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 : Answer

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;
}
```

Problem 3 : [13. Roman to Integer](#) : [Answer](#)

Intuition: Important 🙂

The key intuition lies in the fact that in Roman numerals, when a smaller value appears before a larger value, it represents subtraction, while when a smaller value appears after or equal to a larger value, it represents addition.

You Can Use Map OR Unordered_map

Problem 4: [14. Longest Common Prefix](#) : [Answer](#)

Problem 5: [20. Valid Parentheses](#) : [Answer](#)

Approach

Here is the step-by-step approach of the algorithm:

Initialize an empty stack.

Traverse the input string character by character.

If the current character is an opening bracket (i.e., '(', '{', '['), push it onto the stack.

If the current character is a closing bracket (i.e., ')', '}', ']'), check if the stack is empty. If it is empty, return false, because the closing bracket does not have a corresponding opening bracket. Otherwise, pop the top element from the stack and check if it matches the current closing bracket. If it does not match, return false, because the brackets are not valid.

After traversing the entire input string, if the stack is empty, return true, because all opening brackets have been matched with their corresponding closing brackets. Otherwise, return false, because some opening brackets have not been matched with their corresponding closing brackets.

Implementation

```
bool isValid(string s) {
      stack<char> st; // create an empty stack to store opening brackets
      for (char c : s) { // loop through each character in the string
         if (c == '(' || c == '{' || c == '[') { // if the character is an opening bracket
```

```
            st.push(c); // push it onto the stack
        } else { // if the character is a closing bracket
            if (st.empty() || // if the stack is empty or
                (c == ')' && st.top() != '(') || // the closing bracket doesn't match the
corresponding opening bracket at the top of the stack
                (c == '}' && st.top() != '{') ||
                (c == ']' && st.top() != '[')) {
                return false; // the string is not valid, so return false
            }
            st.pop(); // otherwise, pop the opening bracket from the stack
        }
    }
    return st.empty(); // if the stack is empty, all opening brackets have been matched
with their corresponding closing brackets,
                        // so the string is valid, otherwise, there are unmatched opening
brackets, so return false
  }
```

Complexity Analysis

Time complexity: The time complexity of the solution is O(n)O(n)O(n), where n is the length of the input string. This is because we traverse the string once and perform constant time operations for each character.

Space complexity: The space complexity of the solution is O(n)O(n)O(n), where n is the length of the input string. This is because the worst-case scenario is when all opening brackets are present in the string and the stack will have to store them all.

Problem 6: [21. Merge Two Sorted Lists](#) : *Answer*

EXPLANATION😁

Maintain a head and a tail pointer on the merged linked list.

Then choose the head of the merged linked list by comparing the first node of both linked lists.

For all subsequent nodes in both lists, you choose the smaller current node and link it to the tail of the merged list, and move the current pointer of that list one step forward.

You keep doing this while there are some remaining elements in both the lists.
If there are still some elements in only one of the lists, you link this remaining list to the tail of the merged list.
Initially, the merged linked list is NULL.
Compare the value of the first two nodes and make the node with the smaller value the head node of the merged linked list.
Since it's the first and only node in the merged list, it will also be the tail.
Then move head1 one step forward.

Approach 1 : RECURSIVE APPROACH

Implementation

```
ListNode* mergeTwoLists(ListNode* l1, ListNode* l2)
{
                // if list1 happen to be NULL
                // we will simply return list2.
                if(l1 == NULL)
                 {
                        return l2;
                }

                // if list2 happen to be NULL
                // we will simply return list1.
                if(l2 == NULL)
                {
                        return l1;
                }

                // if value pointed by l1 pointer is less than equal to value pointed by l2
pointer
                // we wall call recursively l1 -> next and whole l2 list.
                if(l1 -> val <= l2 -> val)
                 {
                        l1 -> next = mergeTwoLists(l1 -> next, l2);
                        return l1;
```

```
        }
        // we will call recursive l1 whole list and l2 -> next
        else
        {
            l2 -> next = mergeTwoLists(l1, l2 -> next);
            return l2;
        }
}
```

Complexity Analysis:-

Time complexity: O(n+m)

Space complexity: O(n+m) this is auxiliary stack space due to recursion.

Approach 2: Iterative Approach

Implementation

```
ListNode* mergeTwoLists(ListNode* list1, ListNode* list2) {

        // if list1 happen to be NULL
            // we will simply return list2.
    if(list1 == NULL)
        return list2;

            // if list2 happen to be NULL
            // we will simply return list1.
    if(list2 == NULL)
        return list1;

    ListNode * ptr = list1;
    if(list1 -> val > list2 -> val)
    {
        ptr = list2;
        list2 = list2 -> next;
    }
    else
    {
        list1 = list1 -> next;
    }
```

```
        ListNode *curr = ptr;

                // till one of the list doesn't reaches NULL
        while(list1 &&  list2)
        {
            if(list1 -> val < list2 -> val){
                curr->next = list1;
                list1 = list1 -> next;
            }
            else{
                curr->next = list2;
                list2 = list2 -> next;
            }
            curr = curr -> next;

        }

                // adding remaining elements of bigger list.
        if(!list1)
            curr -> next = list2;
        else
            curr -> next = list1;

        return ptr;
    }
```

Complexity Analysis:-

Time complexity: O(n+m)

Space complexity: O(1)