Leetcode solutions and notes

Nathan Warner

Computer Science
Northern Illinois University
United States

Contents

Key insights

1.1 Complexity

- Creating a constant sized array (for all input sizes) is $\Theta(1)$ space

  If the array has a fixed, constant size (e.g., 10 elements, regardless of the input size n), creating it takes a constant amount of time, which is $\Theta(1)$. The size of the input n does not affect the time taken to allocate a constant amount of memory.

- **Proposition.** Looping through a constant-sized array (for all input sizes) is $\Theta(1)$:

  This statement is **false**. While the array size is constant, looping through it will take a constant amount of time proportional to its size, which is independent of n. However, the time complexity for such an operation is still described as $\Theta(c)$, where c is the constant size of the array. Since $\Theta(1)$ specifically means the operation takes a fixed, constant time (regardless of any other factor), this description is incorrect.

  A better description would be that looping through a constant-sized array is constant time with respect to n, but its complexity is actually $\Theta(c)$, where c is the array size.

  Consider

```
void foo(const vector<int>& v) {
    //  Loop through input specific vector
    for (const auto& item : v) {
        cout << item << endl;
    } cout << endl;

    // Create a constant sized vector
    vector<int> x(10,0);

    // Loop through constant sized vector
    for (const auto& item : x)  {
        cout << item << endl;
    }
}
```

  If v has n elements, the first loop runs n iterations, and each iteration performs $O(1)$ work to print the item. Thus, the complexity of this part is $O(n)$.

  A vector of size 10 is created, and all elements are initialized to 0. Since the size of the vector is constant (10), this operation takes $O(1)$ time.

  Looping through it takes a constant amount of time, specifically $O(10)$, which simplifies to $O(1)$ in Big-O notation because it is independent of the input size.

The overall time complexity is the sum of the complexities of all the components:

$$\mathcal{O}(n) + \mathcal{O}(1) + \mathcal{O}(1) = \mathcal{O}(n)$$

- Creating a variable to store the size of a passed container is $O(1)$. A single variable is created to store the size of the vector. The space required for this variable is constant, O(1), as it occupies a fixed amount of memory (typically 4 or 8 bytes, depending on your system architecture).
- **Space complexity for a collection of strings**: The space complexity for say a vector of strings is $O(n \cdot k)$, where n is the size of the vector and k is the average length of the strings.

  Note that the space complexity for a vector of numeric types is $O(n)$, where n is the size of the vector. The difference in space complexity between a vector of integers (std::vector<int>) and a vector of strings (std::vector<std::string>) lies in the way integers and strings are stored and managed in memory.

  Numeric types occupies a fixed amount of memory. Unlike int, which has a fixed size, a std::string can grow in size and stores its character data in a dynamically allocated buffer. Thus, the space complexity for a container of strings is $O(n \cdot k)$.

1.2 Bitwise operations

1.2.1 XOR

The bitwise XOR (exclusive OR) is a binary operator in programming, typically represented by the symbol ^. It operates on two integers by comparing their binary representations bit by bit.

Recall the truth table for $\oplus$

p	q	$p \oplus q$
1	1	0
1	0	1
0	1	1
0	0	0

XOR has the following properties

1. $a \oplus a = 0$
2. $a \oplus 0 = 0 \oplus a = a$

Consider the integers $a = 5_{10} = 0101_2$, $b = 5_{10} = 0101_2$. Then, $a \oplus b$ is computed by a bitwise xor

$$\begin{array}{rcccc} & 0 & 1 & 0 & 1 \\ \oplus & 0 & 1 & 0 & 1 \\ \hline & 0 & 0 & 0 & 0 \end{array}$$

Next, consider the integers $a = 5_{10} = 0101_2$, and $b = 2_{10} = 0010_2$. Then,

$$\begin{array}{rcccc} & 0 & 1 & 0 & 1 \\ \oplus & 0 & 0 & 1 & 0 \\ \hline & 0 & 1 & 1 & 1 \end{array}$$

Thus, $a \oplus b = 0101 \oplus 0010 = 0111 = 7_{10}$

Arrays and hashing

2.1 Contains duplicates

Given an integer array `nums`, return true if any value appears **at least twice** in the array, and return false if every element is distinct.

2.1.1 O(nlg n) time O(n) space Solution

To achieve a solution of $O(n\lg(n))$, we use a $std :: map$

```
bool containsDuplicate(vector<int>& nums) {
    map<int,int> m;

    for (const auto& item: nums) {
        ++m[item];

        if (m[item] > 1) return true;
    }
    return false;
}
```

Because $std :: map$s are implemented with a red-black tree, insertion into them takes $O(\lg(n))$ time. Since we are inserting n elements, the total time required is therefore $O(n\lg(n))$

In the worst case, there are n unique elements, thus requiring $O(n)$ space to store each element in the map.

2.1.2 O(n) time O(n) space

To achieve $O(n)$ time with $O(n)$ space, we can instead use an $std :: unordered\_map$, which uses a hash map under the hood. Insertion into the hash map is $O(1)$, as opposed to the $O(\lg(n))$ time required for the standard map. Thus, inserting n elements into the hash map takes total time $O(n)$.

```
 bool containsDuplicate(vector<int>& nums) {
    unordered_map<int,int> m;

    for (const auto& item: nums) {
        ++m[item];

        if (m[item] > 1) return true;
    }
    return false;
}
```

2.2 Anagram

Given two strings s and t, return true if t is an anagram of s, and false otherwise. Recall that a string s is an anagram of t if and only if some permutation of s is precisely t

2.2.1 O(nlg n + mlg m) time O(1) space

If we simply sort both strings with an $O(n\lg(n))$ sorting algorithm, then for strings s, t with length n, m, we get running time

$$O(n\lg(n)) + O(m\lg(m)) = O(n\lg(n) + m\lg(m))$$

Comparing each strings requires $O(k)$ time, where k is the length of the smaller string. Formally, $k = \min(|s|, |t|)$. The total running time of the algorithm is therefore

$$O(n\lg(n) + m\lg(m)) + O(k) = O(n\lg(n) + m\lg(m))$$

No additional space is required. The space is therefore $O(1)$.

```cpp
bool isAnagram(string s, string t) {
    if (s.size() != t.size()) return false;

    sort(s.begin(), s.end()); sort(t.begin(), t.end());
    return (s == t ? true : false);
}
```

2.2.2 O(n) time O(1) space

Consider the following solution

```cpp
bool isAnagram(string s, string t) {
    if (s.size() != t.size()) return false;

    vector<int> freq(26, 0);

    for (int i=0; i<(int)s.size(); ++i) {
        ++freq[s[i] - 'a'];
        --freq[t[i] - 'a'];
    }

    for (const auto& item : freq) {
        if (item != 0 ) return false;
    }
    return true;
}
```

Create a vector of zeros with size 26. If s and t differ in size, return false. Otherwise, run an index loop with either the size of s or the size t (at this point we know they are the same). For each character c in s, index the frequency vector with c's ASCII value and increment by one. For each character k in t, index frequency and decrease by one. If all elements in the frequency vector are zero, the strings are anagrams.

Looping through each character in the strings is $O(n)$ time, where $n = |s| = |t|$. Looping through the freq vector is $O(26) = O(1)$ time. Thus, the total running time is

$$O(n) + O(1) = O(n)$$

The space complexity is $O(26) = O(1)$

Two Sum

3.1 $O(n^2)$ time $O(1)$ space

```cpp
#include <ranges>
class Solution {
    public:
    vector<int> twoSum(vector<int>& nums, int target) {
        for (const auto& i :
↪   std::ranges::views::iota(0,(int)nums.size())) {
            for (const auto& j :
↪   std::ranges::views::iota(i+1,(int)nums.size())) {
                if (nums[i] + nums[j] == target) return {i,j};
            }
        }
        return {};
    }
};
```

3.2 O(n) time O(1) space

We can iterate through the array once, and for each element, check if the target minus the current element exists in the hash table. If it does, we have found a valid pair of numbers. If not, we add the current element to the hash table.

```cpp
class Solution {
    public:
    vector<int> twoSum(vector<int>& nums, int target) {
        int n = nums.size();
        unordered_map<int,int> hm;
        for (const auto& i : iota(0,n)) {
            int comp = target - nums[i];
            if (hm.find(comp) != hm.end()) {
                return {i, hm[comp]};
            }
            hm[nums[i]] = i;
        }
        return {};
    }
};
```

On average, the unordered_map.find() method is $O(1)$. Looping through each element in the passed vector is $O(n)$. Thus, the total time is

$$nO(1) = O(n)$$

In the worst case, finding an element in the map takes $O(n)$ time, and the total time would therefore be

$$nO(n) = O(n^2)$$

3.3 Grouped anagrams

3.4 $O(n \cdot k \lg k)$ time $O(n \cdot k)$ space

Given an array of strings strs, group the anagrams together. You can return the answer in any order.

```cpp
template <typename T, typename U>
using umap =  unordered_map<T,U>;

using namespace std::ranges::views;

class Solution {
    public:
    vector<vector<string>> groupAnagrams(vector<string>& strs) {
        umap<string, vector<string>> m;

        for (const auto& i :
    ↪ std::ranges::views::iota(0,(int)strs.size())) {
            string s = strs[i];
            std::sort(s.begin(), s.end());

            m[s].push_back(strs[i]);
        }

        vector<vector<string>> ret;
        for (const auto& [key, value] : m) {
            ret.push_back(value);
        }
        return ret;
    }
};
```

The time taken by the first loop is $nO(k \lg k) = O(n \cdot k \lg k)$. In each iterator of the loop, a string is sorted. Each string is sorted in $O(k \lg k)$ time, where k is the average length of a string. The second for loop takes $O(n)$ time, because the umap is at most size n, where n is the number of strings in the passed vector. Thus, the total time is

$$O(n \cdot k \lg k) + O(n) = O(n \cdot k \lg k)$$

The space required in the map depends on the space required for the keys and the space required for the values. Regarding the keys, the space is $O(n \cdot k)$, where k is the average length of a string. The space required to store the vector of strings is again $O(n \cdot k)$. In the first loop, The string s is a temporary variable holding the sorted version of a string during each iteration. The maximum space required at any time for this is $O(k)$

The output vector ret stores the same strings as the input, grouped into vectors. This effectively duplicates the storage of the input in the worst case. The total space complexity is therefore

$$O(n \cdot k)$$