JASIA ALAM

**Homework 4 Python**

**Two Sum**  
-------------------------------------------------------------------------------------------------------------------------  
Given an array of integers nums and an integer target, return two numbers inside that array such that they add up to target.  
  
You may assume that each input would have at least one solution, and you may not use the same element twice.  
You can return the answer in any order.  
  
 An input MAY HAVE no two numbers at all (an empty one still counts as a solution) - in this case, return a empty array  
- It's an array of integer numbers - these numbers are not necessarily distinct / unique  
- Make sure to discuss your solution - what is the Big O Time & Space complexity? Was there anyway you could've done...  
...better or not? Why or why not? Justify.  
  
EXAMPLE 1:  
---------------------------------------------------------------  
Input: nums = [2,7,11,15], target = 9  
Output: [2, 7]  
Explanation: Because nums[0] + nums[1] == 9, we return [2, 7].  
  
EXAMPLE 2:  
---------------------------------------------------------------  
Input: nums = [3,2,4], target = 6  
Output: [2, 4]  
  
EXAMPLE 3:  
---------------------------------------------------------------  
Input: nums = [3,3], target = 6  
Output: [3, 3]  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Solution1:**  
---------------------------------------------------------------

def twoSum(nums, target):

# Find pair of numbers that add to the target

for i in range(len(nums) - 1):

for j in range(i + 1, len(nums)):

# Add pair of numbers if they meet the target value

if nums[i] + nums[j] == target:

# print the pair of numbers if they successfully equate to target

print("answer : (", nums[i], ",", nums[j], ")")

# array and target

nums = [1, 2, 4, 3, 6 , 5 ]

target = 6

# using twoSum to find the target through the array of numbers

twoSum(nums, target)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Although this returned the pair of numbers that add up to the target 6 like this:

answer : ( 1 , 5 )

answer : ( 2 , 4 )

This solution loops through the array twice so it can find the pair of numbers that add up to the target sum. The time complexity for this is 0(n²). This is because the search is conducted through a nested loop. It has a Quadratic Time when it performs a linear time operation for each element.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Solution 2: A better and perhaps optimal solution would be to use Hash table as shown below:**

def twoSumHashing(nums, target):  
 sums = []  
 hashElements = {}  
  
 for i in range(len(nums)):  
 complement = target - nums[i]  
 if complement in hashElements:  
 print("Pair with sum", target, "is: (", nums[i], ",", complement, ")")  
 hashElements[nums[i]] = nums[i]  
  
  
#array and target  
nums = [10, 6, 4, 0]  
target = 10  
  
# function used  
twoSumHashing(nums, target)

We can look at the order of the array in the hash table.

This involves creating an empty hash table.

So for each element the complementary paired number can be found by subtracting the element in the list from the target value. So our target value is 9 and we can subtract the first element in the list which is 9.

i.e 10-10= 0. So we have 10 as our first element and can pair it with 0.

Using this method, the current element in the hash table is always looking for its complementary pair value to add up to the target value.

The hash table has a time complexity which is linear with a constant (O(1)) time. It’s worse space complexity is 0(n) and this means it has a linear complexity so the space it takes is directly proportional to the input size. The essence for time is considered more important than the space, so the time complexity is considerably better than the first solution I went with. However, since it is creating a new ‘hash’ table to work out the complementary pairs that is the reason it takes more space than the first solution. (O(1)) time complexity is considered the best for solutions as developers are always looking for the quickest solutions to their problems. 0(n²)solutions are considered the worst by the graph below and best to be avoided. So my first solution is worse to go with due to its time complexity being 0(n²).

Graphical user interface, application, PowerPoint

Description automatically generated