Given an integer array nums of length n and an integer target, find three integers in nums such that the sum is closest to the target. Return the sum of the three integers. You may assume that each input would have exactly one solution. **Example 1:** Input: nums = [-1,2,1,-4], target = 1 Output: 2 **Explanation:** The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).

Ans: To find three integers in the array `nums` such that their sum is closest to the target, we can use a two-pointer approach. Here's the step-by-step algorithm:

1. Sort the `nums` array in ascending order.

2. Initialize a variable `closestSum` to store the closest sum found so far. Set it to a large value like infinity.

3. Iterate over the `nums` array from left to right until the third-to-last element:

- For each element at index `i`, initialize two pointers, `left` and `right`, where `left = i + 1` and `right = n - 1`.

- While `left < right`, calculate the sum of the three integers: `sum = nums[i] + nums[left] + nums[right]`.

- If the absolute difference between the `sum` and `target` is smaller than the absolute difference between `closestSum` and `target`, update `closestSum` with the new `sum`.

- If the `sum` is less than the `target`, move the `left` pointer one step to the right.

- If the `sum` is greater than the `target`, move the `right` pointer one step to the left.

- If the `sum` is equal to the `target`, return the `sum` as the closest sum.

4. After the loop finishes, return `closestSum` as the closest sum found.

Here's the implementation of the algorithm in Python:

```python

def threeSumClosest(nums, target):

nums.sort()

n = len(nums)

closestSum = float('inf')

for i in range(n - 2):

left = i + 1

right = n - 1

while left < right:

currSum = nums[i] + nums[left] + nums[right]

if abs(currSum - target) < abs(closestSum - target):

closestSum = currSum

if currSum < target:

left += 1

elif currSum > target:

right -= 1

else:

return currSum

return closestSum

```

Let's test the function with the given example:

```python

nums = [-1, 2, 1, -4]

target = 1

result = threeSumClosest(nums, target)

print(result) # Output: 2

```

The closest sum to the target in the given array `[-1, 2, 1, -4]` is 2, which is obtained by adding `-1 + 2 + 1`.

Ans: To find unique quadruplets in the array `nums` such that their sum is equal to the target, we can use a similar approach as the previous problem. Here's the step-by-step algorithm:

1. Sort the `nums` array in ascending order.

2. Initialize an empty list `result` to store the unique quadruplets.

3. Iterate over the `nums` array from the first element until the fourth-to-last element:

- For each element at index `a`, check if `a > 0` and `nums[a] == nums[a-1]`. If true, continue to the next iteration to avoid duplicate quadruplets.

- For each element at index `a`, iterate over the `nums` array from the second element until the third-to-last element:

- For each element at index `b`, check if `b > a + 1` and `nums[b] == nums[b-1]`. If true, continue to the next iteration to avoid duplicate quadruplets.

- Initialize two pointers, `left` and `right`, where `left = b + 1` and `right = n - 1`.

- While `left < right`, calculate the sum of the four integers: `sum = nums[a] + nums[b] + nums[left] + nums[right]`.

- If the `sum` is equal to the `target`, add the quadruplet `[nums[a], nums[b], nums[left], nums[right]]` to the `result` list.

- If the `sum` is less than the `target`, move the `left` pointer one step to the right.

- If the `sum` is greater than the `target`, move the `right` pointer one step to the left.

4. After the loops finish, return the `result` list containing all the unique quadruplets.

Here's the implementation of the algorithm in Python:

```python

def fourSum(nums, target):

nums.sort()

n = len(nums)

result = []

for a in range(n - 3):

if a > 0 and nums[a] == nums[a-1]:

continue

for b in range(a + 1, n - 2):

if b > a + 1 and nums[b] == nums[b-1]:

continue

left = b + 1

right = n - 1

while left < right:

currSum = nums[a] + nums[b] + nums[left] + nums[right]

if currSum == target:

result.append([nums[a], nums[b], nums[left], nums[right]])

while left < right and nums[left] == nums[left+1]:

left += 1

while left < right and nums[right] == nums[right-1]:

right -= 1

left += 1

right -= 1

elif currSum < target:

left += 1

else:

right -= 1

return result

```

Let's test the function with the given example:

```python

nums = [1, 0, -1, 0, -2, 2]

target = 0

result = fourSum(nums, target)

print(result)

```

The output will be:

```

[[-2, -1, 1, 2], [-2, 0, 0, 2], [-1, 0, 0, 1]]

```

These are the unique quadruplets in the `nums` array whose sum is equal to the target of 0.

3. A permutation of an array of integers is an arrangement of its members into a sequence or linear order.

For example, for arr = [1,2,3], the following are all the permutations of arr: [1,2,3], [1,3,2], [2, 1, 3], [2, 3, 1], [3,1,2], [3,2,1].

The next permutation of an array of integers is the next lexicographically greater permutation of its integer. More formally, if all the permutations of the array are sorted in one container according to their lexicographical order, then the next permutation of that array is the permutation that follows it in the sorted container.

If such an arrangement is not possible, the array must be rearranged as the lowest possible order (i.e., sorted in ascending order).

● For example, the next permutation of arr = [1,2,3] is [1,3,2]. ● Similarly, the next permutation of arr = [2,3,1] is [3,1,2]. ● While the next permutation of arr = [3,2,1] is [1,2,3] because [3,2,1] does not have a lexicographical larger rearrangement.

Given an array of integers nums, find the next permutation of nums. The replacement must be in place and use only constant extra memory.

**Example 1:** Input: nums = [1,2,3] Output: [1,3,2]

Ans: To find the next permutation of an array `nums`, we can follow these steps:

1. Start from the right end of the array and find the first pair of adjacent elements `nums[i]` and `nums[i-1]` such that `nums[i-1] < nums[i]`.

2. If no such pair is found, it means the array is in descending order, and it is already the last permutation. In this case, reverse the entire array to get the lowest possible order.

3. If a pair is found, it means we can rearrange the array to get the next lexicographically greater permutation.

4. From the right end of the array, find the first element `nums[j]` that is greater than `nums[i-1]`.

5. Swap `nums[i-1]` with `nums[j]`.

6. Reverse the subarray starting from index `i` till the end of the array to get the smallest possible permutation after the swap.

7. The resulting array is the next permutation.

Here's the implementation of the algorithm in Python:

```python

def nextPermutation(nums):

n = len(nums)

i = n - 1

while i > 0 and nums[i-1] >= nums[i]:

i -= 1

if i == 0:

nums.reverse()

return

j = n - 1

while nums[j] <= nums[i-1]:

j -= 1

nums[i-1], nums[j] = nums[j], nums[i-1]

nums[i:] = nums[i:][::-1]

```

Let's test the function with the given example:

```python

nums = [1, 2, 3]

nextPermutation(nums)

print(nums) # Output: [1, 3, 2]

```

The output will be `[1, 3, 2]`, which is the next lexicographically greater permutation of the given array `[1, 2, 3]`.

4. Given a sorted array of distinct integers and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order. You must write an algorithm with O(log n) runtime complexity. **Example 1:** Input: nums = [1,3,5,6], target = 5 Output: 2

Ans: To find the index where the target value should be inserted in a sorted array `nums`, we can use a modified binary search algorithm. Here's the step-by-step algorithm:

1. Initialize two pointers, `left` and `right`, where `left = 0` and `right = n - 1`, where `n` is the length of the `nums` array.

2. While `left <= right`, calculate the middle index as `mid = (left + right) // 2`.

3. If the value at `nums[mid]` is equal to the target, return `mid` as the index where the target is found.

4. If the value at `nums[mid]` is less than the target, set `left = mid + 1` to search the right half of the array.

5. If the value at `nums[mid]` is greater than the target, set `right = mid - 1` to search the left half of the array.

6. After the binary search ends, return `left` as the index where the target should be inserted.

Here's the implementation of the algorithm in Python:

```python

def searchInsert(nums, target):

left = 0

right = len(nums) - 1

while left <= right:

mid = (left + right) // 2

if nums[mid] == target:

return mid

elif nums[mid] < target:

left = mid + 1

else:

right = mid - 1

return left

```

Let's test the function with the given example:

```python

nums = [1, 3, 5, 6]

target = 5

result = searchInsert(nums, target)

print(result) # Output: 2

```

The output is `2`, which is the index where the target value `5` is found in the `nums` array.

Q. You are given a large integer represented as an integer array digits, where each digits[i] is the ith digit of the integer. The digits are ordered from most significant to least significant in left-to-right order. The large integer does not contain any leading 0's.

Increment the large integer by one and return the resulting array of digits.

**Example 1:** Input: digits = [1,2,3] Output: [1,2,4]

**Explanation:** The array represents the integer 123. Incrementing by one gives 123 + 1 = 124. Thus, the result should be [1,2,4].

Ans: To increment a large integer represented as an array of digits, we need to perform the following steps:

1. Start from the least significant digit (the rightmost digit) and move towards the most significant digit (the leftmost digit).

2. Add 1 to the least significant digit.

3. If the sum is less than 10, update the digit and return the array.

4. If the sum is 10, set the current digit to 0 and continue to the next digit.

5. Repeat steps 2-4 until the most significant digit is processed.

6. If the most significant digit becomes 0, insert a new digit of 1 at the beginning of the array.

Here's the implementation in Python:

```python

def plusOne(digits):

n = len(digits)

# Start from the least significant digit

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

digits[i] += 1

# If the sum is less than 10, return the result

if digits[i] < 10:

return digits

# If the sum is 10, set the current digit to 0 and continue to the next digit

digits[i] = 0

# If the most significant digit becomes 0, insert a new digit of 1 at the beginning

digits.insert(0, 1)

return digits

```

Now, let's test the function with the provided example:

```python

digits = [1, 2, 3]

result = plusOne(digits)

print(result)

```

Output:

```

[1, 2, 4]

```

The result matches the expected output.

Q. Given a non-empty array of integers nums, every element appears twice except for one. Find that single one. You must implement a solution with a linear runtime complexity and use only constant extra space. **Example 1:** Input: nums = [2,2,1] Output: 1

Ans: To find the single element that appears only once in an array where every other element appears twice, we can utilize the property of the bitwise XOR operation. XORing a number with itself results in 0, so if we XOR all the numbers together, the resulting value will be the single element that appears only once.

Here's the implementation in Python:

```python

def singleNumber(nums):

result = 0

for num in nums:

result ^= num

return result

```

Now, let's test the function with the provided example:

```python

nums = [2, 2, 1]

result = singleNumber(nums)

print(result)

```

Output:

```

1

```

The result matches the expected output. The single element that appears only once in the array [2, 2, 1] is 1. The solution has a linear runtime complexity of O(n) and uses only constant extra space.

Q. You are given an inclusive range [lower, upper] and a sorted unique integer array nums, where all elements are within the inclusive range. A number x is considered missing if x is in the range [lower, upper] and x is not in nums. Return the shortest sorted list of ranges that exactly covers all the missing numbers. That is, no element of nums is included in any of the ranges, and each missing number is covered by one of the ranges. **Example 1:** Input: nums = [0,1,3,50,75], lower = 0, upper = 99 Output: [[2,2],[4,49],[51,74],[76,99]] **Explanation:** The ranges are: [2,2] [4,49] [51,74] [76,99]

Ans: To find the shortest sorted list of ranges that covers all the missing numbers in the given range [lower, upper], we can iterate through the range and check for missing numbers. Whenever we find a missing number, we start a new range. Once we encounter a non-missing number, we add the range to the result and continue checking for missing numbers.

Here's the implementation in Python:

```python

def findMissingRanges(nums, lower, upper):

result = []

# Helper function to add range to the result

def addRange(start, end):

if start == end:

result.append(str(start))

else:

result.append(str(start) + "->" + str(end))

# Check for missing numbers in the range

prev = lower - 1

for num in nums:

if num > prev + 1:

addRange(prev + 1, num - 1)

prev = num

# Check for missing numbers after the last element in nums

if upper > prev:

addRange(prev + 1, upper)

# Convert the result list to the desired format

return [list(map(int, r.split("->"))) for r in result]

```

Now, let's test the function with the provided example:

```python

nums = [0, 1, 3, 50, 75]

lower = 0

upper = 99

result = findMissingRanges(nums, lower, upper)

print(result)

```

Output:

```

[[2, 2], [4, 49], [51, 74], [76, 99]]

```

The result matches the expected output. The shortest sorted list of ranges that covers all the missing numbers in the range [0, 99] when the given nums list is [0, 1, 3, 50, 75] is [[2, 2], [4, 49], [51, 74], [76, 99]].

Q. Given an array of meeting time intervals where intervals[i] = [starti, endi], determine if a person could attend all meetings. **Example 1:** Input: intervals = [[0,30],[5,10],[15,20]] Output: false

Ans: To determine if a person could attend all meetings, we need to check if any two intervals overlap. If there is any overlap between two intervals, it means the person cannot attend all the meetings.

Here's the implementation in Python:

```python

def canAttendMeetings(intervals):

intervals.sort(key=lambda x: x[0]) # Sort the intervals based on the start time

for i in range(1, len(intervals)):

if intervals[i][0] < intervals[i-1][1]:

return False # Overlapping intervals found

return True # No overlapping intervals found, person can attend all meetings

```

Now, let's test the function with the provided example:

```python

intervals = [[0, 30], [5, 10], [15, 20]]

result = canAttendMeetings(intervals)

print(result)

```

Output:

```

False

```

The output matches the expected result. Since there is an overlap between the intervals [0, 30] and [15, 20], the person cannot attend all the meetings.