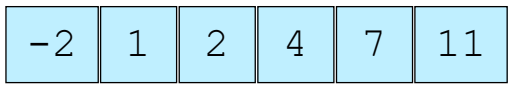
Two Sum Problem

In this lesson, you will learn how to solve the Two Sum Problem using three solutions in Python.

We'll cover the following

In this lesson, we are going to be solving the “Two-Sum Problem”. Let’s begin by defining the problem. Given an array of integers, return `True` or `False` if the array has two numbers that add up to a specific target. You may assume that each input would have exactly one solution.



target = 13

(2,11)
True

We investigate three different approaches to solving this problem.

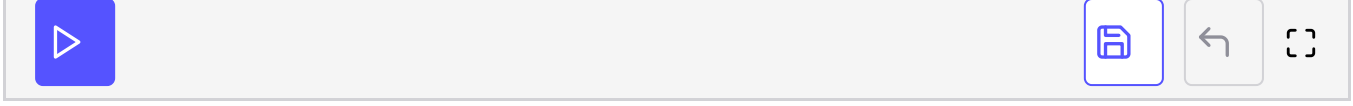
Solution 1

A brute-force approach that takes $O(n^2)$ time to solve with $O(1)$ space where we loop through the array and create all possible pairings of elements.

Implementation

Have a look at the code below:

```
# Time Complexity: O(n^2)
# Space Complexity: O(1)
def two_sum_brute_force(A, target):
  for i in range(len(A)-1):
    for j in range(i+1, len(A)):
      if A[i] + A[j] == target:
        print(A[i], A[j])
        return True
  return False

A = [-2, 1, 2, 4, 7, 11]
target = 13
print(two_sum_brute_force(A,target))
target = 20
print(two_sum_brute_force(A,target))
```



Explanation

The outer loop on **line 4** iterates over all the elements in `A` while the inner loop on **line 5** iterates from the next index of `i` to the last index of `A` in every iteration of the outer loop. These nested loops help us form all possible pairs of array elements in every iteration. On **line 6**, we check if the pair `A[i]` and `A[j]` add up to the `target`. If they do, we have found a solution, and we return `True` on **line 8**. Otherwise, we move on to the next pair. If we are unable to find a solution after both the loops terminate, we return `False` on **line 9**.

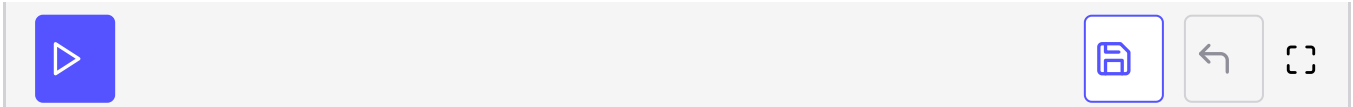
This solution is highly inefficient as the code has a time complexity of $O(n^2)$ because of the nested loops. Let's see if we can improve it.

Solution 2

A slightly better approach time-wise, taking $O(n)$ time, but worse from a space standpoint, with space complexity of $O(n)$. In this approach, we make use of an auxiliary hash table to keep track of whether it's possible to construct the target based on the elements we've processed thus far in the array.

Implementation

```
# Time Complexity: O(n)
# Space Complexity: O(n)
def two_sum_hash_table(A, target):
  ht = dict()
  for i in range(len(A)):
    if A[i] in ht:
      print(ht[A[i]], A[i])
      return True
    else:
      ht[target - A[i]] = A[i]
  return False

A = [-2, 1, 2, 4, 7, 11]
target = 13

print(two_sum_hash_table(A,target))
```



Explanation

In the code above, we use a Python dictionary and name it `ht` on **line 4**. On **line 5**, using a for loop, we iterate over `A` and check if the element `A[i]` is present in `ht` or not (**line 6**). If it's not, we calculate `target - A[i]` and using that as a key, we store `A[i]` as its value in `ht` on **line 10**. Now it is easy for us to check for the next elements if we ever come across an element which is equal to `target - A[i]`. If `A[i]` in any iteration is present as a key in `ht`, the execution jumps to **line 7**. Note that we have been storing `target - A[i]` as keys and `A[i]` as values in the previous iterations. Therefore, if `A[i]` of the current iteration is present as a key in `ht`, we have found a pair as the sum of that key and its value will equal `target`. In this case, we print out the pair and return `True` (**lines 7-8**). On the other hand, if we never come across such a pair after all the iterations of the for loop, we return `False` on **line 11**.

As we iterate the list once, the time complexity for Solution 2 is $O(n)$ where n is the length of the list, and as we make an entry in the dictionary for every element in the list, the space complexity is also in $O(n)$.

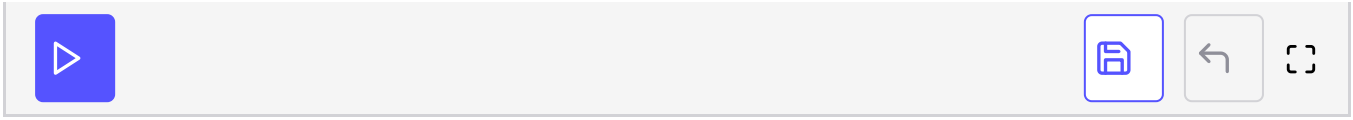
Solution 3

This approach has a time complexity of $O(n)$ and a constant space complexity, $O(1)$. Here, we have two indices that we keep track of, one at the front and one at the back. We move either the left or right indices based on whether the sum of the elements at these indices is either greater than or less than the target element.

Implementation

```
# Time Complexity: O(n)
# Space Complexity: O(1)
def two_sum(A, target):
  i = 0
  j = len(A) - 1
  while i < j:
    if A[i] + A[j] == target:
      print(A[i], A[j])
      return True
    elif A[i] + A[j] < target:
      i += 1
    else:
      j -= 1
  return False

A = [-2, 1, 2, 4, 7, 11]
```

```
target = 13

print(two_sum(A,target))
```



Explanation

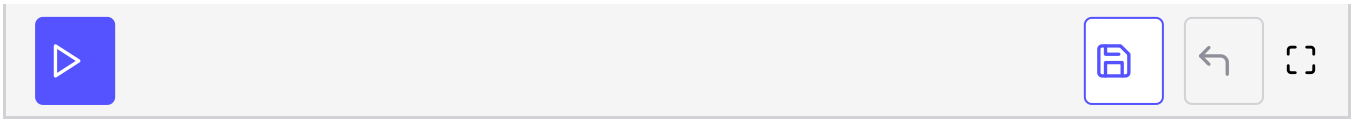
This approach assumes that the array is sorted. So `i` and `j` are set to `0` and `len(A) - 1` respectively (**lines 4-5**), i.e., the minimum and the maximum value. On **line 6**, a while loop is set which will run as long as `i` < `j`. If at any iteration, `A[i] + A[j]` equal `target`, we have found the pairs and we return `True` from the function after printing the pairs (**line 7-9**). However, if the sum of `A[i]` and `A[j]` is less than `target`, we increment `i` by `1` on **line 11**, so that `A[i]` will be a greater value in the next iteration and will produce a greater sum than the current sum. On the other hand, if the sum of `A[i]` and `A[j]` is greater than `target`, we decrement `j` on **line 13** to achieve a lesser sum by moving to an index with a smaller value. After the `while` loop terminates, we return `False` (**line 14**) as we never come across a pair which sums to `target` during the execution of the `while` loop.

There is no extra data structure to store values which implies that the code has a space complexity of $O(1)$ while the time complexity is $O(n)$ as we traverse the entire list `A` using `i` and `j`.

All the above solutions are implemented in the code widget below. Test them on your list and target value!

```
A = [-2, 1, 2, 4, 7, 11]
target = 13

# Time Complexity: O(n^2)
# Space Complexity: O(1)
def two_sum_brute_force(A, target):
  for i in range(len(A)-1):
    for j in range(i+1, len(A)):
      if A[i] + A[j] == target:
        print(A[i], A[j])
        return True
  return False


# Time Complexity: O(n)
# Space Complexity: O(n)
def two_sum_hash_table(A, target):
  ht = dict()
```

```
  for i in range(len(A)):
    if A[i] in ht:
      print(ht[A[i]], A[i])

      return True
    else:
      ht[target - A[i]] = A[i]
  return False


# Time Complexity: O(n)
# Space Complexity: O(1)
def two_sum(A, target):
  i = 0
  j = len(A) - 1
  while i < j:
    if A[i] + A[j] == target:
      print(A[i], A[j])
      return True
    elif A[i] + A[j] < target:
      i += 1
    else:
      j -= 1
  return False


print(two_sum_brute_force(A, target))
print(two_sum_hash_table(A, target))
print(two_sum(A, target))
```



In the next lesson, we will look at the “Optimal Task Assignment” Problem. See you there!