Q1 For a composition with different kinds of words, try to find the top k frequent words from the composition.

Solution: (Yize Liu)

Step1: iterate over the composition, and count each word and its frequency

<**key**=string word, **value** = int counter>

Step2: use a MIN-heap of size k. First, we iterate over all <key, value> from the hash_table, and put the first k pair into the MIN-heap. Second, from k+1-th pair to the n-th pair, we compare MIN-heap.top().value with **value_i** of the current pair <key_i, value_i>

Case1: if MIN-heap.top().value < value_i, then we call **MIN-heap.pop()** and then **MIN-Heap.insert(<key_i, value_i>)**

Case2: do nothing.

Q2. If there is **only one missing number** from 1 to n in an **unsorted** array. How to find it in O(n) time? size of the array is n-1.

Solution1: (Jiangchen Zheng)

Step1: store all numbers into the hash_set

Step2: for (int i = 1…. n) {

check whether i is in the hash_set or not

}

Time =O(n) Space = O(n)

Solution2: find SUM

Step1: expected sum = (1+n) *n/ 2

calculate real sum by using for loop → real_sum

Missing number = expected_sum - real_sum

Time = O(n) Space =O(1) cons: maybe overflow

Solution3: XOR

(1) Number1 XOR Number1 == 0

(2) Number1 XOR Number2 == Number2 XOR Number1

sequence1 : 1 2 3 4 5 6 7 8 9

sequence2: 1 2 3 4 _ 6 7 8 9

Q3. Find the common numbers between two sorted arrays a[N], b[M], N, M

Assumptions:

1. sorted
2. size is large but can be accommodated in 1 machine
3. size a.size ~~~ b.size

Method1:

two pointers.
谁小移动谁
Time = O(n+m)
Space = O(1)

Method2:

size a.size <<<<<< b.size

Run binary search for each element X in a[N], we run binary search in B[M] to check whether X is in B

Time = O(nlogm)
Space = O(1)

Method3:

HashTable

size a.size <<<<<< b.size

Step1: hash all numbers from a into hash_table

Step2: iterate each number X in b, to check whether X is in the hashset or not.

Time = O(n+m) Space = O(**min**(m, n))

Drawback1: erase API is very expensive!

常见错误分析： when 1st U, u1 is removed (by calling input.erase) all the rest chars after i are shifted to left by one, that is, **u2 is moved to [2], e move to [3]** … etc.

so, the consequence is the u2 will not be removed after i++ after this iteration.

Example:

index	0	1	2	3	4	5	6
s[7]	**s**	**t**	**u1**	**u2**	**d**	**n**	**t**

i (slow)-->

j (fast) →

i: slow all letters to the left-hand side of i are the results to return (**not including i**)

j: fast current index

Q2.2 (**Char de-duplication adjacent letters repeatedly**) abbbbaz → abbbbaz → z
ababa

```
a1 bbbb a2 zw    → a1a2 zw → zw                Time= O(n)
                 f→
```

Saixiong

Stack1|| z |w

```
// use stack to check every char, if duplicated, then pop out from the stack.
00 void removeDuplicate(string& s) {
01   if(s.length() <= 1)
02         return;
03   vector<char> st; //  this is our stack
04   int i = 0;
05   while(i < s.size()) {       // i is the fast index
06         char c = s[i];
07         if (st.size() > 0 && s[i] == st.back()) {
08               while (i < s.size() && c == s[i]) {
09                  i++;
10               }
11               st.pop_back();
12         } else {
13               st.push_back(s[i]);
14               i++;
15         }
16   }
17   s.clear();                          stack|| zw
18   for (int j = 0; j < st.size(); j++) {
19      s += st[j];
20   }
21 }
```

Method2 (Advanced version) we do not maintain the stack explicitly

```
     -1    0   1   2 3    4    5  6 7
input =  "z w b2 b3 b4   a2 z w"
           s
                                  f→
```

s (slow) all letters to the left hand side of slow are the results to return (including s). **Essentially, s is pointing to the top element of the (implicitly maintained stack)**

f→ current index.

M2:

Rabin-Karp

s1= "a b c d e" **s2 = "c d"; = 106**

a b = 87

b c = 99

c d = 106

d e ⇒ 101

principle: if we can hash the short string to a hashed value (e.g., integer, which is unique). Then we can just compare each substring of s1's hashed value and compare it with s2's hashvalue.

Assumption: only lower case letter (base = 26 a--z)

```
index    1 0
         c d
a=1
b=2
c=3
d=4
…
z= 26
      a  b        =  1* 26^1  + 2*26^0
           c
```

ab ->hashed to 1*26^1 + 2*26^0 => hashed value and then compare it with cd's hashed value ===> not equal,

ab ->bc when we increment index by 1 to the right, a is removed, and b's index changes from 0 to 1, and then we add c to the right most position

a b

1*26^1 + 2*26^0

b c

2*26^1 + 3*26^0

1. remove the leftmost item from the polynomial function

2. all the rest items of (ab's hashed value) x 26

3. add new item c

For text of length *n* and *1* pattern of length *m*, its **average** and **best case** **running time** is O(*n*+*m*) in space O(*1*), but its **worst-case time** is **O(*nm*) (when hash all matches)**

Things to worry about:

overflow: hashed value is too big to be represented by 64 bit....

application: to catch plagiarism

`hashed value of "c d" = 3*26^1 + 4*26^0 == > x`

s1 = abc

s1' = abc hash(s1) = hash(s1') == integer_value 1

hash(s2) == maybe ⇒ integer value 1

s2 = "zzz"