CS204(2025)

Hashing

Instructions for Implementation

• Write the program in C or C++ using standard input/output.
• Follow the input/output format strictly.
• Ensure your code:
  ◦ Handles edge cases correctly.
  ◦ Meets the time and space complexity constraints.
• Do not use STL containers like vector, set, or map in C++. Use raw arrays unless explicitly allowed.
• You are encouraged to write clean, modular, and well-documented code.

Q1. Hash Table Implementation using Linear Probing

Problem Statement:

Implement a hash table using the **Linear Probing** method for collision resolution.
The hash table should store **integer keys**.
When a collision occurs, the next empty position (circularly) should be used to insert the key.

You need to perform the following operations based on user input:

- **I x** → Insert key x into the hash table.
- **D x** → Delete key x from the hash table (if present).
- **S x** → Search for key x and print whether it is found or not.

Use the hash function:

h(key) = key % TABLE_SIZE

Use `-1` to represent an **empty slot** and `-2` to represent a **deleted slot** in the table.

Input Format:

- First line: Integer `TABLE_SIZE` (size of the hash table).
- Then a series of operations as described above until `S x` is encountered.

Output Format:

- For **Search (S)**: Print `"Found <number_of_probes>"` or `"NotFound <number_of_probes>", where <number_of_probes> is the number of comparisons made during the search.`

Example:

Input:

5

I 10

I 15

I 20

D 15

S 20

Output:

Found 2

Q2. Hash Table Implementation using Chaining

Problem Statement:

Implement a hash table using the **Chaining** method for collision resolution. The hash table should store integer keys and resolve collisions using linked list. You need to perform the following operations based on user input:

- **I x** → Insert key x into the hash table.
- **D x** → Delete key x from the hash table (if present).
- **S x** → Search for key x and print whether it is found or not.

Use a hash function of the form:
`h(key) = key % TABLE_SIZE`

Input Format:

- First line: Integer `TABLE_SIZE` (size of the hash table).
- Then a series of operations as described above until `S x` is encountered.

Output Format:

- For **Search (S)**: Print `"Found"` or `"NotFound"`

Example:

Input:

5

I 10

I 15

I 20

S 15

Output:

Found

Q3. Longest Substring Without Repeating Characters

Problem Statement:

Given a string s, find the **length of the longest substring** that contains **no repeating characters**.

Use **hashing** (e.g., an array or hash table) to efficiently keep track of which characters are currently in the substring.
You should use this information to slide a window over the string and determine the maximum length possible.

Note:
Use a **hash array** (e.g., integer array of size 256) to represent the character hash table.

Input Format:

- One line containing the input string s.

Output Format:

- One line containing the length of the longest substring without repeating characters.

Constraints:

- `0 <= length(s) <= 5 * 10^4`
- s consists of English letters, digits, symbols, and spaces.

Test Case 1:

Input:

abcabcbb

Output:

3

Explanation: The answer is "abc", with the length of 3. Note that "bca" and "cab" are also correct answers.

Test Case 2:

Input:

bbbbb

Output:

1

Explanation: The answer is "b", with the length of 1.

Constraints:

- 0 <= length <= 5 * 10^4
- s consists of English letters, digits, symbols and spaces.

Q4. Roman to Integer

Problem Statement:

Roman numerals are represented by seven different symbols, each with a fixed integer value. To convert a Roman numeral string into its integer value, we can use **hashing** to map each Roman symbol to its corresponding integer.

If a smaller numeral appears **before a larger one**, it is **subtracted** instead of added (e.g., IV = 4, IX = 9).

Symbol	Value
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

For example, 2 is written as II in Roman numeral, just two ones added together. 12 is written as XII, which is simply X + II. The number 27 is written as XXVII, which is XX + V + II.

Roman numerals are usually written largest to smallest from left to right. However, the numeral for four is not IIII. Instead, the number four is written as IV. Because the one is before the five, we subtract it making four. The same principle applies to the number nine, which is written as IX. There are six instances where subtraction is used:

- I can be placed before V (5) and X (10) to make 4 and 9.
- X can be placed before L (50) and C (100) to make 40 and 90.
- C can be placed before D (500) and M (1000) to make 400 and 900.

Given a roman numeral, convert it to an integer.

You are required to **implement this conversion using hashing** — i.e., use a hash table or array lookup to store Roman symbols and their corresponding values.

Input Format

- One line containing a string representing the Roman numeral.

Output Format

- One line containing the integer value of the given Roman numeral.

Test Case 1:

Input:

III

Output:

3

Explanation: III = 3.

Test Case 2:

Input:

LVIII

Output:

58

Explanation: L = 50, V= 5, III = 3.

Constraints:

- 1 <= s.length <= 15
- s contains only the characters ('I', 'V', 'X', 'L', 'C', 'D', 'M').
- It is **guaranteed** that s is a valid roman numeral in the range [1, 3999].

Q5. Most Common Word

Problem Statement:

You are given a paragraph and a list of banned words.
Your task is to find the **most frequent word** in the paragraph that is **not banned**.

Use **hashing** to efficiently count word frequencies.
The words in the paragraph are **case-insensitive** (i.e., "Ball" and "ball" are the same word).
Ignore **punctuation symbols** (such as `. , ; : ? !` etc.), and treat them as word separators.

You must use **hashing (character and word-level)** to store and count word occurrences.
Do **not** use STL containers like `map` or `unordered_map`; instead, implement your own hash-based structure or fixed-size array-based hashing.

Input Format:

- A long string paragraph **terminated by** # (words are separated by spaces or punctuation).
- A list of banned words **separated by spaces and terminated by** #.

Output Format:

- The most frequent non-banned word (in lowercase).

Constraints:

- The input paragraph length ≤ 10^4 characters.
- Each word consists of lowercase or uppercase English letters only.
- There is at least one word that is not banned.
- The answer is unique.

Approach (Using Hashing):

1. Convert the paragraph to lowercase.
2. Replace all punctuation marks with spaces.
3. Tokenize words by spaces.
4. Maintain a **hash table** to count occurrences of each word.
5. Maintain another hash table for **banned words**.
6. Find the most frequent word that is **not in the banned list**.

Test Case 1:

Input:

Bob hit a ball, the hit BALL flew far after it was hit. #

hit #

Output: ball

Explanation:

"hit" occurs 3 times, but it is a banned word.

"ball" occurs twice (and no other word does), so it is the most frequent non-banned word in the paragraph.

Note that words in the paragraph are not case sensitive, that punctuation is ignored (even if adjacent to words, such as "ball,"), and that "hit" isn't the answer even though it occurs more because it is banned.

Test Case 2:

Input:

paragraph = a.#

banned = #

Output: a

Constraints:

- 1 <= paragraph.length <= 1000

- paragraph consists of English letters, space ' ', or one of the symbols: "!?',;.".
- 0 <= banned.length <= 100
- 1 <= banned[i].length <= 10
- banned[i] consists of only lowercase English letters.

Q6. Group Anagrams using Hashing

Problem Statement:
Given a list of words, group all the *anagrams* together.
Two words are considered anagrams if they contain the same characters in a different order (e.g., `"eat"`, `"tea"`, and `"ate"`).

You must solve this problem using **hashing**, where each word is inserted into a hash table based on a key derived from its characters.
Use any sorting algorithm or logic to print the final output groups in dictionary order.

Approach Hint:

- Use a hash map where the **key** is a canonical form of the word (e.g., sorted characters or frequency count).
- The **value** is a list of words that match this key (i.e., are anagrams).
- Finally, print all the anagram groups in sorted order.

Input Format:

- A single line containing space-separated words.
- The input is terminated by #.

Output Format:

- A series of word groups, each enclosed in square brackets `[ ]`, representing a group of anagrams.
- Groups and words within each group should be printed in dictionary order.

Constraints:

- 1 ≤ number of words ≤ 10^4
- Each word contains lowercase English letters only.
- You must use a hashing-based approach for grouping.

Test Case 1:
Input:

```
 eat tea tan ate nat bat #
```

Output:

```
[ate eat tea] [bat] [nat tan]
```

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.

Input:

```
a #
```

Output:

```
 [a]
```

Q7. Destination City using Hashing

Problem Statement:
You are given an array of city pairs, where each pair represents a **directed path** from `cityA` to `cityB`. Your task is to find the **destination city**, i.e., the city that **has no outgoing paths** to any other city.

It is guaranteed that the paths form a **linear chain** without loops, so there will always be exactly **one destination city**.

You must solve this problem using **hashing** — by storing the cities that have outgoing paths and checking which city is not present in that set or hash map.

Input Format:

- The first line contains an integer n — the total number of city pairs.
- The next n lines each contain two space-separated strings — `cityA` and `cityB`, representing a path from `cityA` → `cityB`.

Output Format:

- Print the name of the **destination city** (the one without any outgoing path).

Approach Hint:

- Use a **hash set** to store all cities that have outgoing paths (`cityA`).
- Then, iterate through all destination cities (`cityB`) and find the one not present in the hash set — that is the final destination.

Test Case 1:
Input:

3

London NewYork

NewYork Lima

Lima SaoPaulo

Output:

SaoPaulo

Explanation: Starting at "London" city you will reach "SaoPaulo" city which is the destination city. Your trip consist of: "London" -> "NewYork" -> "Lima" -> "SaoPaulo".

Test Case 2:

Input:

3

B C

D B

C A

Output:

A

Explanation: All possible trips are:

"D" -> "B" -> "C" -> "A".

"B" -> "C" -> "A".

"C" -> "A".

"A".

Clearly the destination city is "A".

Constraints:

- 1 <= paths.length <= 100
- paths[i].length == 2
- 1 <= $cityA_i$.length, $cityB_i$.length <= 10
- $cityA_i$!= $cityB_i$
- All strings consist of lowercase and uppercase English letters and the space character

Q8. Detect Loop in a Singly Linked List using Hashing

Problem Statement:

Given a **singly linked list**, check whether it contains a **loop (cycle)** or not.

Use **hashing** to store the addresses (or values) of visited nodes to detect if a node is visited more than once.

Input Format:

1. First line: Space-separated integers representing node values, terminated by `-1`.
2. Second line: Integer n — number of links.
3. Next n lines: Two integers a b meaning node with value a points to node with value b.

Output Format:

- Print `True` if there is a loop, `False` otherwise.

Constraints:

- 1 ≤ Number of nodes ≤ 10^5
- -10^9 ≤ Node Value ≤ 10^9

Example 1:

Input:

1 2 3 4 5 -1

5

1 2

2 3

3 4

4 5

5 2

Output:

True

Explanation:

There exists a loop: `5 → 2 → 3 → 4 → 5`