إفهم السؤال وإفهم الجواب حتى لو شفته ، بعدين أرجع حله بنفسك مرة ثانية

Trade off / mind mapping شرح سيرش عن مفهوم

Problem 1 : Two Sum : Answer

Approach 1: Brute Force

Algorithm:-

The brute force approach is simple. Loop through each element x and find if there is another value that equals to target-x

Brute force, in algorithms, refers to a straightforward approach that exhaustively tries every possible solution to solve a problem. While simple and direct, it may not be the most efficient method for large or complex problems. For instance, consider the "Two Sum" problem.

A brute force solution involves using nested loops to iterate over each pair of numbers in an array and checking if their sum equals a given target.

Implementation

```
vector<int> twoSum(vector<int>& nums, int target) {
for (int i = 0; i < nums.size(); i++) {
        for (int j = i + 1; j < nums.size(); j++) {
                if (nums[i] + nums[j] == target) {
                 return {i, j};
        } } }
        return {};
}
```

This brute force approach checks every pair of numbers in the array to find the target sum. While it works, **its time complexity is O(n^2), which may not be efficient for large arrays.**

Complexity Analysis:-

Time complexity: O(n^2). For each element, we try to find its complement by looping through the rest of the array which takes O(n) time. Therefore, the time complexity is O(n^2).

Space complexity: O(1). The space required does not depend on the size of the input array, so only constant space is used.

Approach 2: Two-pass Hash Table

To improve our runtime complexity, we need a more efficient way to check if the complement exists in the array. (hash table)

Algorithm

A simple implementation uses two iterations. In the first iteration, we add each element's value as a key and its index as a value to the hash table. Then, in the second iteration, we check if each element's complement ($target - nums[i]$) exists in the hash table. If it does exist, we return the current element's index and its complement's index. Beware that the complement must not be

$nums[i]$ itself!

Implementation

```
vector<int> twoSum(vector<int>& nums, int target) {
```

```
    unordered_map<int, int> hashmap;
    for (int i = 0; i < nums.size(); ++i) {
        hashmap[nums[i]] = i;
    }
    for (int i = 0; i < nums.size(); ++i) {
        int complement = target - nums[i];
        if (hashmap.find(complement) != hashmap.end() && hashmap[complement] != i) {
            return {i, hashmap[complement]};
        }
    }
    return {}; // No solution found
}
```

Complexity Analysis:-

Time complexity:O(n).

We traverse the list containing n elements exactly twice. Since the hash table reduces the lookup time to O(1), the overall time complexity isO(n).

Space complexity: O(n).

The extra space required depends on the number of items stored in the hash table, which stores exactly n elements.

Approach 3: One-pass Hash Table

Algorithm

It turns out we can do it in one-pass. While we are iterating and inserting elements into the hash table, we also look back to check if current element's complement already exists in the hash table. If it exists, we have found a solution and return the indices immediately.

Implementation

```
vector<int> twoSum(vector<int>& nums, int target) {
    unordered_map<int, int> hashmap;
    for (int i = 0; i < nums.size(); ++i) {
        int complement = target - nums[i];
        if (hashmap.find(complement) != hashmap.end()) {
            return {i, hashmap[complement]};
        }
        hashmap[nums[i]] = i;
    }
    return {}; // No solution found
}
```

Complexity Analysis

Time complexity: $O(n)$.

We traverse the list containing n elements only once. Each lookup in the table costs only $O(1)$ time**.**

Space complexity: $O(n)$.

The extra space required depends on the number of items stored in the hash table, which stores at most n elements.

Problem 2: [Palindrome Number]

Approach 1:Reversing the Entire Number

```
bool isPalindrome(int x) {
        if (x < 0) {
                return false;
        }
        long long reversed = 0;
        long long temp = x;
        while (temp != 0) {
                int digit = temp % 10;
                reversed = reversed * 10 + digit;
                temp /= 10;
        }

        return (reversed == x);
}
```

Approach 2: Reversing Half of the Number

```
bool isPalindrome(int x) {
        if (x < 0 || (x != 0 && x % 10 == 0)) {
                return false;
                }

                int reversed = 0;
                while (x > reversed) {
                        reversed = reversed * 10 + x % 10;
                        x /= 10;
                }
                return (x == reversed) || (x == reversed / 10);
}
```

Approach 3: Comparing Digits

```
bool isPalindrome(int x) {
    if (x < 0) {
        return false;
    }
    string str = to_string(x);
    // Compare digits at corresponding positions
    int left = 0, right = str.length() - 1;
    while (left < right) {
        if (str[left] != str[right]) {
            return false;
        }
        left++;
        right--;
    }
    return true;
}
```