2259. Remove Digit From Number to Maximize Result

Example 1:

Input: number = "123", digit = "3"

Output: "12"

Explanation: There is only one '3' in "123". After removing '3', the result is "12".

✅ Summary:

This function:

- Removes **one occurrence** of a given digit from a string number
- Tries **all possible removals** of that digit
- Returns the **lexicographically largest** resulting number (as a string)

📗 Key Concepts to Learn & Write in Notes:

🧩 1. Lexicographical Comparison of Strings:

- **Strings can be compared directly in C++ using > and <**
- **'91' > '90' ✅ but '9' > '10' ✅ too (because it's string comparison, not numeric)**

🧩 2. String Substring Operations:

string s1 = number.substr(0, i);

string s2 = number.substr(i + 1);

- **Use .substr(start, length) to extract parts of a string**
- **Concatenation: s = s1 + s2**

🧩 3. Greedy Approach:

- Try removing each occurrence of digit
- At each step, update the answer if the new string is **lexicographically greater**

🧩 4. Storing Best Result:

if (s > ans) ans = s;

- Keep the **best (largest)** string seen so far

📝 **Notes Version:**

```
// Function to remove one occurrence of a digit
// and return the lexicographically largest result
string removeDigit(string number, char digit) {
  string ans = "";
  for (int i = 0; i < number.length(); i++) {
    if (number[i] == digit) {
      string s = number.substr(0, i) + number.substr(i + 1);
      if (s > ans) ans = s;  // update if current result is better
    }
  }
  return ans;
}
```

✨ **Use Case Example:**

Input: number = "133235", digit = '3'

Output: "13325"

number.substr(i + 1);

Starts extracting from index i + 1

=>

Takes all characters till the end of the string

JAVA CODE:

```
public String removeDigit(String number, char digit) {
  String result = "";
  for (int i = 0; i < number.length(); i++) {
    if (number.charAt(i) == digit) {
      String candidate = number.substring(0, i) + number.substring(i + 1);
      if (candidate.compareTo(result) > 0) {
        result = candidate;
```

```
      }
    }
  }
  return result;
}
```

[2260. Minimum Consecutive Cards to Pick Up]

✅ **Correct Approach (Your Final Code)**

🧠 **Goal:**

Find the **minimum length of a subarray** that contains **two equal cards**.

📘 **Core Idea:**

- Use a **hash map** to store the **last index** where each card was seen.
- If you find a **duplicate card**, calculate the distance between the current and last index → update the minimum length.

📌 **Steps:**

1. **Create unordered_map<int, int> lastSeen to store card → last index.**
2. **Loop through cards:**
   - **If card was seen before:**
     - **Calculate i - lastSeen[card] + 1**
     - **Update minLen if smaller**
   - **Update lastSeen[card] = i**
3. **After loop:**
   - **If minLen == INT_MAX → no duplicates → return -1**
   - **Else → return minLen**

🧪 **Time & Space:**

- **Time:** O(n)
- **Space:** O(n) for map

✅ Code Snippet to Remember:(hashmap)

```
int minimumCardPickup(vector<int>& cards) {
    unordered_map<int, int> lastSeen;
int minLen = INT_MAX;

for (int i = 0; i < cards.size(); i++) {
  if (lastSeen.find(cards[i]) != lastSeen.end()) {
    int prevIndex = lastSeen[cards[i]];
    minLen = min(minLen, i - prevIndex + 1);
  }
  lastSeen[cards[i]] = i;  // update latest index
}

return (minLen == INT_MAX) ? -1 : minLen;
}
```

✅ Correct Sliding Window Code (Modified & Working):

```
int minimumCardPickup(vector<int>& cards) {
  unordered_map<int, int> freq;
  int left = 0, minLen = INT_MAX;

  for (int right = 0; right < cards.size(); right++) {
    freq[cards[right]]++;

    while (freq[cards[right]] > 1) {
      minLen = min(minLen, right - left + 1);
      freq[cards[left]]--;
      left++;
    }
  }
```

```
  return (minLen == INT_MAX) ? -1 : minLen;
}
```

🧠 **How It Works:**

- The right pointer expands the window.
- If a **duplicate** is found (i.e., freq[cards[right]] > 1), shrink the window from the left side until the duplicate is gone.
- • As we move the `right` pointer forward, we track how many times we've seen each card in `freq`.
- • The moment `freq[cards[right]] > 1`, it means we now have **a duplicate** of this card **in the current window** from `left` to `right`.
- Each time a duplicate is found, calculate the **window size** and update minLen.

⏱ **Time and Space:**

- **Time:** O(n) – each element is visited at most twice (once by right, once by left)
- **Space:** O(n) for the frequency map

Sliding window Approach (Generally)

let l, r be 2 pointers, proceed the standard nested loop as follows

```
for (int r=0, l=0; r<n; r++) {
  do_something_by_adding(nums[r]);

  // Try to move the left pointer to maintain at least k pairs
  while (test_condition(k)) {
    // valid subarrays among [l, r0] where r0=r...n-1
    update(ans);
    do_something_by_removing(nums[l]);
  }
}
```

Problem:

You're given an array A of length N and an integer K.

A subarray from index l to r is called good if it contains at most K distinct elements. An empty subarray is also considered good and has a sum of 0.

You need to find the maximum sum of any good subarray.

✅ **Sample Input 1:**

11 2

HERE 11=N AND K=2

2 2 3 3 4 4 5 1 1 1 1

✅ **Sample Output 1:**

12

🔍 **Intuition:**

This is a classic sliding window + hash map problem where the goal is to maintain a window that:

1. Contains at most K distinct elements, and
2. Has the maximum possible sum.

Here's how the sliding window works:

- Use two pointers left and right to represent the current subarray.
- Use a hashmap (unordered_map<int,int>) to count the frequency of elements in the current window.
- Expand the right pointer to include new elements and add them to the current sum.
- If the number of distinct elements exceeds K, shrink the window from the left until the condition is valid again.
- Keep track of the maximum valid subarray sum seen so far.

🧠 **Step-by-Step Example (Sample Input):**

Input: A = [2,2,3,3,4,4,5,1,1,1,1], K = 2

Process:

- **Start with left = 0, right = 0**
- **Slide right to include elements while maintaining at most 2 distinct numbers.**
- **Every time the window is valid, compute the sum and update the max if needed.**
- **If the window becomes invalid (more than 2 distinct elements), slide left until it's valid again.**

✅ **Code Analysis:**

```
unordered_map<int,int> freq;

int left=0, sum=0, max_sum=0;

for(int right=0; right<n; right++){

  freq[A[right]]++;

  sum += A[right];

  while(freq.size() > k){

    freq[A[left]]--;

    sum -= A[left];

    if(freq[A[left]] == 0)

      freq.erase(A[left]);

    left++;

  }

  max_sum = max(max_sum, sum);

}
```

Time Complexity: O(N)
Space Complexity: O(K) (for the map)

🔍 **Code in Context:**

You're using a sliding window with a frequency map (unordered_map<int, int> freq) to track the number of distinct elements in the current window.

Here's the key part you're asking about:

```
if(freq[A[left]] == 0)

  freq.erase(A[left]);
```

☑ Purpose of this line:

This line is essential to keep the freq map accurate — specifically, to ensure that:

☑ freq.size() always reflects the actual number of distinct elements in the current window.

🧠 Why it's needed:

Let's say A[left] = 5, and this is the only 5 in the current window.
Then freq[5] = 1.

When you shrink the window:

freq[A[left]]--; // freq[5] becomes 0

But now, 5 is no longer in the window, yet it's still sitting in the map with value 0.

If you don't remove it, then:

if(freq.size() > k)

...will still count 5 as present, which is wrong.

[2537. Count the Number of Good Subarrays]

C++ CODE:(Sliding window)

```
long long countGood(vector<int>& nums, int k) {
     long long ans=0;
    int left=0;
    int pairs=0;
    int n=nums.size();
     unordered_map<int,int> mp;
    for(int right=0;right<n;right++){
     pairs += mp[nums[right]];
     mp[nums[right]]++;
     while(pairs>=k){
        ans+=n-right;
```

```
      mp[nums[left]]--;
      pairs-=mp[nums[left]];
      left++;
    }
   }
   return ans;
 }
```

Why this line pairs -= mp[nums[left]]; ??

Ye line tab chalti hai jab hum window ko chhota kar rahe hote hain (i.e. left++ kar rahe hain), matlab left se koi element hata rahe hain.

✅ **Jab ek element add karte hain:**

- usse pehle uska count 'f' hota hai

- wo f naye good pairs banata hai → pairs += f

✅ **Jab ek element remove karte hain:**

- pehle mp[x]--

- ab wo element 'f' baar bacha → pairs -= f

Ye ensure karta hai ki 'pairs' variable hamesha sahi number of good pairs show kare.

❓ **What does ans += n - right; mean? Why are we doing it?**

🧠 **Context:**

This line appears inside the while(pairs >= k) loop, where:

- **You're using a sliding window from left to right**
- **You've just found a window where the number of good pairs ≥ k**
- **Your goal: count how many subarrays starting at left (or after) are valid**

🎯 **Key Insight:**

If a window [left, right] is valid (i.e., has at least k good pairs), then any extension of this window (like [left, right+1], [left, right+2], ..., [left, n-1]) will also be valid — because adding more elements can only increase or maintain the number of good pairs.

So, once you find a valid window, the number of valid subarrays starting at left is:

n - right

Why?

- **right is the current last index of the window.**
- **From right to n-1, all subarrays [left, right], [left, right+1], ..., [left, n-1] are valid.**