

Sample input: ['apple', 'banana', 'cherry', 'date']
Sample output: ['date', 'cherry', 'banana', 'apple']

Problem 03:

You are given a queue of integers and a target integer x. Write a function that finds the smallest subsequence of consecutive elements in the queue that add up to at least x. If there are multiple subsequences of the same length that satisfy the condition, return the one that starts first in the queue.

Sample input:
queue = [1, 2, 3, 4, 5, 6]
x = 7
Sample output: [~~2~~, 3, 4]

Problem 04: You are given a queue of integers and an integer k. Write a function to find the sum of every kth element in the queue, starting from the first element.

Sample input:
queue = [1, 2, 3, 4, 5, 6]
k = 2
Sample output: 9

6

Write two separate programs, a linked list version and an array version, to implement a binary search tree with the following features:

- Insertion of a new node
- Deletion of a node
- After each insert and delete operation, you need to print current status of the tree
- Traversal of the tree, write separate functions for each of them (in-order, pre-order, and post-order)
- Searching for a node in the tree
- Write a function that checks whether a current version of the tree is one of the following:
    - Perfect binary tree
    - Full Binary tree ✓
    - Complete Binary tree ✓
    - Balanced binary tree ✓
    - Degenerate (or pathological) tree
        - Left-skewed binary tree
        - Right-skewed binary tree

You should also ensure that the tree maintains the properties of a binary search tree, where the left child of a node has a value less than the node, and the right child has a value greater than the node.

Your program should be able to handle any data type for the values of the nodes. Ideally, we should create node as a class.

should be able to handle cases where there are multiple nodes with the same value.

Note that you are not allowed to use any built-in libraries or modules for implementing the binary search tree.

Problem:

You have been given an unsorted array of integers. [Take a random array of size 150]. Use files for input and output.

1. Create two empty stacks: `inputStack` and `tempStack`.
2. Push all the elements of the unsorted array onto `inputStack`.
3. While `inputStack` is not empty, pop the top element and store it in a variable called `currentElement`.
4. If `tempStack` is empty or `currentElement` is greater than or equal to the top element of `tempStack`, push `currentElement` onto `tempStack`.
5. If `currentElement` is less than the top element of `tempStack`, pop elements from `tempStack` and push them onto `inputStack` until `tempStack` is empty or the top element is greater than or equal to `currentElement`. Then push `currentElement` onto `tempStack`.
6. Repeat steps 3-5 until `inputStack` is empty.
7. Finally, return the sorted array by popping all elements from `tempStack` and pushing them onto `inputStack`.

Print **inputstack**

8. In addition, you also need to insert each of the popped element of Step 7 in a linked list, named **sLink**.
9. Then, implement the bucket sort algorithm on the linked list. In so doing, you need to use 15 buckets. print sLink
10. Then, write a script that separately add the elements of **inputstack** and **sLink**, and display the difference.
11. Use a tailored binary search to find whether the average of first and last element is available in sLink.

Note: You are not allowed to use STL. Make sure your comments on your code reflects the steps (1 - 11).

Problem: Once upon a time, there was a programmer named Alice who loved solving algorithmic puzzles. One day, she came across a programming exercise that required her to implement the quick sort algorithm using a special partitioning scheme called **LT**.

Alice was excited to tackle the challenge, so she sat down at her computer and started writing the code. She began by defining the **quick sort function that takes** 100 randomly generated unsorted integers **from a file** that needed to be sorted.

Next, Alice needed to partition the array using the so called **LT** partitioning algorithm. She started by selecting the last element of the array as the pivot element. Then, she initialized two pointers, **i and j**, to the first and second elements of the array, respectively.

Alice then started iterating through the array, comparing each element with the pivot element. If the element was less than or equal to the pivot, she would swap it with the element at position i and increment i. If the element was greater than the pivot, she would simply increment j.

After iterating through the entire array, Alice swapped the pivot element with the element at position i (or i+1 – pick the current one!!!). This effectively placed the pivot element in its final position in the sorted array.

Finally, Alice recursively called the quick sort function on the subarrays to the left and right of the pivot element, respectively.

As Alice ran the code, she was impressed with how quickly it sorted the array. She used a text file to print both the **inputA** and **outputA** array.

Once Alice had sorted the array using the Quick Sort algorithm and printed the sorted array, she decided to use the linked list version of a circular queue (that she called LCQ) to solve **The Josephus Problem**.

The Josephus Problem is a mathematical puzzle that involves a group of people standing in a circle, where every other person is eliminated until only one person remains. The problem is to determine the position of the last person remaining in the circle, given the total number of people and the step size between eliminations.

Alice began by creating a linked list structure for the LCQ. The structure contained a value field to store the value of each element in the queue, as well as a next field to store a pointer to the next element in the queue.

Next, Alice initialized the LCQ by adding each element of the sorted array, **outputA** to the queue. She then set the next field of the last element in the queue to point to the first element, effectively creating a circular queue.

Alice then implemented the Josephus algorithm using the **LCQ**. She started by setting the step size to 2, and then removed every second element from the queue until only one element remained. To do this, she iterated through the queue and removed every second element by updating the next field of the previous element to skip the current element.

When she removed each element, she put it in a newly created stack **ST1**. As she removed each element from the queue, Alice printed the value of the element to the console. Finally, when only one element remained in the queue, Alice printed the value of the last remaining element, which was the solution to the Josephus Problem. She also printed the current status of ST1.

Alice was delighted with her solution to the Josephus Problem using the LCQ, and she marvelled at how the problem could be solved using different data structures and algorithms. She knew that she had learned a valuable lesson about the power of data structures and algorithms in solving complex problems.

One day, Jack decided to create a **random array (value range 10 to 50)** of 30 elements. He then took that array and inserted it into a linked list. However, Jack didn't want just any ordinary linked list; he wanted to create three separate linked lists using the same array.

For the first linked list, Jack decided to take every third element of the array. He reproduced that array into a new one and inserted it into the linked list. Jack named this linked list "List A." For the second linked list, Jack took every other element of the current array and inserted them into a new array. He then inserted this new array into a second linked list, which he named "List B." For the last linked list, Jack used the remaining values from the array and inserted them into a third linked list, which he named "List C."

Jack was proud of his creation and wanted to test it out. He decided to use a merge sort algorithm to sort the array and see how it would affect the three linked lists. Jack ran the merge sort algorithm on the **original array**, which sorted it in ascending order.

Now, Jack decided to use the merge operation of the merge sort algorithm on his three linked lists: List A, List B, and List C. He knew that merge sort used a divide-and-conquer approach, which meant that it could merge two sorted lists into a single sorted list. To start, Jack merged List A and List B using the merge operation of the merge sort algorithm. The merge operation compared the first elements of each list and sorted them in ascending order, then moved on to the next elements until all elements were sorted. The resulting list was then merged with List C using the same merge operation. Note that Jack used the following function to get the middle of a linked list, feel free to implement this in your preferred language. You also need to take a count of how many times getMiddle function has been called.

```
# Utility function to get the middle
# of the linked list
def getMiddle(self, head):
    if (head == None):
        return head

    slow = head
    fast = head

    while (fast.next != None and
            fast.next.next != None):
        slow = slow.next
        fast = fast.next.next

    return slow
```

Jack decided to create a new **stack ST1** and push the elements of the merged linked list onto it one by one. He then popped each element from the stack and added it to a new array, when you add each element to the array, make sure add 100 to each of them. Now, print the array.

Note: you are not allowed to use any built-in function.