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.

1. Time complexity describes how long it takes for the algorithm to give an output. Essentially, the number of times statements get executed.

2. Space complexity tells us how much space an algorithm takes up specifically concerning the input of the algorithm.  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

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

def twoSum(nums, target):  
 for i in range(len(nums) - 1):  
 for j in range(i + 1, len(nums)):  
 if nums[i] + nums[j] == 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)

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

It returned the pair of numbers that add up to the target 6:

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 takes 0(1) space. This is 2XO(1 ) for the integers, both “I” and “j”.
* Using this method is not an optimised solution as it takes too long.

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

**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("Pairs of numbers", "are: (", nums[i], ",", complement, ")")  
 hashElements[nums[i]] = nums[i]  
  
  
#array and target  
nums = [10, 6, 4, 0]  
target = 10  
  
# function used  
twoSumHashing(nums, target)

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 for solution 2 than the 1st solution I went with.

Graphical user interface, application, PowerPoint

Description automatically generated

* (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 above and best to be avoided. So, my first solution is worse due to its time complexity being 0(n²).
* It’s always best to choose the optimal solution for both time and space complexities as it becomes very expensive if they are not optimal.
* For time-complexity, dictionaries/hash tables are way faster/optimal than lists in python.
* The space-complexity is 0(n) .
* 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.
* In terms of space complexity, it requires more memory allocation. If in your situation, time is more crucial than space -- this is the better option.

**More Solutions and Discussions:**

**For time complexities:** Although I mentioned that the hash table was an optimal solution as hashable dictionaries have a time-lookup that is usually 0(1) they can actually become worse with a t-complexity of 0(n). This can occur if there is a back issue or conflict with the hash table.

I could’ve used a solution with set too. With set and dictionaries the time-complexity is usually 0(1) which is highly optimal, however sometimes for dictionary iterations they can be 0(n) which is not the best but quite ideal.

With has tables the time-lookup is usually 0(1) but can actually be worse and become 0(n).

**For space complexity:** The space-complexity for the second solution is not as ideal compared to the first however when a developer is more pressed from time and not as concerned about space – this is the most optimal solution. If there is lots of space on the hardware, developers will prefer to go with the quicker function. This is because space is much cheaper than time, although cache space is expensive and that is something to be aware of as a growing developer as the performance of it is essential alongside SSD which is also expensive. Mobile apps are a good example of things that take up too much space and it is not ideal for it to take up too much space as the user will have to start deleting things valuable to them or cant use specific apps due to little space. So space-complexities are important in that regard. Storing memory for faster methods is ideal so it is good to ensure that time-complexities and also space-complexities are optimised but time is given priority.

**In terms of optimisation,** it is better to use dictionaries or sets to put the data when trying to search for elements in an array. Frequent use of lists is not advised for optimisation as it can lead to duplication and time inefficiency so it may be better for lists to be changed into a set. Hash tables are considered to be highly optimal in terms of time but there are other methods such as set which have just as good time-complexity but may have better storage-complexity.