In [1]:
```python
def find_pairs_with_sum(arr, target_sum):
    pairs = []
    seen = set()

    for num in arr:
        complement = target_sum - num

        if complement in seen:
            pairs.append((num, complement))

        seen.add(num)

    return pairs

# Example usage:
arr = [1, 2, 3, 4, 5, 6]
target_sum = 7

result = find_pairs_with_sum(arr, target_sum)

if result:
    print("Pairs with sum", target_sum, "are:")
    for pair in result:
        print(pair[0], "+", pair[1], "=", target_sum)
else:
    print("No pairs found with sum", target_sum)
```

```
Pairs with sum 7 are:
4 + 3 = 7
5 + 2 = 7
6 + 1 = 7
```

In [2]:
```python
def reverse_array_in_place(arr):
    left = 0
    right = len(arr) - 1

    while left < right:
        # Swap elements at the left and right indices
        arr[left], arr[right] = arr[right], arr[left]

        # Move the indices toward the center
        left += 1
        right -= 1

# Example usage:
arr = [1, 2, 3, 4, 5]
print("Original Array:", arr)

reverse_array_in_place(arr)

print("Reversed Array:", arr)
```

```
Original Array: [1, 2, 3, 4, 5]
Reversed Array: [5, 4, 3, 2, 1]
```

In [3]:
```python
def are_rotations(str1, str2):
    # Check if both strings have the same length and are not empty
    if len(str1) != len(str2) or len(str1) == 0:
        return False

    # Concatenate str1 with itself
    concatenated = str1 + str1

    # Check if str2 is a substring of the concatenated string
    if str2 in concatenated:
        return True
    else:
        return False

# Example usage:
str1 = "abcde"
str2 = "cdeab"

if are_rotations(str1, str2):
    print("The strings are rotations of each other.")
else:
    print("The strings are not rotations of each other.")
```

```
The strings are rotations of each other.
```

In [4]:
```python
def first_non_repeated_char(input_string):
    char_count = {}  # Dictionary to store character counts

    # Count occurrences of each character in the string
    for char in input_string:
        if char in char_count:
            char_count[char] += 1
        else:
            char_count[char] = 1

    # Find the first non-repeated character
    for char in input_string:
        if char_count[char] == 1:
            return char

    # If there are no non-repeated characters, return None
    return None

# Example usage:
input_string = "hello"
result = first_non_repeated_char(input_string)

if result:
    print("The first non-repeated character is:", result)
else:
    print("No non-repeated character found in the string.")
```

```
The first non-repeated character is: h
```

In [7]:
```python
def tower_of_hanoi(n, source, auxiliary, target):                    #question no 5
    if n == 1:
        print(f"Move disk 1 from {source} to {target}")
        return
    tower_of_hanoi(n-1, source, target, auxiliary)
    print(f"Move disk {n} from {source} to {target}")
    tower_of_hanoi(n-1, auxiliary, source, target)

# Example usage:
n = 3  # Number of disks
tower_of_hanoi(n, 'A', 'B', 'C')
```

```
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C
```

In [8]:
```python
def postfix_to_prefix(postfix_expression):                           #question no 6
    stack = []
    operators = set(['+', '-', '*', '/', '^'])

    for symbol in postfix_expression:
        if symbol not in operators:
            # Operand: Push it onto the stack
            stack.append(symbol)
        else:
            # Operator: Pop two operands from the stack and form a prefix expression
            operand2 = stack.pop()
            operand1 = stack.pop()
            prefix_expression = symbol + operand1 + operand2
            stack.append(prefix_expression)

    # The final element in the stack is the prefix expression
    return stack[0]

# Example usage:
postfix_expression = "3 4 +"
prefix_expression = postfix_to_prefix(postfix_expression)
print("Prefix Expression:", prefix_expression)
```

```
Prefix Expression: 3
```

In [9]:
```python
def is_operator(char):                                               #question no 7
    return char in "+-*/^"

def prefix_to_infix(prefix_expression):
    stack = []

    for symbol in reversed(prefix_expression):
        if not is_operator(symbol):
            # Operand: Push it onto the stack as a single-element list
            stack.append([symbol])
        else:
            # Operator: Pop two operands from the stack and form an infix expression
            operand1 = stack.pop()
            operand2 = stack.pop()
            infix_expression = f"({operand1[0]} {symbol} {operand2[0]})"
            stack.append([infix_expression])

    # The final element in the stack is the infix expression
    return stack[0][0]

# Example usage:
prefix_expression = "+ 3 * 4 5"
infix_expression = prefix_to_infix(prefix_expression)
print("Infix Expression:", infix_expression)
```

```
Infix Expression: 5
```

In [10]:
```python
def are_brackets_balanced(code):                                     #question no 8
    stack = []
    opening_brackets = "{[("
    closing_brackets = "}])"

    for char in code:
        if char in opening_brackets:
            # Push opening brackets onto the stack
            stack.append(char)
        elif char in closing_brackets:
            if not stack:
                # If there are no opening brackets to match, brackets are unbalanced
                return False
            top = stack.pop()
            if (char == ')' and top != '(') or (char == ']' and top != '[') or (char == '}' and top != '{'):
                # Mismatched closing bracket
                return False

    # If the stack is empty at the end, all brackets are balanced
    return len(stack) == 0

# Example usage:
code_snippet = "{[()]}"
if are_brackets_balanced(code_snippet):
    print("Brackets are balanced in the code snippet.")
else:
    print("Brackets are not balanced in the code snippet.")
```

```
Brackets are balanced in the code snippet.
```

In [11]:
```python
class Stack:                                                         #question no 9
    def __init__(self):
        self.items = []

    def is_empty(self):
        return len(self.items) == 0

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

    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        else:
            raise IndexError("Pop from an empty stack")

    def peek(self):
        if not self.is_empty():
            return self.items[-1]
        else:
            raise IndexError("Peek at an empty stack")

    def size(self):
        return len(self.items)

def reverse_stack(input_stack):
    aux_stack = Stack()

    while not input_stack.is_empty():
        item = input_stack.pop()
        aux_stack.push(item)

    return aux_stack

# Example usage:
original_stack = Stack()
original_stack.push(1)
original_stack.push(2)
original_stack.push(3)
original_stack.push(4)

reversed_stack = reverse_stack(original_stack)

print("Original Stack:")
while not original_stack.is_empty():
    print(original_stack.pop())

print("Reversed Stack:")
while not reversed_stack.is_empty():
    print(reversed_stack.pop())
```

```
Original Stack:
Reversed Stack:
1
2
3
4
```

In [12]:
```python
class Stack:                                                         #question no 10
    def __init__(self):
        self.items = []

    def is_empty(self):
        return len(self.items) == 0

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

    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        else:
            raise IndexError("Pop from an empty stack")

    def peek(self):
        if not self.is_empty():
            return self.items[-1]
        else:
            raise IndexError("Peek at an empty stack")

    def size(self):
        return len(self.items)

class MinStack:
    def __init__(self):
        self.main_stack = Stack()  # Main stack to hold elements
        self.min_stack = Stack()   # Auxiliary stack to track minimum values

    def push(self, item):
        self.main_stack.push(item)
        if self.min_stack.is_empty() or item <= self.min_stack.peek():
            self.min_stack.push(item)

    def pop(self):
        if not self.main_stack.is_empty():
            item = self.main_stack.pop()
            if item == self.min_stack.peek():
                self.min_stack.pop()
            return item
        else:
            raise IndexError("Pop from an empty stack")

    def get_min(self):
        if not self.min_stack.is_empty():
            return self.min_stack.peek()
        else:
            raise ValueError("The stack is empty")

# Example usage:
min_stack = MinStack()
min_stack.push(3)
min_stack.push(5)
min_stack.push(2)
min_stack.push(1)
```

```
print("Smallest number in the stack:", min_stack.get_min())
```

```
Smallest number in the stack: 1
```