<https://leetcode.ca/all/1099.html>

Given an array A of integers and integer K, return the maximum S such that there exists i < j with A[i] + A[j] = S and S < K. If no i, j exist satisfying this equation, return -1.

**Example 1:**

Input: A = [34,23,1,24,75,33,54,8], K = 60

Output: 58

Explanation:

We can use 34 and 24 to sum 58 which is less than 60.

**Example 2:**

Input: A = [10,20,30], K = 15

Output: -1

Explanation:

In this case it's not possible to get a pair sum less that 15.

Note:

1 <= A.length <= 100

1 <= A[i] <= 1000

1 <= K <= 2000

**Attempt 1: 2024-01-11**

**Solution 1: Sorting + Two Pointers (10min)**

import java.util.\*;

public class Solution {

public int twoSumLessThanK(int[] numbers, int targetSum) {

Arrays.sort(numbers);

int max = -1;

int i = 0;

int j = numbers.length - 1;

while(i < j) {

if(numbers[i] + numbers[j] < targetSum) {

// The 1st pair which sum up < targetSum not guaranteed

// to be get the maximum sum up less than targetSum

// e.g nums = [1,2,...57], target = 60, the 1st pair

// 1 + 57 = 58 < 60, but the 2nd pair 2 + 57 = 59 < 60

// also, and 59 > 58

max = Math.max(max, numbers[i] + numbers[j]);

i++;

} else {

j--;

}

}

return max;

}

public static void main(String[] args) {

Solution so = new Solution();

int[] nums = new int[]{34, 23, 1, 24, 75, 33, 54, 8};

int result = so.twoSumLessThanK(nums, 60);

System.out.println(result);

}

}

Time Complexity: O(N\*logN)

Space Complexity: O(1)

**Refer to**

<https://algo.monster/liteproblems/1099>

**Problem Description**

Given an array nums of integers and an integer k, the challenge is to find the maximum possible sum of any two distinct elements (i.e., nums[i] and nums[j] where i < j) that is less than k. To clarify:

The sum should be less than k.

You need to find two different elements in the array (no element should be used twice).

The maximum sum needs to be returned.

If it is not possible to find such a pair, the function should return -1.

**Intuition**

The intuition behind the solution involves sorting the array first. When the array is sorted, we can use a two-pointer approach to efficiently find two numbers with the desired property. The first pointer i starts at the beginning of the list, and the second pointer j starts at the end. The sorting allows us to make decisions based on the sum of the elements at the two pointers:

If the sum of the two pointed-to numbers is less than k, then we have a valid pair, and we can try to find a bigger sum by moving pointer i one step to the right (increasing the sum).

If the sum is greater or equal to k, then we must reduce the sum by moving pointer j one step to the left.

Throughout the process, we maintain the highest sum obtained that is less than k, and this becomes our answer. The approach works because sorting the array allows us to know that, as i moves to the right, the sum will increase, and as j moves to the left, the sum will decrease. This helps us avoid considering every possible pair, which reduces the time complexity significantly compared to a brute-force approach.

**Solution Approach**

The solution makes effective use of a two-pointer technique with a sorted array to optimize the process of finding the maximum sum of a pair of integers that is less than k. Here's a step-by-step breakdown of the implementation:

Sort the Array: The first step is to sort the nums array which will arrange the integers in non-decreasing order. Sorting is crucial because it allows us to explore potential sums from the smallest and largest values and move our pointers based on the comparison with k.

Initialize Two Pointers: We set up two pointers, i and j, where i starts from the beginning of the array (0 index) and j starts from the end (len(nums) - 1). These pointers will traverse the array from opposite ends towards each other.

Initialize the Answer Variable: An ans variable is initialized with a value of -1. This variable will hold the maximum sum encountered that is still less than k.

Iterate with Two Pointers: We then enter a while loop that will continue as long as i is less than j. Inside the loop, we do the following:

Calculate the sum (s) of the elements pointed to by i and j.

If this sum is less than k, compare it with the current ans variable and update ans to be the maximum of the two. Since we want to maximize our sum, we then move i one step to the right (i.e., i += 1) to potentially increase our sum.

If the sum is greater than or equal to k, we must reduce it. To do this, we move j one step to the left (i.e., j -= 1), as this will use a smaller number for the sum.

Return the Answer: After the loop terminates, the variable ans will contain the maximum sum less than k that we have found, or -1 if no such sum exists. We return ans.

By following these steps, the solution ensures that every move of either i or j is purposeful and brings us closer to the maximum sum we can achieve under the constraint that it must be less than k. Through this optimization, the solution effectively manages the potential combinations of pairs, resulting in a more efficient algorithm.

**Example Walkthrough**

Suppose we have an array nums = [34, 23, 1, 24, 75, 33, 54, 8] and an integer k = 60.

According to our solution approach:

Sort the Array: First, we sort the array: nums = [1, 8, 23, 24, 33, 34, 54, 75].

Initialize Two Pointers: We set two pointers, i = 0 (pointing to 1) and j = 7 (pointing to 75).

Initialize the Answer Variable: We initialize an answer variable with ans = -1.

Iterate with Two Pointers: We enter the while loop with the condition i < j.

During the first iteration, the sum s = nums[i] + nums[j] = 1 + 75 = 76 which is greater than k. Hence, we decrease j to point to the second last element (j = 6 pointing to 54).

In the next iteration, the sum s = 1 + 54 = 55. This sum is less than k, so we update ans = 55. We want to see if we can get closer to k, so we increase i to point to the next element (i = 1 pointing to 8).

Now, the sum s = 8 + 54 = 62, which is greater than k. So we decrease j to point to j = 5 (pointing to 34).

The sum s = 8 + 34 = 42. This is less than k and greater than the current ans, so we update ans = 42. We try increasing i again (i = 2 pointing to 23).

The sum s = 23 + 34 = 57. It is less than k and greater than ans, so we update ans = 57. Since we are still trying to maximize the sum, we increment i again (i = 3, pointing to 24).

Continuing the iteration, we keep moving i and j to find larger sums until i is no longer less than j.

Return the Answer: Once i >= j, the loop terminates. Our ans is the largest sum found which is less than k, which in this case is 57. Hence, we would return 57.

**Java Solution**

class Solution {

/\*\*

\* Finds the maximum sum of any pair of numbers in the array that is less than K

\*

\* @param numbers Array of integers

\* @param targetSum Target sum K we are trying not to exceed

\* @return Maximum sum of a pair of numbers less than K, or -1 if such a pair does not exist

\*/

public int twoSumLessThanK(int[] numbers, int targetSum) {

// Sort the array in ascending order

Arrays.sort(numbers);

// Initialize the answer as -1 assuming there might be no valid pairs

int maximumSum = -1;

// Initialize two pointers, one at the beginning (left) and one at the end (right) of the array

int left = 0, right = numbers.length - 1;

// Iterate over the array using the two-pointer approach

while (left < right) {

// Calculate the sum of values at the left and right pointers

int sum = numbers[left] + numbers[right];

// If the sum is less than the targetSum, update maximumSum and move the left pointer forward

if (sum < targetSum) {

maximumSum = Math.max(maximumSum, sum);

left++;

} else {

// If the sum is greater than or equal to targetSum, move the right pointer backward

right--;

}

}

// Return the maximum sum found, or -1 if no suitable pair was found

return maximumSum;

}

}

**Time and Space Complexity**

**Time Complexity**

The time complexity of the provided function is O(N log N) because of the sort operation. The subsequent while loop has at most N iterations (where N is the length of nums), since each iteration either increments i or decrements j, ensuring that each element is considered at most once. However, the sort operation dominates the time complexity.

Therefore, the overall time complexity is associated with the sorting step, which for most sorting algorithms like Timsort (used in Python's .sort() method), is O(N log N).

**Space Complexity**

The space complexity of the function is O(1) or constant space complexity, assuming that the sort operation is done in-place. Aside from sorting, only a fixed number of integer variables are introduced (i, j, s, and ans), which do not depend on the size of the input array nums.

If the sorting algorithm used is not in-place, the space complexity would be O(N) due to the space required to create a separate sorted array. However, since Python's default .sort() method sorts the array in place, we consider the space complexity to be O(1).