HASHMAP-3

Assignment Solutions

PW SKILLS

Q1. Given a map m, which is passed a parameter to the following functions. Write line(s) of code to complete the following functions:

a) Add the pair to the map m, where k represents key and v represents value.

```
void add_to_map(map<int,int> &m, int k, int v)
{
    // Write your code here
}
```

b) Return the value of the key k if it is present in the map m otherwise, return -1.

```
int find_in_map(map<int,int> &m, int k)
{
    // Write your code here
}
```

c) Print the map m i.e. key and value.

```
void print_map(map<int,int> &m)
{
    // Write your code here
}
```

A1. The code to complete the following functions:

a) Add the pair to the map m, where k represents key and v represents value.

```
void add_to_map(map<int,int> &m, int k, int v)
{
        m.insert({k, v});
}
```

b) Return the value of the key k if it is present in the map m otherwise, return -1.

```
int find_in_map(map<int,int> &m, int k)
{
        if(m.find(k) != m.end())
            return m[k];
        return -1;
}
```

c) Print the map m i.e. key and value.

```
void print_map(map<int,int> &m)
{
        for(auto i : m)
            cout<<i.first<<" - "<<i.second;
}
```

Q2. Least frequent character

Given a string s. Print the character that is appearing least number of times. If more than one character is appearing least number of times, print the one appearing first in the string.

Example 1:

Input: s = "AAABBcccDDa"
Output: a
Explanation: The count of each character:
A - 3, B - 2, c - 3, D - 2, a - 1
Clearly a is least frequent character. Note that A is different from a.

Example 2:

Input: s = "xxxAyyyBzzzzC"
Output: A
Explanation: The count of each character:
x - 3, A - 1, y - 3, B - 1, z - 4, C - 1
Clearly A, B, C are the least frequent characters. But A is appearing first in the string hence we will print that.

A2. Here, since we have to find the least frequent element, so somewhere we need to store count of each character. That's why we are going to use map here.

- We create a map, where corresponding to each character we store its count.
- Store the answer character in the variable ans.
- Variable min stores INT_MAX initially and this is done to make comparison find the minimum count of characters in the string.
- Then we iterate through the map m make comparisons and find the character with minimum count.
- Since we are using less than < operator, that's why first appearing character will be considered in case of characters with same frequency. However, if we were using less than and equals <= operator then, last appearing character would have been printed.

Solution Code: https://pastebin.com/DefLBd2q
Output:

```
Enter the string: AAABBcccDDa
The character appearing minimum times: a
```

```
Enter the string: xxxAyyyBzzzzC
The character appearing minimum times: A
```

Time complexity: O(N) since we are traversing the entire string at least once.
Space complexity: O(N) because we are creating a map and storing characters and their frequency into it.

Q3. What will be the output of the following code:

```
map<int, int>m = {{1,1}, {2,4}, {3,9}, {4,16}, {5,25}};
   m.insert_or_assign(6, 36);
   m.insert({6, 36});

   m.erase(++m.find(2), m.find(5));

   for(auto i : m)
       cout<<i.first<<" - "<<i.second<<endl;
```

Choose the correct option:
a) Compilation error
b) 1 - 1
2 - 4
5 - 25
6 - 36
c) Syntax error

d) 1 - 1
2 - 4
6 - 36

A3. Answer: b
Explanation: The map m is initially: {{1,1}, {2,4}, {3,9}, {4,16}, {5,25}}
After `m.insert_or_assign(6, 36)`the map m is: {{1,1}, {2,4}, {3,9}, {4,16}, {5,25}, {6,36}}
After `m.insert({6, 36})` no change will happen in map m because {6,36} is already present in the map, also it won't throw any error.
Now `++m.find(2)` will point at {3,9} and `m.find(5)` will point at {5,25}. When erasing, the pairs from [3,5) will be erased. So, {3,9} and {4,16} will be erased, thus leaving the map m as {{1,1}, {2,4}, {5,25}, {6,36}}.

Q4. What will be the output of the following code:

```
map<int, int>m = {{1,1}, {2,4}, {3,9}, {4,16}, {5,25}};

map<int, int>::reverse_iterator it;
for(it = --m.rend(); it =≠ m.rbegin(); it--)
    cout<<it→first<<" "<<it→second<<endl;
```

Choose the correct option:

a) 5 25
4 16
3 9
2 4
1 1

b) 4 16
3 9
2 4
1 1

c) 5 25
4 16
3 9
2 4

d) 1 1
2 4
3 9
4 16

A4. Answer: d
Explanation: Here, reverse_iterator is used traverse the map. So, they will be pointing in the following way:

rend() **rbegin()**

{1, 1}	{2, 4}	{3, 9}	{4, 16}	{5, 25}

Now in the for loop, reverse_iterator it is pointing at --rend(), so it is:

it (--rend())				rbegin()
{1, 1}	{2, 4}	{3, 9}	{4, 16}	{5, 25}

Now as we do, it-- in the for loop, we are going to move towards rbegin() because we are traversing in reverse direction using reverse_iterator.
As soon as it reaches rbegin(), it will hit the for loop condition `it = m.rbegin()` and the control will come out of the for loop so {5,25} pair won't be printed.
Hence, output will be: {{1,1}, {2,4}, {3,9}, {4,16}}.

Q5. Target Sum of 2 elements
Given an integer n representing the number of elements. Then n elements are given. An integer t is given which represents the target. You are required to print the indices of two numbers such that they add up to target t.

You cannot use a single element twice, i.e. the two indices can't be the same.

You can return the answer in any order.

Example 1:

Input:
n = 7
Elements = [1, 4, 5, 11, 13, 10, 2]
Target = 13
Output: [3, 6]
Explanation: Because a[3] + a[6] = 11 + 2 = 13

Example 2:

Input:
n = 5
Elements = [9, 10, 2, 3, 5]
Target = 15
Output: [1, 4]
Explanation: Because a[1] + a[4] = 10 + 5 = 15

A5. Here, once we have n elements and the target value, we pass it to a self-created function called targetSum(). It performs the following steps for a given input array:

- Creates a map.
- Iterates through each element in the array, starting with the first.
- Checks the map for the presence of the required number (required number = target sum – current number) in each iteration.
- If present, returns the number's index & current number's index in the form of vector of size 2.
- Otherwise, adds the current iteration number as a key to the map and its index as a value.
- Repeats until the answer is found.

Solution Code: https://pastebin.com/gTw2MVvu
Output:

```
Enter the number of elements: 7
Enter the elements of the array: 1 4 5 11 13 10 2
Enter the target: 13
The indices of 2 elements are: 3 6
```

```
Enter the number of elements: 5
Enter the elements of the array: 9 10 2 3 5
Enter the target: 15
The indices of 2 elements are: 1 4
```

Time complexity: O(N) since we are traversing all the elements at least once.
Space complexity: O(N) because we are creating a map and storing elements into it.

Q6. Find the majority element in the array. A majority element in an array A[] of size n is an element that appears more than n/2 times. If there is no such element, return -1.

Input1:
n = 9
A[] = {3, 3, 4, 2, 4, 4, 2, 4, 4}
Output1:
4
Input2:
n = 8
A[] = {3, 3, 4, 2, 4, 4, 2, 4}
Output2:
-1

A6. Solution Code: https://pastebin.com/hWUrryeC
Code explanation:

- There can be only one majority element in the entire array as it occurs more than n/2 times.
- We use an unordered_map here that will store the element and its count.
- Traverse the array, enter all the elements in the unordered_map, if found again, increase its value.
- We are storing key-frequency pairs.
- Traverse over the values of the unordered_map, if you find any key whose value is greater than n/2, that is our majority element.
- If you cannot find any, return -1.

Output:

```
Enter the number of elements: 9
Enter the elements: 3 3 4 2 4 4 2 4 4
The Majority Element is: 4
```

```
Enter the number of elements: 8
Enter the elements: 3 3 4 2 4 4 2 4
The Majority Element is: -1
```

```
Enter the number of elements: 5
Enter the elements: 1 2 3 4 5
The Majority Element is: -1
```

Time complexity: O(n) Linear time complexity
Here we are traversing the vector once in order to store the element and its corresponding count in the unordered_map.
Space complexity: O(n) Linear space complexity
Since we are creating an unordered_map to store unique elements and their count.

Q7. Given two arrays, of equal length n, the task is to find if the given arrays are equal or not. Return true if they are equal. [Easy]
Two arrays are said to be equal if:

- **both of them contain the same set of elements,**
- **arrangements (or permutations) of elements might/might not be the same.**
- **If there are repetitions, then counts of repeated elements must also be the same for two arrays to be equal.**

Input1:
n = 5
arr1[] = {1, 2, 5, 4, 0}
arr2[] = {2, 4, 5, 0, 1}
Output1:
Yes

Input2:
n = 3
arr1[] = {1, 7, 1}
arr2[] = {7, 7, 1}
Output2:
No

A7. Solution Code: https://pastebin.com/rdisx37i
Code explanation:

- Use an unordered_map to store elements of the first array.
- Each unique element of the first array will be the key and its value will be its frequency.
- Now, traverse over the second array, and for every element, if it is present in the unordered_map, reduce its frequency by 1.
- If now the frequency becomes 0, remove the element from the unordered_map as it means that the element is no longer present in the first array.
- If an element is not present in the unordered_map, return false there.
- In the end, print true.

Output:

```
n = 5
arr1 = 1 2 5 4 0
arr2 = 2 4 5 0 1
Two arrays are equal? Yes
```

```
n = 5
arr1 = 1 1 2 2 3
arr2 = 3 3 2 2 1
Two arrays are equal? No
```

Time complexity: O(n+n+u) ~ O(2n+u) ~ O(n) Linear time complexity
Here we are traversing the two vectors and once we are traversing the unordered_map that contains unique elements only. Here, u stands for the size of unordered_map.
Space complexity: O(n) Linear space complexity
Since we are creating an unordered_map to store unique elements and their count.

Q8. Given an array of pairs(2D array), find all symmetric pairs in it. Two pairs (a, b) and (c, d) are said to be symmetric if b and c are equal, and a is equal to d. It may be assumed that the first elements of all pairs are distinct. If no pair is present, print nothing. [Easy]

Input1:
n = 5
arr[][] = {{11, 20}, {30, 40}, {5, 10}, {40, 30}, {10, 5}}
Output1:
30 40
5 10

Input2:
n = 3
arr[] = {{11, 20}, {30, 40}, {10, 5}}
Output2:
0

A8. Solution Code: https://pastebin.com/6LmAc1Ky
Code explanation:

- Use a map or unordered_map here.
- The first element of the pair is used as the key and the second element is used as the value.
- Traverse all pairs one by one.
- For every pair, check if its second element is in the unordered_map or not.
- If yes, then compare the first element with the value of the matched entry of the unordered_map.
- If the value and the first element match, then we found symmetric pairs. Print the pair present in the unordered_map as it is the original pair.
- Else, insert the first element as a key and the second element as a value.

Output:

```
Enter the number of 2D pairs: 5
Enter the pairs:
11 20
30 40
5 10
40 30
10 5

Following pairs have symmetric pairs:
(30, 40)
(5, 10)
```

```
Enter the number of 2D pairs: 3
Enter the pairs:
11 20
30 40
10 5

Following pairs have symmetric pairs:
```

Time complexity: O(n) Linear time complexity
Here we are traversing the vector once in order to store the element and find its corresponding symmetric pair.
Space complexity: O(n) Linear space complexity
Since we are creating an unordered_map to store key-value pairs in it.

Q9. Given two arrays A[] and B[] consisting of n and m elements respectively. Find the minimum number of elements to remove from each array such that no common elements exist in both. [Medium]

Input1:
n = 4
A[] = { 1, 2, 3, 4 }
m = 5
B[] = { 2, 3, 4, 5, 8 }
Output1:
3
Input2:
n = 4
A[] = { 1, 2, 3, 4 }
m = 3
B[] = { 5, 6, 7 }
Output2:
0

A9. Solution Code: https://pastebin.com/kZWHJVT9
Code explanation:

- Use an unordered_map to store elements of the first array.
- Each unique element of the first array will be the key and its value will be its frequency.
- Maintain a count variable, initialized with 0 which will keep track of the same elements of both arrays.
- Now, traverse over the second array, and for every element, if it is present in the map, reduce its frequency by 1 and increase the count by 1.
- If now the frequency becomes 0, remove the element from the unordered_map as it means that the element is now only present in the second array.
- In the end, print count.

Output:

```
n = 4
a = 1 2 3 4
m = 5
b = 2 3 4 5 8
Minimum number of elements = 3
```

```
n = 4
a = 1 2 3 4
m = 3
b = 5 6 7
Minimum number of elements = 0
```

Time complexity: O(n+m) Linear time complexity
Here we are traversing the vectors a and b once in order to store the element and its corresponding count in the unordered_map.

Space complexity: O(n+m) Linear space complexity
Since we are creating two unordered_map to store elements and their count.

Q10. Consider a registration system. Each time a new user wants to register, he sends to the system a request with his name. If such a name does not exist in the system database, it is inserted into the database, and the user gets the response OK, confirming the successful registration. If the name already exists in the system database, the system makes up a new user name, sends it to the user as a prompt, and also inserts the prompt into the database. The new name is formed by the following rule. Numbers, starting with 1, are appended one after another to name (name1, name2, ...), among these numbers the least i is found so that name does not yet exist in the database.
Given n names, you need to register these names in the database according to the rules and print the prompt for every registration.

Input1:
6
alice
bob
alice
alice
alice
bob
Output1:
OK
OK
alice1
alice2
alice3
bob1

Input2:
8
first
first
third
second
third
second
third
third
Output2:
OK
first1
OK
OK
third1
second1
third2
third3

A10. Solution Code: https://pastebin.com/5ScRst0Z
Code explanation:

- The system here enacts the functioning of a map.
- We will input names one by one and for name, we need to check if it is already present in the map or not.
- If it is not present, it means we are registering this user for the first time, so we insert the name along with its occurrence, i.e. 1 in the map, and print "OK".
- Else, we need to first print the name followed by the value of the key, i.e. the current number of occurrences. Now, increment the value and place it back in the map.

Output:

```
Enter the number of requests: 6
Enter the requests:
alice
bob
alice
alice
alice
bob

The Prompt:
OK
OK
alice1
alice2
alice3
bob1
```

```
Enter the number of requests: 8
Enter the requests:
first
first
third
second
third
second
third
third

The Prompt:
OK
first1
OK
OK
third1
second1
third2
third3
```

Time complexity: O(n) Linear time complexity
Here we are traversing the vector of strings once in order to store the string and its corresponding count in the unordered_map.
Space complexity: O(n) Linear space complexity
Since we are creating an unordered_map to store unique strings and their count.

Q11. You are given an array of n elements. You have to make subsets from the array such that no subset

contains duplicate elements. Find out the minimum number of subsets possible. [Easy]

Input1:
n = 4
arr[ ] = {1, 2, 3, 4}
Output1:
1
Input2:
n = 6
arr[ ] = {1, 2, 3, 3, 2, 2}
Output2:
3

A11. Solution Code: https://pastebin.com/rSYmx4mq
Code explanation:

- We basically need to find the most frequent element in the array.
- The minimum number of subsets will be equal to the frequency of the most frequent element in the array.
- We will use an unordered_map to store the frequency of every element because the order doesn't matter.
- Traverse over the unordered_map and update the max pointer whenever the value is greater than it.
- Initialize ans with 1 as it is the minimum frequency and in the end, if all elements are unique, we will directly be returning 1 which will be correct, as at least 1 subset will be formed even from all distinct elements.

Output:

```
Enter the number of elements: 4
Enter the elements: 1 2 3 4
Minimum number of subsets: 1
```

```
Enter the number of elements: 6
Enter the elements: 1 2 3 3 2 2
Minimum number of subsets: 3
```

Time complexity: O(n) Linear time complexity
Here we are traversing the vector once in order to store the element and its corresponding count in the unordered_map.
Space complexity: O(n) Linear space complexity
Since we are creating an unordered_map to store unique elements and their count.

Q12. Given an array of size N and an integer K, return the count of distinct numbers in all windows of size K. [Medium]

Input1:
n = 7
arr[ ] = {1, 2, 1, 3, 4, 2, 3}
k = 4
Output1:
3 4 4 3

Input2:
n = 4
arr[ ] = {1, 2, 4, 4}
k = 2

Output2:
2 2 1

A12. Solution Code: https://pastebin.com/Dk2JaNUE
Code explanation:

- The trick is to use the count of the previous window while sliding the window.
- To do this a map can be used that stores elements of the current window.
- The map is also operated on by simultaneous addition and removal of an element while keeping track of distinct elements.
- Elements from index i to i + k – 1 will be stored in a map as an element-frequency pair. So, while updating the map in range i + 1 to i + k, reduce the frequency of the i-th element by 1 and increase the frequency of (i + k)-th element by 1.
- Initialize the count of distinct elements as count to 0.
- Traverse through the first window and insert elements of the first window to m. The elements are used as key and their counts as the value in m. Also, keep updating the count.
- Print distinct count for the first window.
- Traverse through the remaining array (or other windows).
- Remove the first element of the previous window, if the removed element appeared only once, remove it from map and decrease the distinct count, else (appeared multiple times in map), then decrement its count in map.
- Add the current element (last element of the new window), If the added element is not present in map, add it to map and increase the distinct count, else (the added element appeared multiple times), increment its count in map.

Output:

```
Enter the number of elements: 7
Enter the elements: 1 2 1 3 4 2 3
Enter the size of window: 4
The count of distinct number in all windows: 3 4 4 3
```

```
Enter the number of elements: 4
Enter the elements: 1 2 4 4
Enter the size of window: 2
The count of distinct number in all windows: 2 2 1
```

Time complexity: O(N), A single traversal of the array is required.
Space complexity: O(N), Since the hashmap requires linear space.

Q13. Given a string s, find the length of the longest substring without repeating characters. [Medium]

Input1:
s = "abcabcbb"
Output1:
3

Input2:
s = "bbbbb"
Output2:
1

A13. Solution Code: https://pastebin.com/ktErQPrB
Code explanation:

- We will use a sliding window approach here.
- Use 2 pointer left and right to traverse over windows and ans to keep track of the longest substring without repeating characters.
- Use a map that will store frequency of each character in a window.
- Start with left and right from 0.
- Increment right in a while loop until right is in index bounds.
- Everytime add character at index right to the map, if already present increase its frequency by 1.
- While frequency of character at index right is greater than 0, it means that this is the element getting repeated in this window, so we increase left until frequency of element at right index becomes 1.
- At this point, all elements in this window are unique, so update ans if length of this window is the longest so far.
- In the end, print *ans*.

Output:

```
Enter the string: abcabcbb
The length is: 3
```

```
Enter the string: bbbb
The length is: 1
```

Time complexity: O(n + 26) ~ O(n) where n is length of the input string.
Space complexity: O(26) Constant space complexity. Since we are creating map to store unique elements i.e. English alphabets.

Q14. Given a string s consisting only of characters a, b and c, print the number of substrings containing at least one occurrence of all these characters a, b and c. [Medium]

Input1:
s = "abcabc"
Output1:
10
Explanation:
The substrings containing at least one occurrence of the characters a, b and c are "abc", "abca", "abcab", "abcabc", "bca", "bcab", "bcabc", "cab", "cabc" and "abc" (again).

Input2:
s = "abc"
Output2:
1

A14. Solution Code: https://pastebin.com/bwGPZ6DH
Code explanation:

- Use a sliding window approach here.
- Use a map here where its size will specify the number of distinct characters in the current window.
- Intitialize pointers l, r with 0, signifying the left and right index of current window.
- Keep track of total count of such substrings with a variable count.
- Run a while loop with condition that r is less than s.length().
- Put the current character at index r in the map, if already present increase its frequency.
- Run a while loop again until size of map is 3.

- Increase the count and add all substrings after right pointer to current answer since those also satisfy the problem condition.
- Shrink the window size by incrementing left pointer but before that, decrease the frequency of element at current l index, if frequency is 0, remove it from the map.
- Increment r after inner while loop.
- Print count in the end.

Output:

```
Enter the string: abcabc
The number of Substrings = 10
```

```
Enter the string: abc
The number of Substrings = 1
```

Time complexity: O(n) Linear time complexity, where n is the size of the string.
Space complexity: O(3) ~ O(1) Constant space complexity
Since we are only going to have 3 keys and their values in map i.e. a,b,c.

Q15. Given an unsorted array of integers, find a subarray that adds to a given number. If there is more than one subarray with the sum of the given number, print any of them. [Medium]

Input1:
n = 6
arr[ ] = {1, 4, 20, 3, 10, 5}
sum = 33
Output1:
Sum found between indexes 2 and 4
Explanation: Sum of elements between indices 2 and 4 is 20 + 3 + 10 = 33

Input2:
n = 5
arr[ ] = {10, 2, -2, -20, 10}
sum = -10
Output2:
Sum found between indexes 0 and 3

A15. Solution Code: https://pastebin.com/Z97ZBWBb
Code explanation:

- We will use an unordered_map here.
- We will store the sum of all prefix arrays in this unordered_map.
- So to check if there is a subarray with a sum equal to s, check for every index i, and sum up to that index as x.
- If there is a prefix with a sum equal to (x – s), then the subarray with the given sum is found.
- Traverse through the array from start to end.
- For every element update the sum, i.e sum = sum + array[i]
- If the sum is equal to s then print that the subarray with the given sum is from 0 to i.
- If there is any key in the unordered_map which is equal to sum – s then print that the subarray with the given sum is from map[sum – s] to i.
- Put the sum and index in the unordered_map as a key-value pair.

Output:

```
Enter the number of elements: 6
Enter the elements: 1 4 20 3 10 5
Enter the sum: 33
Sum found between indexes 2 to 4
```

Time complexity: O(n) Linear time complexity as we are traversing the vector.
Space complexity: O(n) Linear space complexity as we are creating an unordered_map here