Problem Statement

Given an array of integers nums and an integer target, return indices of the two numbers such that they add up to target. You may assume that each input would have exactly one solution, and you may not use the same element twice. You can return the answer in any order.

Example:

Input: nums = [2,7,11,15], target = 9 Output: [0,1] Explanation: Because nums[0] + nums[1] == 9, we return [0, 1].

```
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        # generate all the possible pairs of numbers from nums array
        # n -> len(nums) the total number of items in the array nums
        # iterate a loop with i from 0 to n-2
        for i in range(0, len(nums)-1):
            # for every i, iterate the loop with j from i+1 to n-1
            for j in range(i+1, len(nums)):
                # check if the sum of a pair is equal to the target
                if(nums[i] + nums[j] == target):
                    # if true, then i, j are the indices we are looking
for
                    # return those indices
                    return [i, j]
```

```
/**
 * @param {number[]} nums
 * @param {number} target
 * @return {number[]}
 */
var twoSum = function (nums, target) {
  // generate all the possible pairs from nums
  for (let i = 0; i < nums.length - 1; i++) {
    for (let j = i + 1; j < nums.length; j++) {
      if (nums[i] + nums[j] === target) {
        return [i, j];
      }
    }
  }
};
```

```
class Solution {
    public int[] twoSum(int[] nums, int target) {
        // created an array with the resulting indices to return
        int[] indices = new int[2];
        // generate all the possible pairs
        for(int i=0; i<nums.length-1; i++){
            for(int j=i+1; j<nums.length; j++){
                if(nums[i] + nums[j] == target){
                    indices[0] = i;
                    indices[1] = j;
                    return indices;
                }
            }
        }

        return indices;
    }
}
```

All the above solutions have a time complexity of O(n^2) and space complexity of O(1). Can you solve this problem with a time complexity of O(n) and space complexity of O(n)?

Algorithm:

// 1. create an empty hash table <- map

```
// 2. iterate the nums array i <- 0 to n-1
    // 3. for every num in nums array, do the following
    // 4. check if target-nums[i] exists in the hash table or not
        // 5. if it does, then return the indices of i and map[target-
nums[i]]

        // 6. else, if it does not exist, then we should create a new
entry map[nums[i]] = i

// 7. return an empty array
```

```
/**
 * @param {number[]} nums
 * @param {number} target
 * @return {number[]}
 */
var twoSum = function (nums, target) {
  // 1. create an empty hash table <- map
  let map = {};

  // 2. iterate the nums array i <- 0 to n-1
  for (let i = 0; i < nums.length; i++) {
    // 3. for every num in nums array, do the following
    // 4. check if target-nums[i] exists in the hash table or not
    if (target - nums[i] in map) {
      // 5. if it does, then return the indices of i and map[target -
nums[i]]
      return [map[target - nums[i]], i];
    } else {
      // 6. else, if it does not exist, then we should create a new entry
map[nums[i]] = i
      map[nums[i]] = i;
    }
  }
};
```

```
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        #  1. create an empty hash table <- map
        map = {}

        #  2. iterate the nums array i <- 0 to n-1
        for i in range(0, len(nums)):
            #  3. for every num in nums array, do the following
            #  4. check if target-nums[i] exists in the hash table or not
            if (target-nums[i]) in map:
                #  5. if it does, then return the indices of i and
map[target-nums[i]]
                return [map[target-nums[i]], i]
            else:
                #  6. else, if it does not exist, then we should create a
new entry map[nums[i]] = i
                map[nums[i]] = i
```

```
class Solution {
    public int[] twoSum(int[] nums, int target) {

        // 1. create an empty hash table <- map
        HashMap<Integer, Integer> map = new HashMap<>();

        // create an result indices array to return
        int[] indices = new int[2];

        // 2. iterate the nums array i <- 0 to n-1
        for(int i=0; i<nums.length; i++){
            // 3. for every num in nums array, do the following
            // 4. check if target-nums[i] exists in the hash table or not
            if(map.containsKey(target-nums[i])){
                // 5. if it does, then return the indices of i and
map[target-nums[i]]
                indices[0] = map.get(target-nums[i]);
                indices[1] = i;

                return indices;
            } else {
                // 6. else, if it does not exist, then we should create a
new entry map[nums[i]] = i
                map.put(nums[i], i);
            }
        }

        // 7. return an empty array
        return indices;
    }
}
```

Two Pointers Technique

Two pointers is a technique where you use two pointers to traverse the array. The two pointers can be the same direction or opposite direction. The two pointers technique is used in many problems like

```
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        # create a 2d array with the original indices
        numsWithIndices = [[nums[index], index] for index in
range(len(nums))]

        numsWithIndices.sort(reverse = False)

        # deploy the two pointers
        start = 0
        end = len(nums) - 1

        # iteratively do the following
        while start<end:
            # check if the sum of the elements at start and end are equal
            if (numsWithIndices[start][0] + numsWithIndices[end][0]) ==
target:
                return [numsWithIndices[start][1], numsWithIndices[end]
[1]]
            elif numsWithIndices[start][0] + numsWithIndices[end][0] <
target:
                start += 1
            else:
                end -= 1
```

```
/**
 * @param {number[]} nums
 * @param {number} target
 * @return {number[]}
 */
var twoSum = function (nums, target) {
  let nums2d = nums.map((num, index) => [num, index]);
```

```
    nums2d.sort((a, b) => a[0] - b[0]);

    let start = 0;
    let end = nums.length - 1;

    while (start < end) {
      if (nums2d[start][0] + nums2d[end][0] === target) {
        return [nums2d[start][1], nums2d[end][1]];
      } else if (nums2d[start][0] + nums2d[end][0] > target) {
        end--;
      } else {
        start++;
      }
    }
  };
```

```
class Solution {
    public int[] twoSum(int[] nums, int target) {
        int[][] nums2d = new int[nums.length][2];

        int[] indices = new int[2];

        for(int i=0; i<nums.length; i++){
            nums2d[i][0] = nums[i];
            nums2d[i][1] = i;
        }

        Arrays.sort(nums2d, new Comparator<int[]>(){
            public int compare(int[] a, int[] b){
                return Integer.compare(a[0], b[0]);
            }
        });

        int start = 0;
        int end = nums.length - 1;

        while(start < end){
            if(nums2d[start][0] + nums2d[end][0] == target){
                indices[0] = nums2d[start][1];
                indices[1] = nums2d[end][1];
                return indices;
            } else if(nums2d[start][0] + nums2d[end][0] < target){
                start++;
            } else {
                end--;
            }
        }

        return indices;
    }
}
```

Abstract Data Types

1. Linked List
2. Stack
3. Queue
4. Trees and Graphs

Linked List

A linked list is a linear data structure where each element is a separate object. Each element (we will call it a node) of a list is comprising of two items - the data and a reference to the next node. The last node has a reference to null. The entry point into a linked list is called the head of the list. It should be noted that the head is not a separate node, but the reference to the first node. If the list is empty then the head is a null reference.

Leetcode: finding the middle of a linked list

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next =
next; }
 * }
 */
class Solution {
    public int findLength(ListNode head){
        ListNode thead = head;
        int nodes = 0;
        while(thead != null){
            nodes++;
            thead = thead.next;
        }
        return nodes;
    }

    public ListNode middleNode(ListNode head) {
        // find the length of the linked list
        int length = findLength(head);

        // divide the length by 2 and get rid of the fractional part <-
moves
        int moves = length/2;

        // move the head pointer to 'moves' number of times
        while(moves>0){
            head = head.next;
```

```
                moves--;
            }

            return head;
        }
    }
```

```python
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution:
    def findLength(self, head):
        nodes = 0
        thead = head
        while thead != None:
            nodes += 1
            thead = thead.next
        return nodes

    def middleNode(self, head: Optional[ListNode]) -> Optional[ListNode]:
        # find the length of the list
        length = self.findLength(head)

        moves = length//2

        while moves>0:
            head = head.next
            moves-=1

        return head
```

```javascript
/**
 * Definition for singly-linked list.
 * function ListNode(val, next) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.next = (next===undefined ? null : next)
 * }
 */
/**
 * @param {ListNode} head
 * @return {ListNode}
 */
function findLength(head) {
  let nodes = 0;
  let thead = head;

  while (thead != null) {
    nodes++;
    thead = thead.next;
  }
  return nodes;
}

var middleNode = function (head) {
  let length = findLength(head);

  let moves = Math.floor(length / 2);

  while (moves--) {
    head = head.next;
  }

  return head;
};
```