god-stack-dsa-problem-solving

> "Data Structures and Algorithms (DSA) should be viewed as essential tools, akin to the finely tuned parts of a Formula 1 car. The act of problem-solving with DSA serves as a crucial platform to exhibit both intelligence and creative thinking. The coding challenges themselves are simply various permutations of external factors; like the weather, track, wind, and rain in an F1 race. Ultimately, what dictates success in both domains; coding and Formula 1; is the mastery of planning, strategizing, maintaining flow, and ensuring precise code orchestration." - Siddhant Bali

- Author: Kintsugi-Programmer

> Disclaimer: The content presented here is a curated blend of my personal learning journey, experiences, open-source documentation, and invaluable knowledge gained from diverse sources. I do not claim sole ownership over all the material; this is a community-driven effort to learn, share, and grow together.

Table of Contents

- god-stack-dsa-problem-solving
  - Table of Contents
  - About
  - Array & Hashing
    - 1 Concatenation of Array [Easy]
    - 2 Contains Duplicate [Easy]
    - 3 Valid Anagram [Easy]
    - 4 Two Sum [Easy]
    - 5 Group Anagrams [Medium]
    - 6 Top K Frequent Elements [Medium]
- Template
  - Topic
    - Sr.No. Question [Easy/Medium/Hard]

About

DSA LeetCode Mastery for Problem Solving Abilities. https://github.com/kintsugi-programmer/god-stack

File Structure

```
README.md - Notes
[n].cpp - Code For Prob.Sol.
pr[n].cpp - Code For Pre.Req.
qn[n].cpp - Code For Ques.Notes
rough[n].cpp - Rough Code Notebooks
```

TARGETS

- TARGET 1:
  - https://neetcode.io/practice?tab=blind75 **[ON-SIGHT]**
  - DSA Basics to Intermediate **[ON-SIGHT]**
  - CPP_MASTERY **[ON-SIGHT]**
  - CPP_STL_MASTERY **[ON-SIGHT]**
- TARGET 2:
  - DSA Advanced
  - https://neetcode.io/practice?tab=neetcode150
- TARGET 3: https://neetcode.io/practice?tab=neetcode250
- TARGET ∞: https://neetcode.io/practice?tab=allNC

RAM(Random-access memory)

- Data Structures and RAM
  - A data structure is a way of structuring data.
  - In computer science, data structures involve structuring data inside of RAM (Random Access Memory).
  - RAM is where all variables are stored.
  - Example: An array containing the numbers 1, 3, and 5 is information stored in RAM.
- RAM Measurement and Composition
  - RAM is measured in bytes.
  - It is common for many computers to have 8 gigabytes of RAM.
  - Giga means about 10^9, or approximately a billion.
  - A byte is eight bits.
  - A bit can be thought of as a position that stores a digit.
  - The restriction on the digit stored in a bit is that it can only be a zero or a one.
  - Zeros and ones are the language of computers.
  - Storage Hierarchy: Individual bits form multiple bits, which form bytes, which ultimately form RAM.
  - RAM is used to store advanced data structures.
- Storing Values and Addressing in RAM

```
[ Array storage in RAM ]


  Array: [ 1 | 3 | 5 ]
            |   |   |
    +-------+   |   +-------+
    |           |           |
    V           V           V
  +-----------+-----+-----+-----+----------+
  | Value     |  1  |  3  |  5  |   ...    |
  +-----------+-----+-----+-----+----------+
  | Address   | $0  | $4  | $8  |  $12 ... |
  +-----------+-----+-----+-----+----------+
```

  - RAM can be conceptually viewed as a contiguous block of data.

        - i.e. Nothing Can be B/w 2 adjacent blocks
    - RAM has two components: values and addresses.
    - Values are the data being stored.
    - Every value is stored at a distinct location called an address.
    - In representations, a dollar sign ($) may be used in front of every address to distinguish it from the stored values.
    - Example: The first address might be $0.
- Representation of Integers in RAM
    - It is common for integers to be represented by four bytes, not a single byte.
    - Four bytes is equivalent to 32 bits.
    - Example of storing the integer '1' using 32 bits:
        - The representation will be 31 zeros, followed by a single one at the end.
    - The process involves taking the value, representing it in terms of bytes, and then placing that representation into RAM.

PreReqs Overview: Fundamental Data Structures and Algorithms

- Introduction and Primary Focus
    - This PreReqs focuses on teaching all the **fundamental data structures and algorithms** that individuals need to know. There is a specific emphasis on preparing for **coding interviews**.
- Target Audience
    - This curriculum is suitable for two main groups:
        - 1. **Beginners**.
        - 2. Anyone who **just needs a refresher**.
- PreReqs Content and Scope
    - The instruction will cover all the common data structures and algorithms. The discussion will include detailed analysis of four key areas related to these structures and algorithms:
        - 1. **Design**
        - 2. **Implementation**
        - 3. **Tradeoffs**
        - 4. **Analysis**
    - Career Impact and Compensation Value
        - Mastering the skills taught in this PreReqs is critical because the ability to perform efficiently can significantly impact compensation. This difference in compensation can amount to **hundreds of thousands of dollars**.
    - The core competencies that yield this high value include:
        - Solving these problems **efficiently**.
        - **Analyzing** them.
        - Discussing their **tradeoffs**.
        - **Communicating the idea** to others.
    - The goal is that the problem solving skills acquired in this course will serve the participant for their **entire career**.
- Next Steps
    - For participants seeking **more details on what is going to be covered**, they should **scroll down**. Once participants are ready, they are encouraged to **get started**.

Array & Hashing

TotalCompanyTags

> Baidu, Airbnb, Netease, Cisco, Amazon, Aetion, Box, Mathworks, Zoom, Google, Cloudera, Intel, Indeed, Godaddy, Walmart Global Tech, Salesforce, Didi, Affirm, Vmware, Yandex, Microsoft, Adobe, Alibaba, Jpmorgan, Linkedin, Citadel, Emc, Groupon, Intuit, Twitter, Nvidia, Twilio, Valve, Expedia, Yahoo, Zoho, Bookingcom, Wish, Zillow, Morgan-stanley, Drawbridge, Paypal, Huawei, Dropbox, Radius, Zomato, Roblox, Accenture, Goldman-sachs, Lyft, Yelp, Splunk, Bloomberg, Samsung, Bytedance, Servicenow, Quora, Goldman Sachs, Blackrock, Ebay, Ge-digital, Oracle, Qualcomm, Tencent, Uber, Tableau, Spotify, Morgan Stanley, American Express, Sap, Ibm, Deutsche-bank, Snapchat, Dell, Apple, Visa, Works-applications, Facebook, Factset, Audible, Google, Adobe, Facebook, Twilio, Salesforce, Affirm, Docusign, Yahoo, Cisco, Servicenow, Blackrock, Goldman Sachs, Ebay, Vmware, Tiktok, Bookingcom, Electronic-arts, Amazon, Wish, Microsoft, Yandex, Oracle, Qualtrics, Bloomberg, Adobe, Alation, Uber, Nutanix, Jpmorgan, Tesla, Mathworks, Zulily, Google, Hulu, Ibm, Snapchat, Apple, Intuit, Visa, Goldman-sachs, Yelp, Facebook, Walmart Global Tech, Bytedance, Yahoo, Cisco, Vmware, Ebay, Pocket-gems, Amazon, Microsoft, Oracle, Adobe, Uber, Spotify, Google, Linkedin, Hulu, Snapchat, Apple, Goldman-sachs, Yelp, Facebook, Bloomberg

---

PreReqs

- Arrays
  - Arrays are considered the most simple data structure.
  - **Definition/Key Property:** Arrays are always contiguous.
  - Contiguous storage means that nothing is stored in between the values of the array (e.g., nothing between the 3 and the 5).
  - Arrays look exactly the same in memory as they are represented conceptually: a contiguous set of values.
  - When storing an array in RAM, the starting location (address) is not always chosen by the user.
  - Arrays can store different types of values, such as integers or characters.
- Array Storage and Address Incrementing (The Size Rule)
  - Values are stored contiguously regardless of how large or small they are.
  - The address must be incremented by the size of the value being stored.
  - **Example: Storing 32-bit Integers (4 bytes)**

```
Array: [ 1 | 3 | 5 ]
           |   |   |
    +------+   |   +-------+
    |          |           |
    V          V           V
  +-----------+-----+-----+-----+----------+
  | Value     |  1  |  3  |  5  |   ...    |
  +-----------+-----+-----+-----+----------+
  | Address   | $0  | $4  | $8  |  $12 ... |
  +-----------+-----+-----+-----+----------+
```

    - Each integer takes four bytes to store.

    - If the address stored the value '1' starting at address $0.
    - It uses four bytes ($0, $1, $2, $3).
    - The next value ('3') must be stored starting at address $4 (incremented by four).
    - The final value ('5') would be stored starting at address $8.
    - If each value only took one byte, the addresses would increment by one ($0, $1, $2), but 32-bit integers require four bytes.

  - **Example: Storing Characters (1 byte)**

```
Array: [ a | b | c ]
            |   |   |
     +------+   |   +-------+
     |          |           |
     V          V           V
   +-----------+-----+-----+-----+----------+
   | Value     |  a  |  b  |  c  |   ...    |
   +-----------+-----+-----+-----+----------+
   | Address   | $0  | $1  | $2  |   $3 ... |
   +-----------+-----+-----+-----+----------+
```

    - Characters (like A, B, C) are stored continuously.
    - Characters typically take only one byte to store in memory.
    - This one-byte size is typical when storing ASKY characters, although knowing the specific encoding (ASKY) is not super important.
    - If the first character 'A' starts at address $0.
    - The next address is incremented by one: 'B' is stored at address $1.
    - The next character 'C' is stored at address $2.

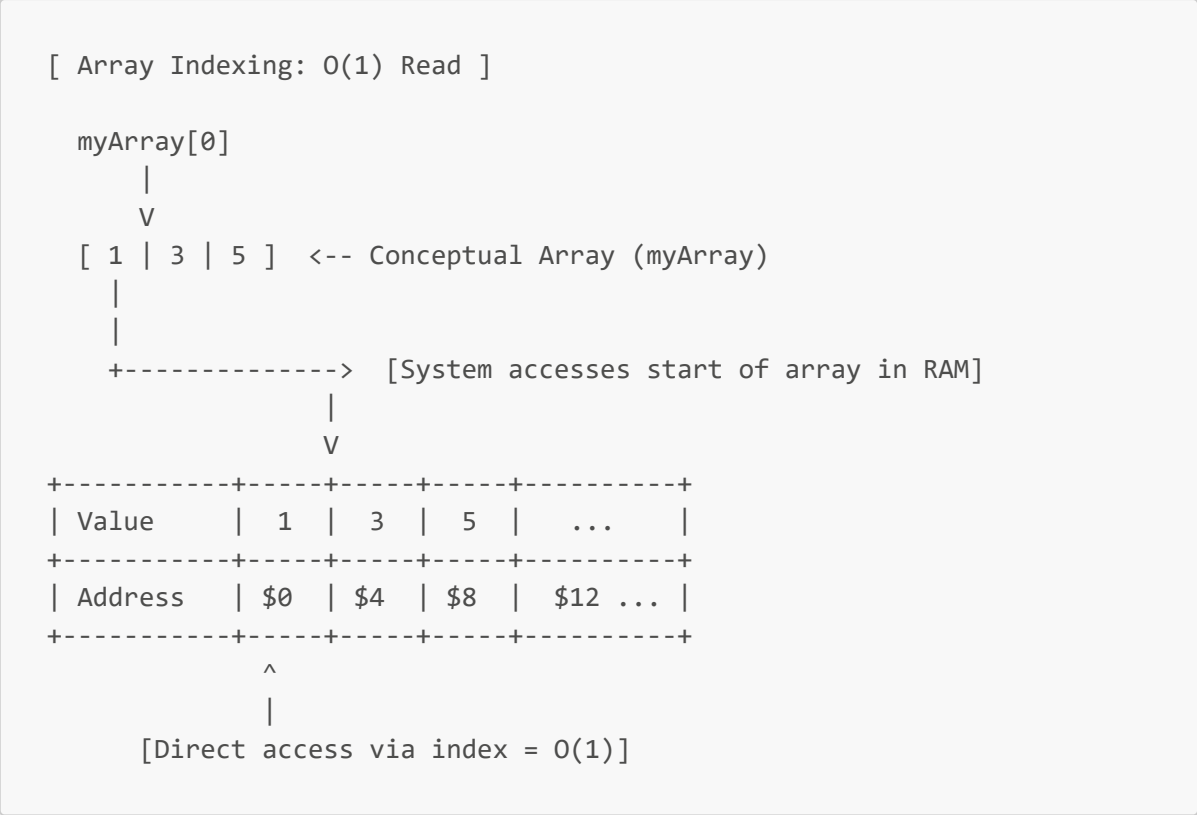
- Course Focus
  - The information presented covers theoretical aspects of memory and how arrays are stored.
  - This course will mostly focus on practical knowledge.
  - Upcoming topics will cover array properties, usage, and tradeoffs.
- ARRAY PROPERTIES AND OPERATIONS (STATIC ARRAYS)

```
+---------------------+--------------+
|      Operation      | Big-O Time   |
+---------------------+--------------+
| r / w i-th element  |     O(1)     |
| Insert / Remove End |     O(1)     |
| Insert Middle       |     O(n)     |
| Remove Middle       |     O(n)     |
+---------------------+--------------+
```

  - Array Definition and Structure
    - An array is defined as a **contiguous block of data**.
    - Arrays are stored in RAM (Random Access Memory) as a contiguous block of data.
  - Common Array Operations

- The two most common operations for any data structure are reading to the data or writing to the data.

- I. Reading Data from Arrays



  - Indexing
    - Programmers use indexes to access values in an array.
    - The first value is **always at index zero**, not index one.
    - Subsequent values are at index one, two, etc..
  - Reading Mechanism
    - To read the first element of an array (e.g., `my array`), the intuitive action is to go to the first address in memory and read the value.
    - Under the hood, this is exactly what happens.
    - Programmers do not need to know the exact address of every single value.
    - When reading the value at index zero, the system automatically goes to that location in memory and reads the stored value.
  - Efficiency of Reading
    - Since any index in the array can be automatically mapped to a location in memory, any value can be read **instantly** if the index is available.
    - This efficient operation is represented by **big O of one (O(1))**.
    - O(1) means that in the worst case, reading a value (assuming the index is known, e.g., index 2) is an **instant operation**.
    - Reading happens in **constant time**, regardless of how large the array becomes.
  - Role of RAM
    - The ability to go to any arbitrary address in RAM and read the value is a property of RAM.
    - **RAM stands for Random Access Memory**.
    - Random access means memory can be accessed randomly in constant time.

        - We do not have to start from the beginning of RAM and keep looking for a value; we can access it automatically.
        - This is important because if there is a lot of RAM, we avoid going through the entire RAM every time.
        - If accessing the third value, we do not have to go through the first and second values.
    - Looping Through Arrays (Reading All Values)
        - After reading the first value, we read the second, third, and continue until the end of the array.
        - In code, reading a value at index `i` is typically represented by `my array at index i`.
        - The index `i` is incremented (to one, then two, etc.).
        - The loop stops when `i` equals the length of the array.
        - This process loops through the entire input array.
        - In most languages, this can be done using a `for loop` or a `while loop`.
- II. Writing Data and Limitations of Static Arrays
    - Fixed Size Property
        - Arrays possess the property of **fixed size**.
        - Example: If an array was declared to hold three values, it is fixed size.
    - The Contiguity Constraint for Adding Values
        - If we want to add a fourth value (e.g., a seven) to an array of size three, the new value must be added to maintain contiguity.
        - Example: If the existing data ends just before address 12, the seven would need to be added precisely at address 12 to maintain a contiguous array of size four.
        - Limitations in RAM allocation prevent simple expansion.
            - We do not always get to decide exactly where RAM is allocated or where values are stored.
            - The required spot in RAM might already be used by another array or the operating system.
        - If a value is put anywhere else in memory, the array loses its contiguous property, resulting in two separate arrays, which is not the intended outcome.
        - Loss of Contiguity Breaks Operations
            - If contiguity is lost, looping through the array breaks.
            - Example: If a loop is at index two and increments to index three, it will attempt to access the expected contiguous spot; if the value is stored elsewhere, the operation will fail.
    - Static Arrays Defined
        - This fixed-size constraint is the biggest limitation of **static arrays**.
        - Static arrays are defined as **fixed size arrays**.
        - **Recommendation is to first define upperbound static arrays with initialisation with zeros or use vector array**
    - Static vs. Dynamic Arrays
        - Programming languages like Python or JavaScript often use **dynamic arrays** by default, which means programmers rarely encounter the fixed-size limitation or worry about running out of space.
    - Initialization

        - When a static array (e.g., size three) is allocated, there must be something stored in memory.
        - Default initialization varies:
            - Languages usually initialize values to all zero.
            - Sometimes they do not initialize them at all, resulting in random arbitrary values.
    - III. Overwriting and Efficient Operations (O(1))
        - Writing/Adding at the End
            - Adding a new value to the end of the array (at the next empty spot being tracked) is an **instant operation**.
            - We know the index (e.g., index 2) and thus instantly know the position in RAM.
            - **Writing to any position** is a **big O of one (O(1)) operation**; it is constant time.
            - Inserting a value at the end of the array is also a **big O of one (O(1))** time operation, assuming an empty space is available.
        - Removing/Overwriting
            - When "removing" a value in a static array, we cannot actually delete or deallocate it from memory.
            - Removal means **overwriting** the value (e.g., replacing it with a zero).
            - We effectively declare that the original value is "no longer relevant," even though it remains in memory.
            - **Removing a value** is also efficient: **big O of one (O(1))**.
            - It is a constant time operation because it involves going to that memory position and overwriting it (e.g., replacing it with a zero or negative 1).
            - Removing the value at the end of the array is a **big O of one (O(1))** time operation, assuming at least one element exists to remove.
        - Summary of Efficient Operations
            - Reading or overwriting the arbitrary element `E` (meaning any index) is done in **big O of one (O(1))** time, or constant time.
    - IV. Arbitrary Insertion and Removal (Inefficient Operations: O(N))
        - Ordered Values
            - In arrays, including static arrays, the **order of the values matters**.
            - Values are ordered because of the way indexes work (e.g., five comes before six).
        - Inserting into the Middle or Beginning
            - Inserting at an arbitrary position (middle or beginning) is **not efficient**.
            - If we simply overwrite an existing value to insert a new one, the desired order is lost (e.g., replacing five with four results in four, six, not four, five, six).
        - The Shifting Mechanism for Insertion
            - To insert a value (e.g., four) while maintaining order (e.g., four, five, six), the existing elements must be **shifted to the right**.
            - **Step 1: Shift existing values.** We must shift the elements starting from the end towards the right.
                - Example: Before shifting the five, the six must be shifted to prevent loss of data.
                - Six (at index 1) moves to index 1 + 1 (index 2).
                - Five (at index 0) moves to index 0 + 1 (index 1).

        - **Step 2: Insert the new value.** Only after shifting is completed can the new value (four) be placed in the now-available position (index 0).
        - **Downside:** This requires moving every existing value in the array.
    - Insertion Time Complexity (Worst Case)
        - Shifting many elements is **not very efficient**.
        - This operation is called **big O of N (O(N))**.
        - **N** refers to the **number of elements** in the array (the length), not the size of the array.
        - **Big O** notation refers to the **worst case** scenario.
        - **Worst Case for Insertion:** If inserting a value at the beginning of a filled array, we must shift every value in the array (N values) over to the right by one.
        - **Worst Case Time Complexity:** Inserting into the middle of an array is **big O of N**.
            - Note: While inserting into an empty array would be O(1), the worst case determines the Big O notation.
        - Example of a better case: If inserting between five and six to achieve five, four, six, only the six would need to be shifted (one value), not every value. However, the generalized worst case is still O(N).
    - Arbitrary Removal Time Complexity
        - Removing a value at an arbitrary index requires similar shifting.
        - Arbitrary removal means we want to act as if the value no longer exists, not just replace it with a zero.
    - The Shifting Mechanism for Removal
        - Removal requires shifting values to the **left**.
        - Example: If removing the value at index 0 from (5, 6, 7), the goal is to shift the 6 and 7 forward.
            - Six (at index 1) moves to index 1 - 1 (index 0).
            - Seven (at index 2) moves to index 2 - 1 (index 1).
        - Memory Note: The memory allocated for the element that was removed still exists and is reflected in RAM, even if the resulting array (6, 7) appears smaller.
        - **Worst Case Time Complexity:** Removing in the middle of an array is **big O of N (O(N))**, as we might have to shift every value.
- ARRAY PROPERTIES AND OPERATIONS (STATIC ARRAYS) CODE

```
#include <iostream>

using std::cout;
using std::endl;

// Insert n into arr at the next open position.
// Length is the number of 'real' values in arr, and capacity
// is the size (aka memory allocated for the fixed size array).
void insertEnd(int arr[], int n, int length, int capacity) {
    if (length < capacity) {
        arr[length] = n;
    }
}

// Remove from the last position in the array if the array
```

```
// is not empty (i.e. length is non-zero).
void removeEnd(int arr[], int length) {
    if (length > 0) {
        arr[length - 1] = 0;
    }
}

// Insert n into index i after shifting elements to the right.
// Assuming i is a valid index and arr is not full.
void insertMiddle(int arr[], int i, int n, int length) {
    for (int index = length - 1; index >= i; index--) {
        arr[index + 1] = arr[index];
    }
    arr[i] = n;
}

// Remove value at index i before shifting elements to the left.
// Assuming i is a valid index.
void removeMiddle(int arr[], int i, int length) {
    for (int index = i + 1; index < length; index++) {
        arr[index - 1] = arr[index];
    }
}

void printArr(int arr[], int capacity) {
    for (int i = 0; i < capacity; i++) {
        cout << arr[i] << ' ';
    }
    cout << endl;
}
```

QuesNotes

- Read Question Multiple times pls

```
    // Read Question Multiple times pls
    // Read Question Multiple times pls
```

```
    // Read Question Multiple times pls
    // return {}; // No solution found, return an empty vector
    // else, this error comes: Line 10: Char 5: error: non-void function does not
return a value in all control paths [-Werror,-Wreturn-type] 10 | }^1 error
generated.
```

- STL Lib & namespace std

```

```

```
    // #include<bits/stdc++.h> // STL Lib
    // using namespace std;
```

- Vector Array

```
    // Vector Array
    // a way to store and access the list of elements.
    // dynamic array in C++ is std::vector
    // Why It's Useful: You don't need to know the exact number of elements you'll
be storing ahead of time
    // #include <vector> // lib
    std::vector<int> arr1; // empty vector
    std::vector<int> arr2={1,2,3}; // initialised vector
    std::vector<int> arr3(5); // size =5, each element value=default value=0
    std::cout<<arr3[0]<<std::endl; //0 // Direct Access (0based)
    std::vector<int> arr4(5,90); // size =5, each element value=90
    std::cout<<arr4[0]<<std::endl; //90
    std::cout<<arr1.size()<<std::endl; //0 // Get Length
    arr4.push_back(5); // Append // btw each push in dynamic array is O(1)
    std::cout<<arr4[5]<<std::endl; //5
    for ( int i =0 ; i<arr2.size(); i++) { std::cout<<arr2[i]<<"    ";} // Vector
array traversal
    std::cout<<std::endl;
    // std::vector it's just a dynamic array that you can access with integer
indices (like tempo[0], tempo[1], etc.). It doesn't have a .find() method that
takes a string, nor can you use a string inside the square brackets [].
    // The perfect tool for this is a std::unordered_map. It allows you to store
key-value pairs, which is exactly what you need: the sorted string as the key and
the list of anagrams as the value.
```

- Bucket/2D Vector

```
    // std::vector<std::vector<int>> (Bucket/2D Vector)
    // What It Is: This is simply a vector where each element is another vector.
You can visualize it as a 2D grid or a list of lists.
    // Why It's Useful: It's great for grouping items into "buckets."
    //      For instance, you could have a main vector where the index represents
a frequency.
    //      The element at my_buckets[5] would then be a vector containing all the
numbers that appeared 5 times.
    //      This technique, often called Bucket Sort, is a powerful way to
organize data by a specific property (like frequency).
    // Create a vector of 5 "buckets", where each bucket is an empty vector of
ints.
    int num_buck =5 ;
    vector < vector<int> > buckets(num_buck);
    // Let's put numbers into buckets based on some property.
```

```
    // For example, put numbers 10 and 11 into bucket 2.
    buckets[2].push_back(10);
    buckets[2].push_back(11);
    // Put number 25 into bucket 4.
    buckets[4].push_back(25);
    // Now, let's see what's in bucket 2.
    std::cout<< "Items in bucket 2: ";
    for (int items: buckets[2]) // buckets[2], not buckets
    {
        std::cout << items << " "; // Output: 10 11
    }
    std::cout << std::endl;
    // Items in bucket 2: 10 11
```

- For Loop

```
    // For Loop
    for ( int i=0; i<5; i++) std::cout<<i;
    // 01234
```

- Range-Based for Loop

```
    // Range-Based for Loop
    // What It Is: A cleaner, more modern syntax for a for loop that iterates through all the elements of a container (like a vector or map) without needing to manage indices or iterators manually.
    // Why It's Useful: It makes code much easier to read and write, and it reduces the chance of off-by-one errors with indices. You just declare a variable that will take the value of each element in the container, one by one.
    vector<int> numbers = { 1,2,3 };
    cout<< "nos:";
    // => 'num' will take the value of each element in 'numbers'.
    for (int num: numbers){
        cout<<num<<" "; // 1 2 3
    }
    cout<<endl;
    // nos:1 2 3
    // => It works with maps, too. 'const auto&' is efficient here.
    unordered_map<char, int> meow = { {'a',3}, {'b',5 }}; // "c" NO CHAR, 'c' YES CHAR
    cout<< "nos:";
    for ( const auto& pair : meow) {cout<<"{"<<pair.first<<","<<pair.second<<"}";}
// 'pair' is a key-value pair.
    cout<<endl;
    // nos:{b,5}{a,3}
```

- Hashset

```
    // Hashset
    std::unordered_set<int> hashSet;
    hashSet.insert(1);
    std::cout<<(
        hashSet.find(1)
        !=
        hashSet.end() // i.e if find give hashSet.end() ptr, then element doesnt
exist
    )<<std::endl; // true as 1 exists in hashset
    // 1
    // re-insert array elements in hashmap & checker while insertion: number not
existed in hashmap tip
    //  // given: vector<int>& nums
    //     unordered_set<int> hashbrown; // re-insert array elements in hashmap
    //     for (int i=0; i<nums.size(); i++){
    //         if ( hashbrown.find(nums[i]) != hashbrown.end() ) // checker while
insertion: number not existed in hashmap
    //         {
    //         }
    //     }
    // ( hashbrown.find(nums[i] ) only returns ptr, not index, tip
```

- Sorting

```
    // Sorting
    // #include <algorithm> // Required for std::sort

    // Sorting a string
    std::string word = "cab";
    std::sort(word.begin(), word.end());
    std::cout << "Sorted word: " << word << std::endl;
    // Sorted word: abc

    // std::sort with Reverse Iterators
    std::vector<int> data = {40, 10, 50, 20, 30};

    // 1. Sort in default ascending order
    std::sort(data.begin(), data.end());
    std::cout << "Ascending: ";
    for (int x : data) std::cout << x << " "; // Output: 10 20 30 40 50
    std::cout << std::endl;
    // Ascending: 10 20 30 40 50

    // 2. Sort in descending order using reverse iterators
    sort(data.rbegin(), data.rend());
    std::cout << "Descending: ";
    for (int x : data) std::cout << x << " "; // Output: 50 40 30 20 10
    std::cout << std::endl;
    // Descending: 50 40 30 20 10
```

```
    // Sorting in Vector
    // (TC): O(nlogn) on average and in the worst case
    // (SC): O(logn)
    // std::sort sorts the vector in-place and does not return a sorted vector.
Its return type is void
    std::vector<int> v1 = {1,4,2,0};
    std::cout<<"v1: "<<v1[0]<<" "<< v1[1] << " " << v1[2] << " " << v1[3] <<
std::endl;
    // v1: 1 4 2 0
    std::sort(v1.begin(), v1.end()); // asc default
    std::cout<<"v1(sorted): "<<v1[0]<<" "<< v1[1] << " " << v1[2] << " " << v1[3]
<< std::endl;
    // v1(sorted): 0 1 2 4
    std::sort(v1.begin(), v1.end(), std::greater<int>()); // dsc
    std::cout<<"v1(sorted dsc): "<<v1[0]<<" "<< v1[1] << " " << v1[2] << " " <<
v1[3] << std::endl;
    // v1(sorted dsc): 4 2 1 0
    // - Downside of This STL Sort
    //     - Differs from lang to lang
    //     - good sort algos do O(nlogn)
    //     - bad sort algos do O(n**2)
    //     - uncertain of SC
    //     - uncertain of TC
    //     - lenghty at cases where STL is not available
```

- unordered sets

```
    // unordered sets in c++ contains unique elements
    // if we even insert duplicate, it will reject it
    // we can make unord set with method to copy full the vector array, rejecting
the duplicates
    v1.push_back(1);
    v1.push_back(1);
    std::unordered_set<int> hashSet2(v1.begin(), v1.end()) ;// TC: O(n) avg tc,
O(n**2) worst rare tc where glitches happen &SC: O(k) Avg, O(n) worst, k is number
of unique elements
    std::cout<< hashSet2.size() << std::endl; // 4 // unique elements in unord set
    std::cout<< v1.size() << std::endl; // 6 // duplicates elements in vect arr
```

- string equality

```
    string s = "aba" ;
    string t = "aab" ;
    // string equality
    cout<<
    ( s==t ) // SC: O(min(N,M)), TC: O(1)
    << endl; // 0
    // cout<<s==t<< endl; // NO, always use explicit brackets to avoid glitches
    sort(s.begin(),s.end());
```

```
    sort(t.begin(),t.end());
    cout<<( s==t )<< endl; // 1
```

- Storing Character Frequencies

```
    // Storing Character Frequencies
    // Storing Character Frequencies Way 1: Dynamic Array/ Vector
    vector<int> counts(26,0); // Creates a vector of size 26, with all elements
initialized to 0.
    // Since the problem states the strings only contain lowercase English
letters, you don't need a flexible map. A simple array of size 26 is faster.
    // Here idx 0 -> 'a' -> 97
    // 25 -> 'z' -> 122
    char ch = 'a';
    counts[ch-97]++;
    counts[ch-97]++;
    cout<<counts[0]<<endl; //2

    // Storing Character Frequencies Way 2: Hashmap/ Unordered Map
    unordered_map<char,int> freq_map;
    freq_map['a']++;
    // If 'a' isn't in the map, it's added with value 1.
    // If it is, its value is incremented.
    // it has capability of direct comparison
        // freq_map == freq_map2 is valid
    string text = "hello world";
    unordered_map<char,int> char_freq;
    for ( char c: text){
        char_freq[c]++;
    }
    int charindex = 2;
    cout<< "Char "<<text[charindex]<<" comes "<<char_freq[text[charindex]]<<"
times\n";
    // Char l comes 3 times
```

- Hashmap/ Unordered Map

```
    // Hashmap/ Unordered Map
    // This is the most powerful tool.
    // An unordered_map (or hash map) is like a super-fast dictionary.
    // You give it a key and it stores an associated value.
    // Its superpower is checking if a key exists or retrieving its value in an
average of constant time, $O(1)$, which is incredibly fast.
    // Why It's Useful: It provides incredibly fast lookups, insertions, and
deletions of elements. If you know the key, you can find its value almost
instantly, no matter how many items are in the map. This is perfect for grouping
items or counting frequencies.
    // For a problem where you're looking for target - nums[i], a hash map is
```

```
perfect for instantly checking: "Have I seen the number I need before?"
    unordered_map<string,int> std_scores;
    std_scores["Bali"]=101;
    std_scores["Bali"]=100;
    cout<<std_scores["Bali"]<<endl;// 100


    std::vector<int> nums1 = {3, 4, 5, 6, 4, 5, 4};
    std::unordered_map<int,int> momos;

    // Application : freq count for array containing duplicates, and index find
for array containing unique elements

    // Application : freq count for array containing duplicates
    for (int i: nums1)
    // special iterator
    // here i is not i , it's actually nums[i] of traditional for loop
    // faster
    // but can only use nums[i](value only), not i (index)

    {
        momos[i]++ ;// If num is not in the map, it's added. Then its count is
incremented.
    }
    // Fast Lookups in hashmap
    int target =4;
    if ( momos.find(target) != momos.end()){
        std:: cout << target << "'s momo :"<< momos[target] << std::endl ;
    }
    // 4's momo :3

    // Application :index find for array containing unique elements
    std::unordered_map<int,int> momos2;
    for ( int i=0 ; i<nums1.size(); i++){
        momos2[nums1[i]]=i; // not momos2[nums[i]]++ as it dont make sense
        // or momos2.insert( { nums[i] , i } );
    }
```

- Character Arithmatics

```
    // Character Arithmatics
    // char types are internally represented as numbers (like ASCII values). This
allows you to perform math on them
    char c1 = 'a';
    char c2 = 'b';
    char c3 = 'c';
    int i1 = (c3 - c1); //2
    cout<<i1<<endl; //2
    // ASCII Values
    // 'a' = 97
    // 'b' = 98
    // 'y' = 121
```

```
    // 'z' = 122
    // 'A' = 65
    // 'B' = 66
    // 'Y' = 89
    // 'Z' = 90

    // Calculate the 0-based index for any character
    char mychar = 'e';
    int index = mychar - 'a';
    std::cout << "The character '" << mychar << "' has an index of: " << index <<
std::endl;
    // The character 'e' has an index of: 4

    // General Looping tip
    // for (int i=0; i<s.size(); i++){
    //     counts[s[i]-97]++;
    // }
    // for (int i=0; i<s.size(); i++){
    //     counts[t[i]-97]--;
    // }
    // // you can just put them in one loop na
    // for (int i=0; i<s.size(); i++){
    //     counts[s[i]-97]++;
    //     counts[t[i]-97]--;
    // }
    // // You may forget this while focusing on knowledge
```

- Pair

```
    // Pair
    // you need to link two pieces of information together.
    // For example, what if you need to sort the numbers but not lose their
original positions ?
    // A std::pair is perfect for this. It holds exactly two items, which you can
access with .first and .second.
    // #include <utility> // lib
    std::pair<std::string, int> student1= {"Bali",101}; // {} way in cpp
    std::pair<std::string, int> student2{"Bhati",100}; // classic way
    auto student3 = std::make_pair("Bhaskar",100); // way of std::make_pair()
helper function ,This is useful because the compiler can often figure out the
types for you.
    student1.second=100;
    std::cout<<student1.first<<"\t"<<student1.second<<std::endl;// Bali    100

    // Vector of Pairs
    // value_index_pairs_vector
    std::vector<std::pair<int,int>> pv ; // vector of value_index_pairs
    std::vector<int> nums = {15, 8, 22, 5}; // random vect array, given in problem
    for (int i=0; i <nums.size(); i++){ // Store each number with its original
index
        pv.push_back( { nums[i] , i } );
        // push_back() method
```

```
        // , with to push pairs: push_back({})
        // & our case :  push_back({value,index})
    }
    std::sort(pv.begin(),pv.end()); // Sort the pairs. By default, it sorts by the
first element (the value).
    for (int i=0; i<pv.size(); i++){
        std::cout<<pv[i].first<<"<-"<<pv[i].second<<std::endl;
    }
    // 5<-3
    // 8<-1
    // 15<-0
    // 22<-2
```

- String

```
    // String
    // #include <string>
    // String Concatenation
    // std::to_string: A simple function that converts a number (like an int) into
its string representation.
    // String Concatenation: The process of joining two or more strings together
to form a new, single string. In C++, this is easily done with the + or +=
operator
    // Why They're Useful: You often need to build a string from different pieces
of data, including numbers. std::to_string allows you to seamlessly integrate
numbers into your strings.
    std::string greet = "Hello World";
    std::string name= "Bali";

    // 1. String Concatenation using '+'
    std::string message = greet + ", I am " + name + " !!!";
    cout<<message<<endl;
    // Hello World, I am Bali !!!

    // 2. Using std::to_string
    int version = 5;
    string appname = " KintsugiDev.Studio ver."+ to_string(version); // We must
convert the number '5' to a string before we can join it
    cout<< appname << endl;
    //  KintsugiDev.Studio ver.5

    // 3. Building a key from parts using '+='
    std::string key = "";
    key+="user";
    key+=":";
    key+="Bali";
    key+=",";
    key+="id";
    key+=":";
    key+=to_string(2022496);
    cout<<key<<endl;
    // user:Bali,id:2022496
```

- priority_queue (Heap)

```
    // std::priority_queue (Heap)
    // priority_queue is a container that organizes elements based on priority.
    // By default, it's a max-heap, meaning that whenever you look at the top()
element, it's always the largest one in the container.
    // When you pop() an element, the largest one is removed.
    // Why It's Useful: It's perfect for problems where you need to keep track of
the "top K" items without sorting the entire collection. For example, you can
maintain a priority queue of size k. As you process new items, you can compare
them to the smallest item in your "top K" set and replace it if the new item is
larger. This is much more efficient than sorting everything.
    // #include <queue>
    // 1. Default priority_queue (Max-Heap)
    priority_queue<int> max_heap;
    max_heap.push(10);
    max_heap.push(20);
    max_heap.push(30);
    // The top element is always the largest.
    cout<< "Largest of max_heap :"<<max_heap.top()<<endl; //30
    // Largest of max_heap :30
    // 2. Min-Heap (keeps the smallest element at the top)
    // You must provide extra template arguments to change its behavior.
    priority_queue<int, vector<int>, greater<int>> min_heap;
    min_heap.push(10);
    min_heap.push(20);
    min_heap.push(30);
    // Now, the top element is always the smallest.
    cout<< "Smallest of min_heap :"<<min_heap.top()<<endl; //10
    // Smallest of min_heap :10
```

---

1 Concatenation of Array [Easy]

- https://leetcode.com/problems/concatenation-of-array/

> Google, Adobe, Facebook

Ques

You are given an integer array nums of length n. Create an array ans of length 2n where ans[i] == nums[i] and ans[i + n] == nums[i] for 0 <= i < n (0-indexed).

Specifically, **ans is the concatenation of two nums arrays**.

Return the array ans.

Example 1:

```
Input: nums = [1,4,1,2]

Output: [1,4,1,2,1,4,1,2]
```

Example 2:

```
Input: nums = [22,21,20,1]

Output: [22,21,20,1,22,21,20,1]
```

Constraints:

- 1 <= nums.length <= 1000.
- 1 <= nums[i] <= 1000

Solutions

- Basically We have to make final array of 2x input array
- **We can't do just Vector Concatnation like strings**
- Solution 1
    - assume nums as input vector
    - n = size of vector
    - **make new arr1 of size 2n default 0**
    - for loop 0 to n-1
        - arr1[i]= nums[i] **(insert 1 to n)**
    - for loop 0 to n-1
        - arr1[i+n]=arr[i] **(insert n+1 to 2n)**
    - return arr1

```cpp
// Solution 1
class Solution {
public:
    vector<int> getConcatenation(vector<int>& nums) {
        int n = nums.size();
        vector<int> ans(2*n);
        for (int i=0; i<n; i++){
            ans[i]= nums[i];
        }
        for (int i=0; i<n; i++){
            ans[n+i]= nums[i];
        }
        return ans;
    }
```

```
};
// Time complexity:
// O(n)
// Space complexity:
// O(n) for the output array.
```

- Solution 2
  - **Iteration (One Pass): Same Insertion in New Array, but in 1 loop, 2ops per iteration**
  - assume nums as input vector
  - n = size of vector
  - make new arr1 of size 2n default 0
  - for loop 0 to n-1
    - **arr1[i]= nums[i]**
    - **arr1[i+n]=arr[i]**
  - return arr1

```
// Solution 2
class Solution {
public:
    vector<int> getConcatenation(vector<int>& nums) {
        int n = nums.size();
        vector<int> ans(2*n);
        for (int i=0; i<n; i++){
            ans[i]= nums[i];
            ans[i+n]= ans[i];
            // ans[i] = ans[i + n] = nums[i]; // we can write this too !!!

        }
        return ans;
    }
};
// Time complexity:
// O(n)
// Space complexity:
// O(n) for the output array.
```

- Solution 3
  - **just append in orginal array and return it**
  - **not making any new array ,saving space**
  - assume nums as input vector
  - n = size of vector
  - for loop 0 to n-1
    - nums.push_back(nums[i])
  - return nums
- **btw each push in dynamic array is O(1)**

```cpp
// Solution 3
class Solution {
public:
    vector<int> getConcatenation(vector<int>& nums) {
        int n = nums.size();

        for (int i=0; i<n; i++){
            nums.push_back(nums[i]);
        }

        return nums;
    }
};

// Time complexity:
// O(n)
// Space complexity:
// O(n) for the output array.
```

- Solution 4 **-- optimal**
  - Iteration (Two Pass)
  - **generic sol. with no. of times** var, here =2
  - assume nums as input vector
  - n = size of vector
  - for loop 1 to times-1
    - for loop 0 to n-1
      - nums.push_back(nums[i])
  - return nums
- btw each push in dynamic array is O(1)

```cpp
// Solution 4
class Solution {
public:
    vector<int> getConcatenation(vector<int>& nums) {
        int n = nums.size();
        int times= 2;
        for (int j=1;j<times; j++)// j is 1 because ones occurence is already here
        {
            for (int i=0; i<n; i++){
                nums.push_back(nums[i]);
            }
        }

        return nums;
    }
};

// Time complexity:
```

```
// O(n)
// Space complexity:
// O(n) for the output array.
```

---

2 Contains Duplicate [Easy]

- https://leetcode.com/problems/contains-duplicate/

> Airbnb, Amazon, Apple ,Microsoft, Tcs, Google, Yahoo, Oracle, Palantir-technologies, Adobe, Uber, Facebook, Bloomberg

---

Ques

Given an integer array nums, return true if any value appears more than once in the array, otherwise return false.

Example 1:

```
Input: nums = [1, 2, 3, 3]

Output: true
```

Example 2:

```
Input: nums = [1, 2, 3, 4]

Output: false
```

Constraints:

1 <= nums.length <= 105 -109 <= nums[i] <= 109

Recommended Time & Space Complexity You should aim for a solution with O(n) time and O(n) space, where n is the size of the input array.

Hint 1 A brute force solution would be to check every element against every other element in the array. This would be an O(n^2) solution. Can you think of a better way?

Hint 2 Is there a way to check if an element is a duplicate without comparing it to every other element? Maybe there's a data structure that is useful here.

Hint 3 We can use a hash data structure like a hash set or hash map to store elements we've already seen. This will allow us to check if an element is a duplicate in constant time.

Solutions

- So in this ques array
  - if any any element occurence >1 or **have duplicates**
    - **return true**
  - else
    - then the array have distinct elements
    - return false
- Solution 1 **-- brute force**
  - we are **checking each element to find it's same value by traversing each array eachtime**
  - given nums vect array
  - let n = nums vect array length
  - let counter = 0
  - if counter become more than 1 , i.e. item has multiple occurence
  - for i 0->n-1
    - for j 0->n-1
      - if nums[i]==nums[j] (not i==j)
        - counter++
        - **if counter>1**
          - **return true**
    - counter = 0 // reset counter for next elements turn
  - return false // at case where counter didnt inc from 1 , i.e. not even onces occurence

```
// Solution 1
class Solution {
public:
    bool containsDuplicate(vector<int>& nums) {
        int n = nums.size();
        int counter = 0;
        for ( int i=0; i<n; i++){
            for ( int j=0; j<n; j++){
                if (nums[i]==nums[j]){
                    counter++;
                    if (counter> 1){
                        return true;
                    }
                }
            }
            counter=0;
        }
        return false;
    }
};
// // Time & Space Complexity
// // Time complexity:
// // O(n**2)
// // Space complexity:
// // O(1)
```

- Solution 2 **-- optimal**
  - hashset
  - efficient tc O(n)
  - directly checking if element's occurence > 1
  - using another ds hash set
  - we are **inserting in hash set one by one** & **parallely checking if incoming element already exists in hash set**, if it already exists, so it mean ; at that point of time its duplicate is incoming
    - so this hashset contain duplicates
    - return true and exit, no need to continue, we got our ans
  - else try until any occurence is duplicate
  - if not true at any case and didn't exited earlier
    - return false
  - **now only 1 loop**, which is even just insertion in ds is only req.
  - thought:
    - i thought for an approach where we can remove that element in an array and still find if it's exist as duplicate or not.
    - NO
    - **this Solution similar acts , by not deleting it but checking during genesis of array**

```
// Solution 2
class Solution {
public:
    bool containsDuplicate(vector<int>& nums) {
        int n = nums.size();
        unordered_set<int> hashSet;
        for ( int i=0; i<n; i++){ // or // for (int num : nums) {
            if (hashSet.find(nums[i])!=hashSet.end()){ // or // if
(find.count(num)) {
                return true;
            }
            else {
                hashSet.insert(nums[i]);
            }
        }
        return false;
    }
};
// Time & Space Complexity
// Time complexity:
// O(n)
// Space complexity:
// O(n)
```

- Solution 3
  - **2 Pointer Approach & Sorting**
  - **no need of new space**
  - **sort the array**
  - O(nlogn)

    - then the duplicates would be adjacent to each other
    - even even once 2 adjacent elements are same, then we got our duplicate array proof
    - do **one loop** to just **check if arr[i]==arr[i+1]**
        - if true, return true
    - if loops ends without returning true;then array is distinct
        - return false

```
// Solution 3
class Solution {
public:
    bool containsDuplicate(vector<int>& nums) {
        int n = nums.size(); // or just use the method in loop too
        sort(nums.begin(), nums.end());
        for ( int i=0; i<n-1; i++){ // we took n-1 as to not going to case (where
i=n-1 index(last index), i+1=n index(exceeding limit)), and now at i= n-2 (2nd
last element index), then i+1 = n-1(last element index) :)
            if( nums[i]==nums[i+1]) return true;
        }
        return false;

    }
};
// Time & Space Complexity
// Time complexity:
// O(nlogn)
// Space complexity:
// O(1) or O(n) depending on the sorting algorithm.
```

- Solution 4 **-- optimal**
    - Hash Set Length
    - we can just **make set of distinct elements and compare size of old array**, if same, then false, else true
    - **unordered sets in c++ contains unique elements**
        - if we even insert duplicate, it will reject it
    - we can make unord set with method to copy full the vector array, rejecting the duplicates
    - if set size < vector size, it means that vec had duplicate elements
        - return true
    - else return false

```
// Solution 4
class Solution {
public:
    bool containsDuplicate(vector<int>& nums) {
        return (
            unordered_set<int>(nums.begin(), nums.end())
            .size()
            <
            nums.size()
```

```
        );

    }
};
// Time & Space Complexity
// Time complexity:
// O(n)
// Space complexity:
// O(n)
```

---

3 Valid Anagram [Easy]

- https://leetcode.com/problems/valid-anagram
- Expedia, Affirm, Docusign, Yahoo, Cisco, Servicenow, Goldman Sachs, Amazon, Microsoft, Oracle, Morgan-stanley, Uber, Spotify, Zulily, Google, Paypal, Snapchat, Apple, Goldman-sachs, Yelp, Facebook, Bloomberg

---

Ques

Given two strings s and t, return true if the two strings are anagrams of each other, otherwise return false.

An anagram is a string that contains the exact same characters as another string, but the order of the characters can be different.

> Anagram : Multiple String Arrays where each have
>
> 1. Same Elements
> 2. Same Length
> 3. Different Order

Example 1:

```
Input: s = "racecar", t = "carrace"

Output: true
```

Example 2:

```
Input: s = "jar", t = "jam"

Output: false
```

Constraints:

```
s and t consist of lowercase English letters.
```

Recommended Time & Space Complexity You should aim for a solution with O(n + m) time and O(1) space, where n is the length of the string s and m is the length of the string t.

Hint 1 A brute force solution would be to sort the given strings and check for their equality. This would be an O(nlogn + mlogm) solution. Though this solution is acceptable, can you think of a better way without sorting the given strings?

Hint 2 By the definition of the anagram, we can rearrange the characters. Does the order of characters matter in both the strings? Then what matters?

Hint 3 We can just consider maintaining the frequency of each character. We can do this by having two separate hash tables for the two strings. Then, we can check whether the frequency of each character in string s is equal to that in string t and vice versa.

Solutions

- "anagram" has {a:3, n:1, g:1, r:1, m:1 } , "nagaram" has same {a:3, n:1, g:1, r:1, m:1 } thus anagram
- So basically if both string array have same elements despite any order , they are anagram
- thought:
  - so we have to check count of alphabets of both strings and compare
  - sounds like hashmap stuff
  - YES
- Solution 1
  - Sort Both String Arrays
  - check 1 if both lenght are equal or not
  - check 2 traverse and check each element of both array for equality
  - Downside of This STL Sort
    - Differs from lang to lang
    - good sort algos do O(nlogn)
    - bad sort algos do O(n**2)
    - uncertain of SC
    - uncertain of TC
    - lenghty at cases where STL is not available

```
// Solution 1
class Solution {
public:
    bool isAnagram(string s, string t) {
        sort(s.begin(),s.end());
        sort(t.begin(),t.end());
        if (s.size() != t.size()) { return false; }
        for(int i=0; i<s.size(); i++){
            if (s[i]!=t[i]){
                return false;
            }
        }
        return true;
```

```
    }
};
// Time & Space Complexity
// Time complexity:
// O(nlogn+mlogm)
// Space complexity:
// O(1) or
// O(n+m) depending on the sorting algorithm.
// Where n is the length of string s and m is the length of string t.
```

- Solution 2
    - sort both s & t
    - return s==t ?

```
// Solution 2
class Solution {
public:
    bool isAnagram(string s, string t) {
        sort(s.begin(),s.end());
        sort(t.begin(),t.end());
        if (s.size() != t.size()) { return false; }
        return (s==t) ;
    }
};
// Time & Space Complexity
// Time complexity:
// O(nlogn+mlogm)
// Space complexity:
// O(1) or
// O(n+m) depending on the sorting algorithm.
// Where n is the length of string s and m is the length of string t.
```

- thought: Acc to ASCII, lets assume each char as a number bars
    - just add all ascii numbers as score 1 and score 2
    - if they are same then the strings are anagrams
    - NO, as 1+4 = 2+3
- Solution 3
    - The Hash Map Approach COUNTING
    - make char freq hashmap
    - traverse through str1 , inc freq. of freq hashmap
    - traverse through str1 , dec freq. of same freq hashmap
    - if both strs are anagrams, then second string traversal would cancel out all increasing of freq in freq hashmap

```

```

```cpp
// Solution 3
class Solution{
    public:
        bool isAnagram(string s, string t){
            if (s.size() != t.size()) { return false; }
            unordered_map<char, int> freq_map;
            for (int i=0; i<s.size(); i++){
                freq_map[s[i]]++;
            }
            for (int i=0; i<s.size(); i++){
                freq_map[t[i]]--;
            }
            for (int i=0; i<s.size(); i++){
                if(freq_map[s[i]]>0) {
                    return false;
                }
            }
            return true;
        }
};
// Time & Space Complexity
// Time complexity:
// O(n+m)
// Space complexity:
// O(1) or O(k) since we have at most 26 different characters.
// The space is proportional to the number of unique characters, let's call it k
// Where n is the length of string s and m is the length of string t.
```

- Solution 4 -- optimal
    - The Array-as-Counter Approach
        - SAME "The Hash Map Approach COUNTING" as Array
    - make char freq array with index
    - it dont stores character like hashmaps, indexes play save some space
    - traverse through str1 , inc freq. of arr
    - traverse through str1 , dec freq. of same arr
    - if both strs are anagrams, then second string traversal would cancel out all increasing of freq in arr

```cpp
// Solution 4
class Solution{
    public:
        bool isAnagram(string s, string t){
            if (s.size() != t.size()) { return false; }
            vector<int> counts(26,0);
            for (int i=0; i<s.size(); i++){
                counts[s[i]-97]++;
            }
            for (int i=0; i<s.size(); i++){
```

```
                counts[t[i]-97]--;
            }
            // or merge them
            // for (int i = 0; i < s.length(); i++) {
            //     count[s[i] - 'a']++;
            //     count[t[i] - 'a']--;
            // }
            for (int i=0; i<s.size(); i++){
                if(counts[s[i]-97]>0) {
                    return false;
                }
            }
            // or traverse like this
            // for (int val : count) {
            //     if (val != 0) {
            //         return false;
            //     }
            // }
            return true;
        }
};
// Time & Space Complexity
// Time complexity:
// O(n+m)
// Space complexity:
// O(1) since we have at most 26 different characters.
// Where n is the length of string s and m is the length of string t.
```

- Solution 5
  - The Hash Map Approach COUNTING WAY 2
    - make char freq hashmap
    - traverse through str1 , inc freq. of freq hashmap
    - traverse through str1 , inc freq. of another freq hashmap2
    - return freq_map == freq_map2

```
// Solution 5
class Solution{
    public:
        bool isAnagram(string s, string t){
            if (s.size() != t.size()) { return false; }
            unordered_map<char, int> freq_map;
            unordered_map<char, int> freq_map2;
            for (int i=0; i<s.size(); i++){
                freq_map[s[i]]++;
            }
            for (int i=0; i<s.size(); i++){
                freq_map2[t[i]]++;
            }

            return freq_map == freq_map2 ;
        }
```

```
};
// Time & Space Complexity
// Time complexity:
// O(n+m)
// Space complexity:
// O(1) since we have at most 26 different characters.
// Where n is the length of string s and m is the length of string t.
```

---

4 Two Sum [Easy]

- https://leetcode.com/problems/two-sum

> Baidu, Airbnb, Netease, Cisco, Amazon, Aetion, Box, Mathworks, Zoom, Google, Cloudera, Intel, Indeed, Godaddy, Walmart Global Tech, Salesforce, Didi, Affirm, Vmware, Yandex, Microsoft, Adobe, Alibaba, Jpmorgan, Linkedin, Citadel, Emc, Groupon, Intuit, Twitter, Nvidia, Twilio, Valve, Expedia, Yahoo, Zoho, Bookingcom, Wish, Zillow, Morgan-stanley, Drawbridge, Paypal, Huawei, Dropbox, Radius, Zomato, Roblox, Accenture, Goldman-sachs, Lyft, Yelp, Splunk, Bloomberg, Samsung, Bytedance, Servicenow, Quora, Goldman Sachs, Blackrock, Ebay, Ge-digital, Oracle, Qualcomm, Tencent, Uber, Tableau, Spotify, Morgan Stanley, American Express, Sap, Ibm, Deutsche-bank, Snapchat, Dell, Apple, Visa, Works-applications, Facebook, Factset, Audible, Expedia, Affirm, Docusign, Yahoo, Cisco, Servicenow, Goldman Sachs, Amazon, Microsoft, Oracle, Morgan-stanley, Uber, Spotify, Zulily, Google, Paypal, Snapchat, Apple, Goldman-sachs, Yelp, Facebook, Bloomberg

---

Ques

Given an array of integers nums and an integer target, return the indices i and j such that nums[i] + nums[j] == target and i != j.

You may assume that every input has exactly one pair of indices i and j that satisfy the condition.

Return the answer with the smaller index first.

Example 1:

```
Input:
nums = [3,4,5,6], target = 7

Output: [0,1]
Explanation: nums[0] + nums[1] == 7, so we return [0, 1].
```

Example 2:

```
Input: nums = [4,5,6], target = 10

Output: [0,2]
```

Example 3:

```
Input: nums = [5,5], target = 10

Output: [0,1]
```

Constraints:

```
2 <= nums.length <= 1000
-10,000,000 <= nums[i] <= 10,000,000
-10,000,000 <= target <= 10,000,000
```

Recommended Time & Space Complexity You should aim for a solution with O(n) time and O(n) space, where n is the size of the input array.

Hint 1 A brute force solution would be to check every pair of numbers in the array. This would be an O(n^2) solution. Can you think of a better way? Maybe in terms of mathematical equation?

Hint 2 Given, We need to find indices i and j such that i != j and nums[i] + nums[j] == target. Can you rearrange the equation and try to fix any index to iterate on?

Hint 3 we can iterate through nums with index i. Let difference = target - nums[i] and check if difference exists in the hash map as we iterate through the array, else store the current element in the hashmap with its index and continue. We use a hashmap for O(1) lookups.

Solutions

- so basically,
  - given
    - Array
    - target
  - to find
    - index i &index j of array
    - && arr[i]+arr[j]=target
    - && i != j
    - Return the answer with the smaller index first.
    - You may assume that every input has exactly one pair of indices i and j that satisfy the condition.
- Solution 1 -- brute force
  - loop 1 : i 0 to n-1
    - loop 2 : j i+1 to n-1
      - if arr[i]+arr[j]=target
        - return {i,j}
  - if not found ,return {}

```
// Solution 1
class Solution {
public:
    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 {};
    }
};
// Time & Space Complexity
// Time complexity:
// O(n**2)
// Space complexity:
// O(1)
```

- thought: let's just make checks of `(nums[i]+nums[nums.size()-i-1] == target`
  - NO, as ans can be `[0,1]`
- thought: arr[i]+arr[j]=target
  - arr[j]=target-arr[i]
  - we know target, if we just do hardwork to know arr[i] then, arr[j] will automatically come
  - YES, but then we can see arr[j] but need to extract index j
- thought: change the data stucture
  - hashset: NO, `( hashbrown.find(nums[i] )` only returns ptr, not index, tip
  - hashmap: YES, with < value, index >
  - pairs: YES, with < value, index >
  - YES
- Solution 2
  - Hash Map (Two Pass)
  - here we are finding i then j
    - { i,hash_map[target-nums[i]] }
  - make hashmap
    - where hashmap[value]=index
  - now loop traverse, first add all array elements with index nos. in hashmap
  - then another loop, to check
    - if element's Right Side Complement exists
      - i.e. target - arr[i] exists
      - , i.e. arr[j] && target - arr[i] != arr[i] (i.e. not point back to same element)
      - we are checking if the element 1's complement to sum the target exists to the RIGHT SIDE
      - i.e. element + Right Side Complement = target
    - if YES, then return the i ,then j as {i,j}
  - if nothing found, it didn't exist, return empty {}

```cpp
// Solution 2
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        unordered_map<int,int> momos;
        for ( int i = 0; i<nums.size(); i++ ) { momos[nums[i]]=i; }
        for ( int i = 0; i<nums.size(); i++ )
        {
            if ( momos.find(target-nums[i]) != momos.end() // or
momos.count(target-nums[i])
            && momos[target-nums[i]]!= i // to avoid the case of j = i
            )
            {
                return  { i,momos[target-nums[i]]};
            }
            // or
            // int diff = target - nums[i];
            // if (momos.count(diff) && momos[diff] != i) {
            //     return {i, momos[diff]};
            // }
        }
        return {};
    }
};
// Time & Space Complexity
// Time complexity:
// O(n)
// Space complexity:
// O(n)
```

- Solution 3 -- optimal
  - Hash Map (One Pass)
  - Solution 2's Clever/Smart Way
  - One Pass
  - Free from Edge cases
  - here we are finding j then i
    - { hash_map[target-nums[j]] , j }
  - now in one loop,
    - first check
      - if element's Left Side Complement Exists
        - i.e. Left Side Complement + Element = target
        - now we are finding LEFT SIDE, not right side; because this time we are checking before insertion
          - i.e. there is no right side elements of selected element
      - if exists then return i,then j as {i,j}
      - else array element with index no. in hashmap
  - if nothing found, it didn't exist, return empty {}
  - here we already avoide the case of j = i

```
// Solution 3
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        unordered_map<int,int> momos;
        for ( int j = 0; j<nums.size(); j++ ){
            if ( momos.find(target-nums[j]) != momos.end() )
            {
                return  {momos[target-nums[j]] ,j}; // not { j,momos[target-
nums[j]]};
            }
            momos[nums[j]]=j; // not momos[nums[j]]++ as it dont make sense
            // or momos.insert( nums[j], j );
            // insertion should be second
            // else Let's say nums = [3, 2, 4] and target = 6
                // On the first loop (i=0, nums[0]=3), it calculates the
complement: 6 - 3 = 3.
                // Because you just added nums[0] to the map, the code finds 3 in
the map and incorrectly matches the element with itself, returning {momos[3], 0}
which is {0, 0}. This violates the i != j rule.
        }
        return {};
    }
};
// Time & Space Complexity
// Time complexity:
// O(n)
// Space complexity:
// O(n)
```

- thought: let's just sort the array and play first and last in one loop
  - NO, as we lost the indexes of array
  - thought: if we use value_index_pairs_vector instead of vector array
    - our for loop checks pairs at symmetrical indices:
      - When i = 0, it checks the 1st smallest (pv[0]) and the 1st largest (pv[size-1]).
      - When i = 1, it checks the 2nd smallest (pv[1]) and the 2nd largest (pv[size-2]).
      - ...and so on.
    -

```
// // WRONG
// class Solution {
// public:
//     vector<int> twoSum(vector<int>& nums, int target) {
//         vector< pair<int,int> > pv;
//         for (int i=0; i< nums.size(); i++){
//             pv.push_back( {nums[i],i} );
//         }
//         sort(pv.begin(), pv.end());
//         for (int i=0 ; i<pv.size(); i++)
//             if (
//                 pv[i].first
```

```
//                 +
//                 pv[pv.size()-i-1].first
//                 ==
//                 target
//             )
//             {
//                 return (
//                     {
//                         pv[i].second
//                         ,
//                         pv[pv.size()-1-i].second
//                     }
//                 );
//             }
//         return {};
//     }
// };
```

    - NO
        - The problem is that the correct pair isn't always symmetrical. The solution might be the 1st smallest and the 3rd largest
- Solution 4
    - Sorting
    - This is the two-pointer technique. I just need to put this logic into a while (left < right) loop. It's guaranteed to find the solution.
    - thought: if we use value_index_pairs_vector instead of vector array. Let's go back to the sorted list and think again. I have one pointer at the start (left) and one at the end (right).
        - The Core Question: I have sum = left_value + right_value. How does this sum guide my next move?
        - Case 1: sum > target: The total is too big. To make it smaller, the only logical move is to decrease the larger number. So, I must move the right pointer inward (right--).
        - Case 2: sum < target: The total is too small. To make it bigger, the only logical move is to increase the smaller number. So, I must move the left pointer inward (left++).
        - YES
        - textbook-quality solution !!!

```
// Solution 4
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        vector< pair<int,int> > pv;
        for (int i=0; i< nums.size(); i++){
            pv.push_back( {nums[i],i} );
        }
        sort(pv.begin(), pv.end());
        int i =0;
        int j = nums.size() -1;
        while (i<j){
            if ( pv[i].first + pv[j].first == target )
```

```
            {
                return  { min(pv[i].second , pv[j].second) , max(pv[i].second ,
pv[j].second) } ;
                // NOT return ( { min(pv[i].second , pv[j].second) ,
max(pv[i].second , pv[j].second) } );
                // The syntax ( { ... } ) is not the standard way to return a
newly created vector. The compiler misinterprets it and complains that it can't
convert the result to a std::vector<int>
            }
            else if ( pv[i].first + pv[j].first < target ) { i++; }
            else if ( pv[i].first + pv[j].first > target ) { j--; }

        }

        return {};
    }
};
// Time & Space Complexity
// Time complexity:
// O(nlogn)
// Space complexity:
// O(n)
```

---

5 Group Anagrams [Medium]

- https://leetcode.com/problems/group-anagrams/

> Twilio, Salesforce, Affirm, Docusign, Yahoo, Cisco, Servicenow, Blackrock, Goldman Sachs, Ebay, Vmware, Tiktok, Bookingcom, Electronic-arts, Amazon, Wish, Microsoft, Yandex, Oracle, Qualtrics, Bloomberg, Adobe, Alation, Uber, Nutanix, Jpmorgan, Tesla, Mathworks, Zulily, Google, Hulu, Ibm, Snapchat, Apple, Intuit, Visa, Goldman-sachs, Yelp, Facebook, Walmart Global Tech

---

Ques

Given an array of strings strs, group all anagrams together into sublists. You may return the output in any order.

An anagram is a string that contains the exact same characters as another string, but the order of the characters can be different.

Example 1:

```
Input: strs = ["eat","tea","tan","ate","nat","bat"]

Output: [["bat"],["nat","tan"],["ate","eat","tea"]]
```

Explanation:

```
There is no string in strs that can be rearranged to form "bat".
The strings "nat" and "tan" are anagrams as they can be rearranged to form each
other.
The strings "ate", "eat", and "tea" are anagrams as they can be rearranged to form
each other.
```

Example 2:

```
Input: strs = ["x"]

Output: [["x"]]
```

Example 3:

```
Input: strs = ["a"]

Output: [["a"]]
```

Constraints:

```
1 <= strs.length <= 1000
0 <= strs[i].length <= 100
strs[i] consists of lowercase English letters.
```

Recommended Time & Space Complexity You should aim for a solution with O(m * n) time and O(m) space, where m is the number of strings and n is the length of the longest string.

Hint 1 A naive solution would be to sort each string and group them using a hash map. This would be an O(m * nlogn) solution. Though this solution is acceptable, can you think of a better way without sorting the strings?

Hint 2 By the definition of an anagram, we only care about the frequency of each character in a string. How is this helpful in solving the problem?

Hint 3 We can simply use an array of size O(26), since the character set is a through z (26 continuous characters), to count the frequency of each character in a string. Then, we can use this array as the key in the hash map to group the strings.

Solutions

- basically
    - given array of strs
    - return array of sublists of anagram string groups

- thought: You're trying to use a std::vector like a dictionary or a hash map, where you can look up a value using a string key.
    - NO
    - However, a std::vector in C++ doesn't work that way; it's just a dynamic array that you can access with integer indices (like tempo[0], tempo[1], etc.). It doesn't have a .find() method that takes a string, nor can you use a string inside the square brackets [].
- thought: The perfect tool for this is a std::unordered_map. It allows you to store key-value pairs, which is exactly what you need: the sorted string as the key and the list of anagrams as the value.
    - YES
- Solution 1 -- brute force
    - Sorting
    - thought:
        - i grouped them based on id as identity of sorted string
        - so i need to do one sort operation each traversal to find id which is sorted str
        - YES
    - make unord hashmap tempo
        - contain stuff as pairs {"abc", {"acb", "bac"}}
            - "abc"
            - {"acb", "bac"}
    - traverse through array of strs
        - each element
        - sort it using sort stl
        - if sorted string exists in tempo
            - append the element in sorted pair
        - else
            - make pair of
                - sorted element key
                - empty array
    - make new vector of vector of string fin
    - traverse through tempo
        - append second of pair into fin
    - return fin
    - TC
        - for each str : n log n from sort
        - for arra of m elements : m* n log n

```cpp
// Solution 1
class Solution {
public:
    vector<vector<string>> groupAnagrams(vector<string>& strs) {
        unordered_map< string, vector<string> > tempo;

        for ( int i=0 ; i<strs.size() ; i++){
            // one string iteration "string1" strs[i]
            string s = strs[i];
            sort(s.begin(),s.end()); // s is sorted strs[i]
```

```
            // Add the original string to the vector associated with the sorted
key.
            // If the key doesn't exist, C++ creates it automatically!
            tempo[s].push_back(strs[i]);
        //     if ( tempo.find(s) != tempo.end ) // if s exist in tempo
        //     {
        //         // append strs[i] in tempo[s] tempo[s].push_back(strs[i])
        //         tempo[s].push_back(strs[i]);
        //     }
        //     else
        //     {
        //         // make new str,{} in tempo
        //         tempo[s].push_back(strs[i]);
        //     }
        }
        vector<vector<string>> fin;
        for ( auto const& pairs: tempo){ // eg: pair { "abc", {array of anagrams}
}
            fin.push_back(pairs.second);
        }
        return fin;
    }
};
// Time & Space Complexity
// Time complexity:
// O(m*nlogn)
// Space complexity:
// O(m*n)
// Where
// m is the number of strings and
// n is the length of the longest string.
```

- Solution 2 -- optimal
  - Hash Map/Table
  - same working of Solution 1
  - but another ID
  - thought:
    - i grouped them based on id as identity
    - so i did earlier to do one sort operation each traversal to find id which is sorted str
    - is there any other way to make ids and group play??
    - idea: make an id
    - made of 0s and 1s of freq array of 26 alphabet & one seperator
      - eg: abc => 1.1.1.0.0.0.0.....0
        - bca => 1.1.1.0.0.....0
          - same as abc, cab, bac etc.
    - YES, really like adj matrix
  - TC:
    - just using hashmap
    - just counting stuff
    - O(m * n * 26)

- n is elements size
- m is array size
- 26 is Alphabets iteration

```cpp
// Solution 2
class Solution {
public:
    vector<vector<string>> groupAnagrams(vector<string>& strs) {
        unordered_map< string, vector<string> > tempo;
        for ( const string& s : strs)
        {
            // Create a character count array for each string
            // An array of 26 integers, one for each letter 'a' through 'z'.
            vector<int> freq(26,0);
            for (char c: s)
            {
                freq[c-'a']++;
            }
            // Build a unique key string from the count array.
            // For "eat": count['a'-'a']=1, count['e'-'a']=1, count['t'-'a']=1
            // Key might look like: "1.0.0.0.1.....1...."
            string _ID = "";
            for ( int i =0; i<26; i++)
            {
                _ID+= to_string(freq[i]);
                _ID+="."; // as a seperator
            }
            tempo[_ID].push_back(s);
        }
        vector< vector<string>> fin={};
        for (auto const& [key,val]: tempo)
        {
            fin.push_back(val);
        }
        return fin;
    }
};
// Time & Space Complexity
// Time complexity:
// O(m*n)
// Space complexity:
// O(m) extra space.
// O(m*n) space for the output list.
// Where
// m is the number of strings and
// n is the length of the longest string.
```

---

6 Top K Frequent Elements [Medium]

- https://leetcode.com/problems/top-k-frequent-elements/description/

> Bytedance, Yahoo, Cisco, Vmware, Ebay, Pocket-gems, Amazon, Microsoft, Oracle, Adobe, Uber, Spotify, Google, Linkedin, Hulu, Snapchat, Apple, Goldman-sachs, Yelp, Facebook, Bloomberg

---

Ques

Given an integer array nums and an integer k, return the k most frequent elements within the array.

The test cases are generated such that the answer is always unique.

You may return the output in any order.

Example 1:

```
Input: nums = [1,2,2,3,3,3], k = 2

Output: [2,3]
```

Example 2:

```
Input: nums = [7,7], k = 1

Output: [7]
```

Constraints:

```
1 <= nums.length <= 10^4.
-1000 <= nums[i] <= 1000
1 <= k <= number of distinct elements in nums.
```

Recommended Time & Space Complexity You should aim for a solution with O(n) time and O(n) space, where n is the size of the input array.

Hint 1 A naive solution would be to count the frequency of each number and then sort the array based on each element's frequency. After that, we would select the top k frequent elements. This would be an O(nlogn) solution. Though this solution is acceptable, can you think of a better way?

Hint 2 Can you think of an algorithm which involves grouping numbers based on their frequency?

Hint 3 Use the bucket sort algorithm to create n buckets, grouping numbers based on their frequencies from 1 to n. Then, pick the top k numbers from the buckets, starting from n down to 1.

Solutions

- this question seems to be easy
- but this is quite hard to solve once

- basically
    - Given an integer array nums and an integer k
    - return the k most frequent elements within the array.
    - The test cases are generated such that the answer is always unique.
- thought :
    - was blackout, full algo didn't came at first try
    - origin thinking was to make a map containing freq, element
    - sort them desc
    - make fin arr
    - and while k--
        - append fin freq's element
    - return
    - NO
        - was beginner, other way too as iteration
    - Solution 1 came out of incapability of using just unordered_map or vector or vector pair
        - i tried here to use their synergy
        - YES
    - Will try this thought inspiration in Sol 2
        - YES
- Solution 1 -- optimal
    - Bucket Sort
    - Need to Revisit after Getting Grip
    - thought: the "bucket" array, where the index represents the frequency (count) Advantage

```
eg:
[1,1,2,2,2,3,10]
- this is unbounded
  - i.e. maxelement of this array exceed size of array
  - i.e. any element 100000,99999,etc. can be in this array
  - COUNT IS BTW TIMES OF OCCURENCE
- way 1 to store
  - i :       [0,1,2,3,4,5,6,7,8,9,10] IDX
  - count :   [0,2,3,1,0,0,0,0,0,0,1] VALUES
- way 2 to store -- optimal
  - count : [0,1,2,3,4,5,6] IDX
  - i : [[null],[10,3],[1],[2],X,X,X] VALUES
  - very much TC & SC Efficient
  - + new way to process
    - also even if array is big, still stop way 2 bucket till k ,
ignore after that
      - starting from reverse obv.
      - i.e. if k=2, so
        - 6,5,4 is out ,no data
        - process 3,2
```

```
          - ignore rest !!!
      - linear time
```

- The two-step process is the standard, efficient pattern.
- Use a map for what it's best at: fast lookups and counting.
- Use a vector for what it's best at: storing data in a sequence that can be sorted.
- make map named tempo
  - {{number,freq},{number,freq},{number,freq}}
  - Count the frequency of each number.
- make bucket vector tempo2
  - {{number,number,number},{number,number,number},{number,number,number}}
- here 0 index vector have 0 freq nos, 1 index nos have 1 freq nos, etc
- 0 would be {} obv
  - useage
- eg: buckets[5] = a list of all numbers that appeared exactly 5 times.
- eg: buckets[4] = a list of all numbers that appeared exactly 4 times.
- now traverse tempo
  - store tempo2[freq].push_back(number)
- done
- now to just append to fin array and return fin
- actual_size_bucket = tempo2.size() -1
- travers tempo2 reverse order
  - You can't sort the buckets vector itself, because the frequency information is stored in its indices, not in the values it holds. Sorting it would scramble this crucial relationship.
  - so now
  - traverse i lastFreq -> 0 && fin<=k
- for nums in tempo[i]
  - fin.push_back(nums)
- check if fin==k
  - The test cases are generated such that the answer is always unique
  - break
- else
  - continue
- return fin

```cpp
// Solution 1
class Solution {
public:
    vector<int> topKFrequent(vector<int>& nums, int k) {
        unordered_map<int,int> tempo; // COUNT
        // 1 Count the frequency of each number.
        for ( const int i : nums )
        {
            tempo[i]++;
        }

        // The two-step process is the standard, efficient pattern.
```

```
        // Use a map for what it's best at: fast lookups and counting.
        // Use a vector for what it's best at: storing data in a sequence that can
be sorted.

        // 2 Create buckets. The index is the frequency.
        // The size is nums.size() + 1 because a number can appear at most
nums.size() times.
        vector<vector<int>> tempo2(nums.size() + 1); // FREQ
        // Connecting Frequency to Array Indexing To store the number 5 in our
buckets vector, we need to place it at the index corresponding to its frequency.
So, we need to access buckets[4].
        // Because C++ vectors are 0-indexed, to have a valid index at 4, the
vector must have a size of at least 5 (to contain indices 0, 1, 2, 3, and 4).
        // Therefore, if the maximum possible frequency is nums.size(), the
required size for the buckets vector is nums.size() + 1.
        for ( const auto& [key,value] : tempo) // key,value are num, freq res.
        {
            tempo2[value].push_back(key);
        }

        // NO sort(tempo2.rbegin(),tempo2.rend());
        // You can't sort the buckets vector itself, because the frequency
information is stored in its indices, not in the values it holds. Sorting it would
scramble this crucial relationship.
        // You can't sort the buckets vector itself, because the frequency
information is stored in its indices, not in the values it holds. Sorting it would
scramble this crucial relationship.
        // The Role of the buckets Vector
        //     Think of the buckets vector like a series of filing cabinet
drawers. The number on each drawer is the frequency (the index), and inside the
drawer are the numbers that appeared that many times.
        //     eg: buckets[5] = a list of all numbers that appeared exactly 5
times.
        //     eg: buckets[4] = a list of all numbers that appeared exactly 4
times.
        vector<int> fin;
        // 3 Iterate backwards from the highest possible frequency.
        // Add elements to the result until we have k elements.
        int actual_size_bucket = tempo2.size() -1;
        for ( int i = actual_size_bucket ; i >=0 && fin.size()<k ; --i )
        {
            for ( int num: tempo2[i]){
                fin.push_back(num);
                if ( fin.size() == k){
                    break;
                }
            }
        }
        return fin;


    }
};
// Time & Space Complexity
```

```
// Time complexity:
// O(n)
// Space complexity:
// O(n)
```

- Solution 2 -- brute force
- Sorting
  - Count frequencies
    - make unord map <int,int> freqMap
    - for nums in nums
      - freqMap num ++
        - it will inc freq of existing fre in freq map
        - or if new so make new freq
  - Convert to a vector of pairs { freq, number}
    - make vect of pairs freqVec
    - traverse map_pair in freqMap
      - freqVec.push_back({ map_pair.second, map_pair.first })
  - Sort the vector by frequency in descending order
    - use sort(freqVec.rbegin(), freqVec.rend())
  - Extract the top k elements
    - make fin which is vect int array
    - i 0->k-1
      - fin.push_back freqVec [i] Second
  - return fin

```
// Solution 2
class Solution {
public:
    vector<int> topKFrequent(vector<int>& nums, int k) {
        unordered_map<int,int> freqMap;
        for ( const int i : nums )
        {
            freqMap[i]++;
        }
        vector<pair<int,int>> freqVect;
        for ( const auto& [k,v]: freqMap){
            freqVect.push_back({v,k});
        }
        sort(freqVect.rbegin(),freqVect.rend());
        vector<int> fin;
        int idx=0;
        while(k--)
        {
            fin.push_back(freqVect[idx].second);
            idx++;
        }
        return fin;
    }
```

```
};
// Time & Space Complexity
// Time complexity:
// O(nlogn)
// Space complexity:
// O(n)
```

- Solution 3
  - Min-Heap
  - Need to Revisit After Getting grip on Heap
  - We don't really need to sort whole arr
  - We just need to find k freq. so extract k times too..
  - heap push O(n)
  - pop only k times O(klogn)
    - each pop O(logn)

```
// Solution 3
// Defines the 'Solution' class, which holds our function
class Solution {public:
// Declares the function 'topKFrequent' which takes a vector of integers 'nums'
// and an integer 'k', and returns a vector of integers.
     vector<int> topKFrequent(vector<int>& nums, int k) {
// 1. --- Frequency Counting ---

// Create an unordered_map (hash map) to store the frequency of each number.
// The 'key' will be the number from 'nums', and the 'value' will be its count.
    unordered_map<int,int> count;
// Loop through each number ('num') in the input vector 'nums'.
// Increment the count for the current 'num' in the map.
// If 'num' is not in the map yet, it's automatically added with a count of 1.
    for (int num: nums) {count[num]++;}

// 2. --- Min-Heap for Top K ---

// Create a min-heap (priority_queue). This is the clever part!
// It stores pairs: {frequency, number}.
// 'greater<...>' makes it a min-heap, meaning the pair with the *smallest* //
frequency will always be at the top.
    priority_queue<pair<int,int> , vector<pair<int,int>> ,greater<pair<int,int>>>
hp; // heap
// Loop through each 'entry' (a {number, frequency} pair) in our 'count' map.
// Note: 'entry.first' is the number, 'entry.second' is its frequency.
    for ( auto& i : count){

// Push the {frequency, number} pair onto the min-heap.
// We put frequency *first* so the heap sorts by frequency.
        hp.push( {i.second,i.first} ) ;


// This is the key optimization:
```

```
// If the heap's size exceeds 'k', we remove the smallest element.
// Since it's a min-heap, 'heap.pop()' removes the element with the
// *lowest frequency* currently in the heap.
        if ( hp.size() > k) {hp.pop();}
    }


// After the loop, the heap contains exactly the 'k' elements
// with the highest frequencies.

// 3. --- Final Result ---

// Create a vector 'res' to store our final result.
    vector<int> fin;

// Loop 'k' times to extract all elements from the heap.
// (Alternatively, you could use 'while (!heap.empty())')
    for( int i = 0; i<k ; i++){

// Get the top element from the heap (which is a {frequency, number} pair).
// 'heap.top().second' accesses the 'number' part of the pair.
        fin.push_back(hp.top().second);
        // NO fin.push_back(hp.top().second());
        // NO fin.push_back(hp.top.second());
        // NO fin.push_back(hp.top.second);

// Remove the top element from the heap to access the next one.
        hp.pop();
    }

// Return the 'res' vector containing the top k frequent numbers.
// Note: The order in 'res' isn't guaranteed (e.g., it might be
// from k-th most frequent to 1st most frequent), but the problem
// usually allows any order.
    return fin;
    }
};
// Time & Space Complexity
// Time complexity:
// O(nlogk)
// Space complexity:
// O(n+k)
// Where
// n is the length of the array and
// k is the number of top frequent elements.
```

Template

Topic

TotalCompanyTags

> Total Company Tags

PreReqs QuesNotes

- a

```

```

```
#include<bits/stdc++.h>
int main(){

    return 0;
}
```

Sr.No. Question [Easy/Medium/Hard]

- Link

> Company Tags, Ctrl+Shift+Alt+ArrowKeys

Ques

Content

Solutions

```

```

- Prompt for PreReqs of My Multiple Resources(courses, notes, papers)

```
You are an expert C++ tutor. Your goal is to teach me the necessary concepts to
solve a specific coding problem, but without revealing the final answer's logic. I
will provide you with both the problem description and the complete, working C++
solution. Based on this information, you must first identify all the core
programming concepts, data structures, and C++ features used in the solution.
Then, for each of these topics, you must explain it to me from a beginner's
perspective. Each explanation should cover what the concept is, why it's useful in
general, its basic C++ syntax, and a small, self-contained code snippet that
demonstrates only that single concept in isolation. It is crucial that you do not
explain the line-by-line logic of the solution I provided or combine your examples
```

```
into the final answer. Your entire purpose is to give me the individual building
blocks so that I can construct the final solution myself.
```

- Prompts for Notes of My Multiple Resources(courses, notes, papers)

```
Create super depth notes in Markdown (.md) format with 100% information preserved,
no loss. Use simple grammar and keep everything clear, direct, and well-
structured. using headings, subheadings,paragraphs, statements and code blocks
when needed. Include every detail, definition, example, and step exactly from the
source. transform the given content into clean, readable .md format.
and no #, just nested - lines plaintext
```

End-of-File

The god-stack repository, authored by Kintsugi-Programmer, is less a comprehensive resource and more an Artifact of Continuous Research and Deep Inquiry into Computer Science and Software Engineering. It serves as a transparent ledger of the author's relentless pursuit of mastery, from the foundational algorithms to modern full-stack implementation.

> Made with 🤍 Kintsugi-Programmer