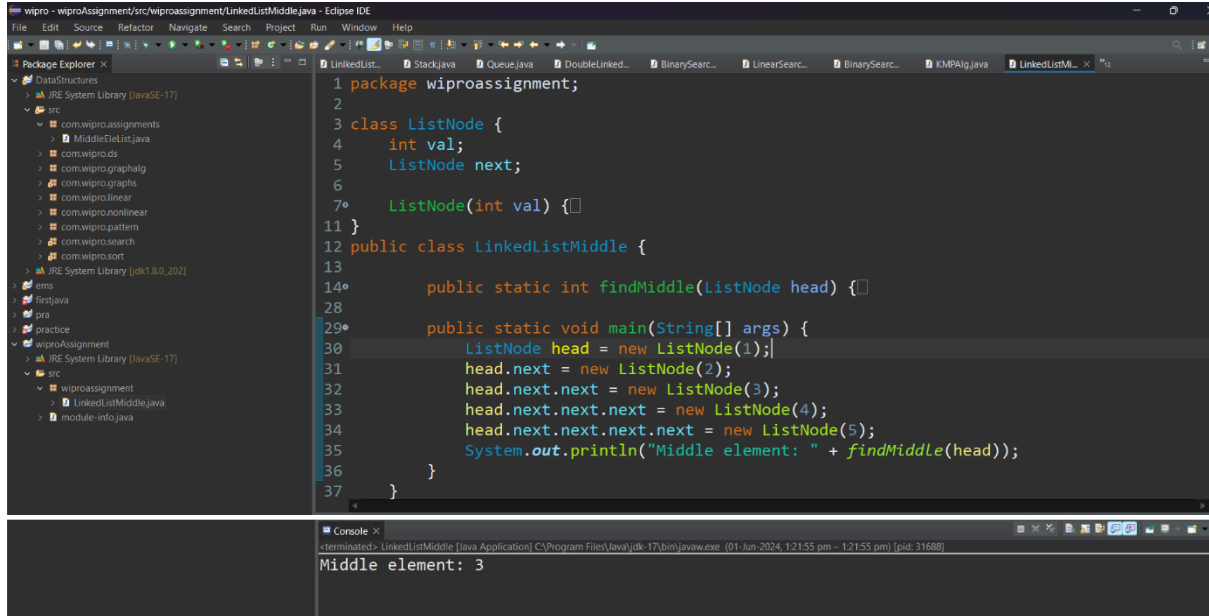


## Task 2: Linked List Middle Element Search

You are given a singly linked list. Write a function to find the middle element without using any extra space and only one traversal through the linked list.



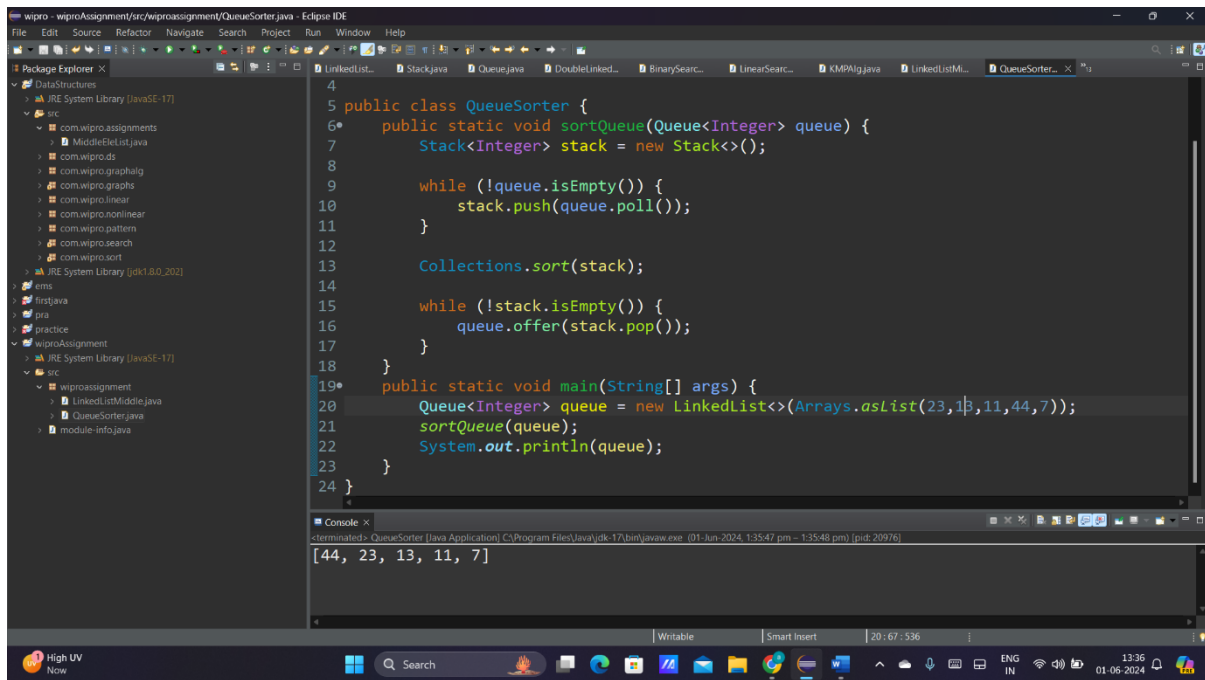
```
1 package wiproassignment;
2
3 class ListNode {
4     int val;
5     ListNode next;
6
7     ListNode(int val) {}
8
9 }
10
11
12 public class LinkedListMiddle {
13
14     public static int findMiddle(ListNode head) {}
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29     public static void main(String[] args) {
30         ListNode head = new ListNode(1);
31         head.next = new ListNode(2);
32         head.next.next = new ListNode(3);
33         head.next.next.next = new ListNode(4);
34         head.next.next.next.next = new ListNode(5);
35         System.out.println("Middle element: " + findMiddle(head));
36     }
37 }
```

Console Output:

```
<terminated> LinkedListMiddle [Java Application] C:\Program Files\Java\jdk-17\bin\java.exe (01-Jun-2024, 12:15:55 pm - 12:15:55 pm) [pid: 31688]
Middle element: 3
```

## Task 3: Queue Sorting with Limited Space

You have a queue of integers that you need to sort. You can only use additional space equivalent to one stack. Describe the steps you would take to sort the elements in the queue.



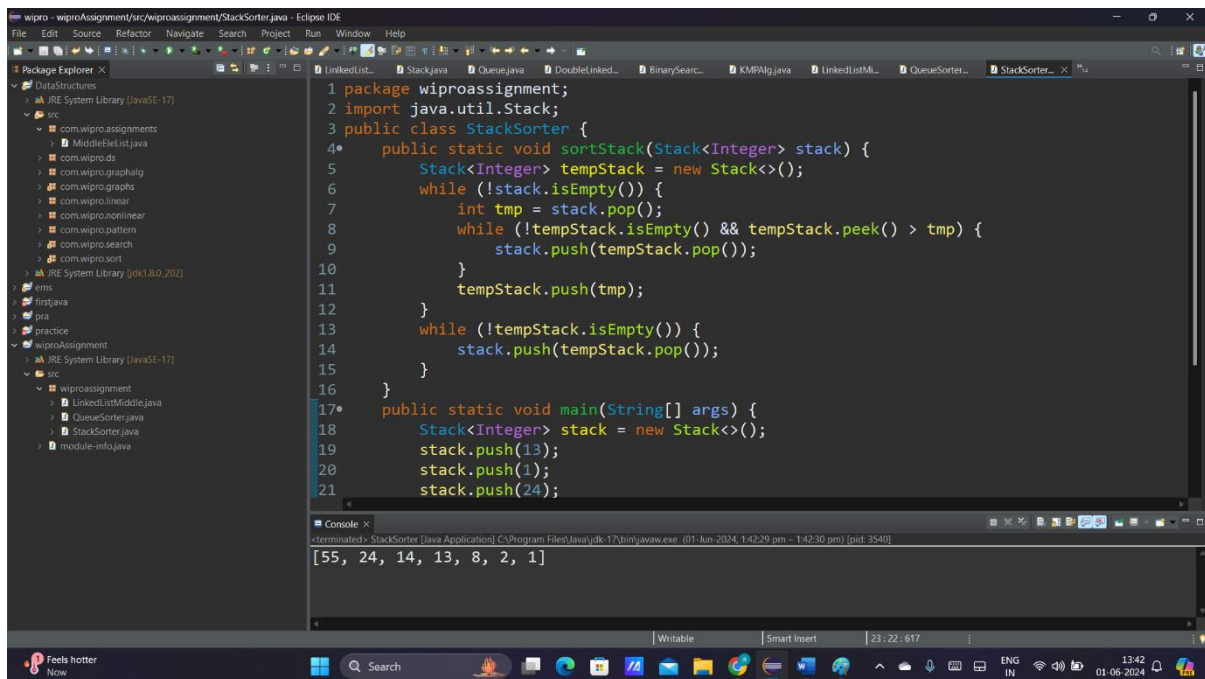
The screenshot shows the Eclipse IDE with the file `QueueSorter.java` open. The code implements a sorting algorithm using a queue and a stack. The `sortQueue` method uses a stack to sort elements from a queue. The `main` method initializes a queue with the values `[23, 13, 11, 44, 7]` and calls `sortQueue`. The console output shows the sorted array: `[44, 23, 13, 11, 7]`.

```
4 public class QueueSorter {
5     public static void sortQueue(Queue<Integer> queue) {
6         Stack<Integer> stack = new Stack<>();
7
8         while (!queue.isEmpty()) {
9             stack.push(queue.poll());
10        }
11
12        Collections.sort(stack);
13
14        while (!stack.isEmpty()) {
15            queue.offer(stack.pop());
16        }
17    }
18
19    public static void main(String[] args) {
20        Queue<Integer> queue = new LinkedList<>(Arrays.asList(23, 13, 11, 44, 7));
21        sortQueue(queue);
22        System.out.println(queue);
23    }
24 }
```

Console output: `[44, 23, 13, 11, 7]`

#### Task 4: Stack Sorting In-Place

You must write a function to sort a stack such that the smallest items are on the top. You can use an additional temporary stack, but you may not copy the elements into any other data structure such as an array. The stack supports the following operations: push, pop, peek, and isEmpty.



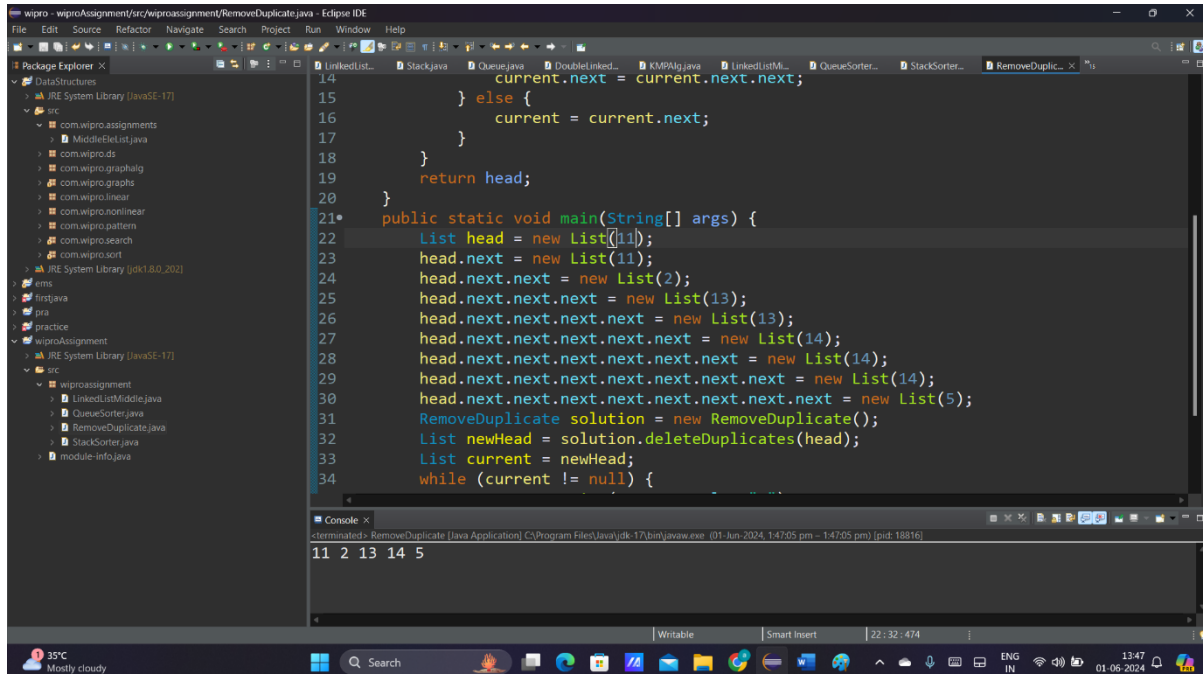
The screenshot shows the Eclipse IDE with the file `StackSorter.java` open. The code implements an in-place stack sorting algorithm using a temporary stack. The `sortStack` method uses a temporary stack to sort elements from the input stack. The `main` method initializes a stack with the values `[13, 1, 24]` and calls `sortStack`. The console output shows the sorted stack: `[55, 24, 14, 13, 8, 2, 1]`.

```
1 package wiproassignment;
2 import java.util.Stack;
3 public class StackSorter {
4     public static void sortStack(Stack<Integer> stack) {
5         Stack<Integer> tempStack = new Stack<>();
6         while (!stack.isEmpty()) {
7             int tmp = stack.pop();
8             while (!tempStack.isEmpty() && tempStack.peek() > tmp) {
9                 stack.push(tempStack.pop());
10            }
11            tempStack.push(tmp);
12        }
13        while (!tempStack.isEmpty()) {
14            stack.push(tempStack.pop());
15        }
16    }
17    public static void main(String[] args) {
18        Stack<Integer> stack = new Stack<>();
19        stack.push(13);
20        stack.push(1);
21        stack.push(24);
22    }
```

Console output: `[55, 24, 14, 13, 8, 2, 1]`

## Task 5: Removing Duplicates from a Sorted Linked List

A sorted linked list has been constructed with repeated elements. Describe an algorithm to remove all duplicates from the linked list efficiently.



```
14         current.next = current.next.next;
15     } else {
16         current = current.next;
17     }
18 }
19 return head;
20 }
21 public static void main(String[] args) {
22     List head = new List(11);
23     head.next = new List(11);
24     head.next.next = new List(2);
25     head.next.next.next = new List(13);
26     head.next.next.next.next = new List(13);
27     head.next.next.next.next.next = new List(14);
28     head.next.next.next.next.next.next = new List(14);
29     head.next.next.next.next.next.next.next = new List(14);
30     head.next.next.next.next.next.next.next.next = new List(5);
31     RemoveDuplicate solution = new RemoveDuplicate();
32     List newHead = solution.deleteDuplicates(head);
33     List current = newHead;
34     while (current != null) {
35         System.out.print(current.data + " ");
36         current = current.next;
37     }
38 }
```

Console Output: 11 2 13 14 5

## Task 6: Searching for a Sequence in a Stack

Given a stack and a smaller array representing a sequence, write a function that determines if the sequence is present in the stack. Consider the sequence present if, upon popping the elements, all elements of the array appear consecutively in the stack.

The screenshot shows the Eclipse IDE with the file `wiproAssignment/src/wiproassignment/SequenceInStack.java` open. The code implements a method `isSequenceInStack` that checks if a sequence of integers is a subsequence of a stack. The stack is represented by a `Stack<Integer>`. The method uses a `tempStack` to store elements from the original stack. It iterates through the sequence, pushing elements from the original stack onto `tempStack` until it finds the current element in the sequence. If the sequence is exhausted, it returns `seqIndex < 0`. Otherwise, it returns `seqIndex < 0` after popping all elements from `tempStack` back into the original stack.

```
1 package wiproassignment;
2
3 import java.util.Stack;
4
5 public class SequenceInStack {
6     public static boolean isSequenceInStack(Stack<Integer> stack, int[] sequence) {
7         Stack<Integer> tempStack = new Stack<>();
8         int seqIndex = sequence.length - 1;
9
10        while (!stack.isEmpty()) {
11            int element = stack.pop();
12            tempStack.push(element);
13            if (element == sequence[seqIndex]) {
14                seqIndex--;
15                if (seqIndex < 0) {
16                    break;
17                }
18            }
19        }
20        while (!tempStack.isEmpty()) {
21            stack.push(tempStack.pop());
22        }
23        return seqIndex < 0;
24    }
25    public static void main(String[] args) {
26        Stack<Integer> stack = new Stack<>();
27        stack.push(11);
```

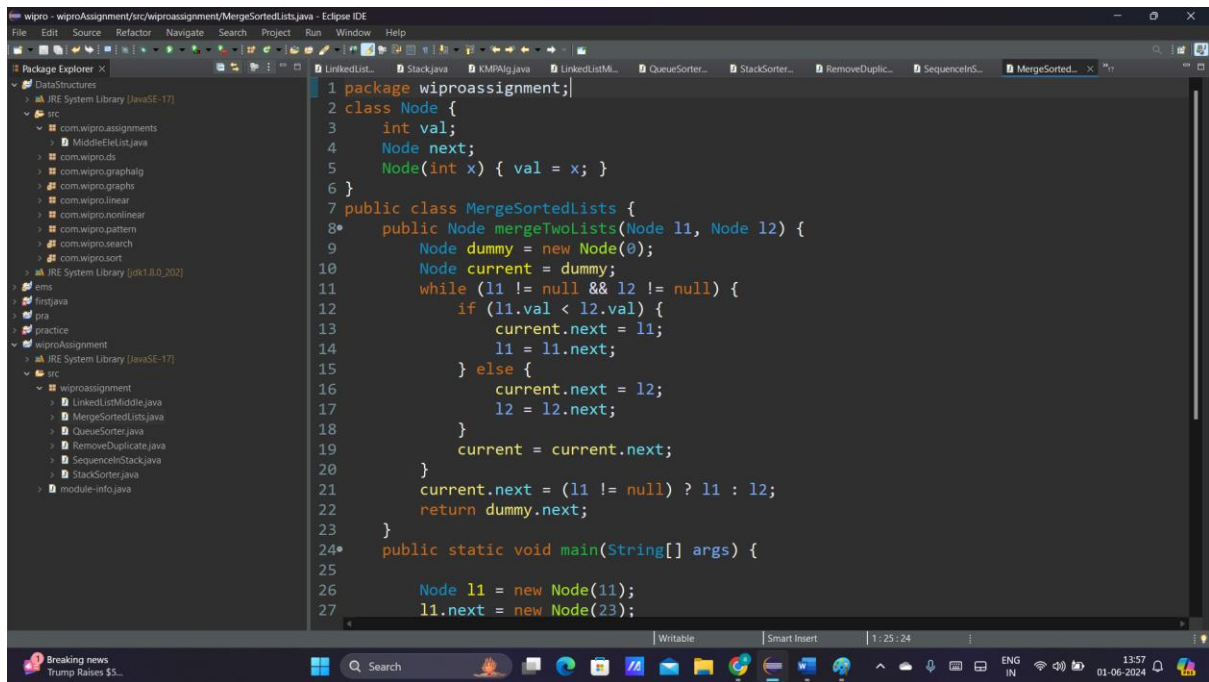
The screenshot shows the completion of the `main` method in `SequenceInStack.java`. The stack is initialized with values 11, 12, 13, 14, and 15. The sequence to be checked is `{ 13, 14, 15 }`. The `isSequenceInStack` method is called, and the result is printed to the console.

```
25    public static void main(String[] args) {
26        Stack<Integer> stack = new Stack<>();
27        stack.push(11);
28        stack.push(12);
29        stack.push(13);
30        stack.push(14);
31        stack.push(15);
32
33        int[] sequence = { 13, 14, 15 };
34        System.out.println(isSequenceInStack(stack, sequence));
35    }
36 }
37
```

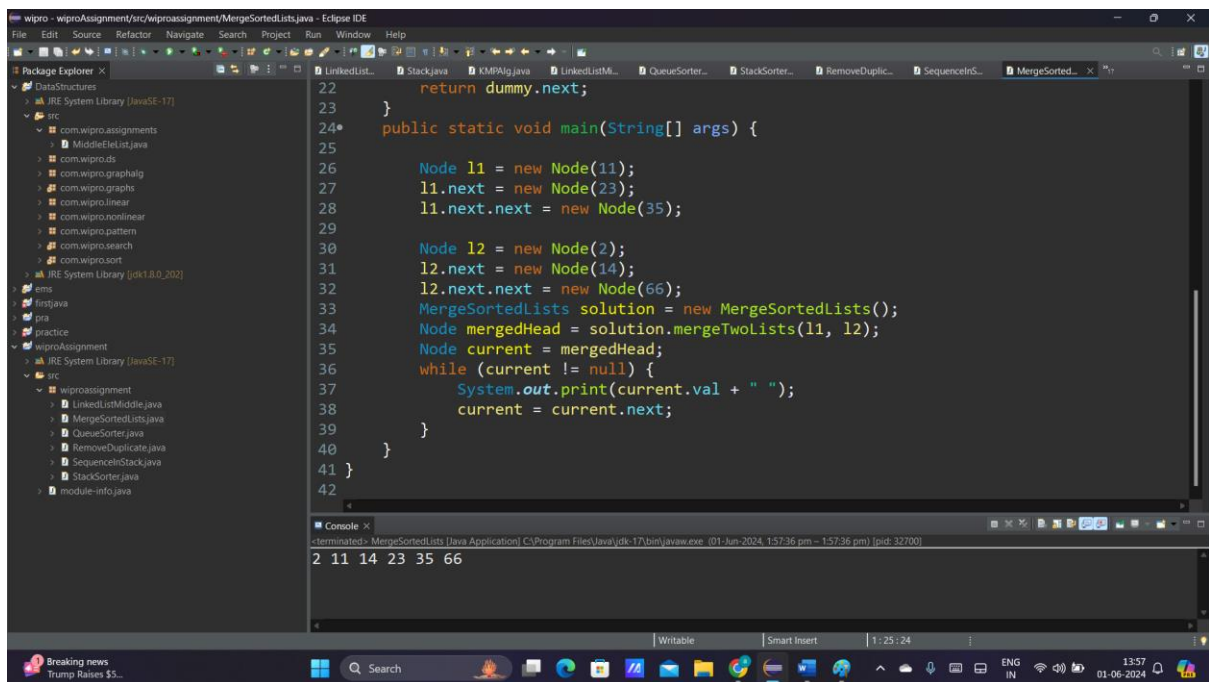
The console output shows `true`, indicating that the sequence `{ 13, 14, 15 }` is indeed a subsequence of the stack.

## Task 7: Merging Two Sorted Linked Lists

You are provided with the heads of two sorted linked lists. The lists are sorted in ascending order. Create a merged linked list in ascending order from the two input lists without using any extra space (i.e., do not create any new nodes).



```
1 package wiproassignment;
2 class Node {
3     int val;
4     Node next;
5     Node(int x) { val = x; }
6 }
7 public class MergeSortedLists {
8     public Node mergeTwoLists(Node l1, Node l2) {
9         Node dummy = new Node(0);
10        Node current = dummy;
11        while (l1 != null && l2 != null) {
12            if (l1.val < l2.val) {
13                current.next = l1;
14                l1 = l1.next;
15            } else {
16                current.next = l2;
17                l2 = l2.next;
18            }
19            current = current.next;
20        }
21        current.next = (l1 != null) ? l1 : l2;
22        return dummy.next;
23    }
24    public static void main(String[] args) {
25
26        Node l1 = new Node(11);
27        l1.next = new Node(23);
```



```
22        return dummy.next;
23    }
24    public static void main(String[] args) {
25
26        Node l1 = new Node(11);
27        l1.next = new Node(23);
28        l1.next.next = new Node(35);
29
30        Node l2 = new Node(2);
31        l2.next = new Node(14);
32        l2.next.next = new Node(66);
33        MergeSortedLists solution = new MergeSortedLists();
34        Node mergedHead = solution.mergeTwoLists(l1, l2);
35        Node current = mergedHead;
36        while (current != null) {
37            System.out.print(current.val + " ");
38            current = current.next;
39        }
40    }
41 }
42 }
```

Console Output:

```
<terminated> MergeSortedLists [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.exe (01-Jun-2024, 1:57:36 pm - 1:57:36 pm) [pid: 32700]
2 11 14 23 35 66
```

## Task 8: Circular Queue Binary Search

Consider a circular queue (implemented using a fixed-size array) where the elements are sorted but have been rotated at an unknown index. Describe an approach to perform a binary search for a given element within this circular queue.

```
1 package wiproassignment;
2
3 public class CircularQueueBinarySearch {
4     public static int search(int[] nums, int target) {
5         int left = 0, right = nums.length - 1;
6         while (left <= right) {
7             int mid = (left + right) / 2;
8             if (nums[mid] == target) {
9                 return mid;
10            }
11            if (nums[left] <= nums[mid]) {
12                if (nums[left] <= target && target < nums[mid]) {
13                    right = mid - 1;
14                } else {
15                    left = mid + 1;
16                }
17            } else {
18                if (nums[mid] < target && target <= nums[right]) {
19                    left = mid + 1;
20                } else {
21                    right = mid - 1;
22                }
23            }
24        }
25        return -1;
26    }
27    public static void main(String[] args) {
```

```
12         if (nums[left] <= target && target < nums[mid]) {
13             right = mid - 1;
14         } else {
15             left = mid + 1;
16         }
17     } else {
18         if (nums[mid] < target && target <= nums[right]) {
19             left = mid + 1;
20         } else {
21             right = mid - 1;
22         }
23     }
24 }
25 return -1;
26 }
27 public static void main(String[] args) {
28     int[] nums = {4, 45, 36, 57, 10, 21, 72};
29     int target = 0;
30     System.out.println(search(nums, target));
31 }
32 }
```

Console x  
-1