LeetCode Training Day 9 Queue and Stack

We all know queue is a first in, first out data structure and stack is last in first out. In algorithm we often use queue as BFS and stack for DFS (which is implicitly under recursive function).

There are other utilizations for queue and stack. Queue can be used as a slide window and stack is also known as monotone stack.

Monotone stack is that we try to keep a series of elements in as ascending order or descending order to get a minimum or maximum permutation.

Both queue and stack can be used to pair the opposite elements to cancel each other. It depends on which one you want to cancel, in case of cancelling earliest ones, you use queue, in case of cancelling recent ones, you use stack.

Stack, if used in expression parsing, can also change the process order as we know, and this scenario will be discussed in later chapters.

## 1475. Final Prices With a Special Discount in a Shop

Easy

Given the array prices where prices[i] is the price of the ith item in a shop. There is a special discount for items in the shop, if you buy the ith item, then you will receive a discount equivalent to prices[j] where j is the **minimum** index such that j > i and prices[j] <= prices[i], otherwise, you will not receive any discount at all.

*Return an array where the ith element is the final price you will pay for the ith item of the shop considering the special discount.*

**Example 1:**

**Input:** prices = [8,4,6,2,3]

**Output:** [4,2,4,2,3]

**Explanation:**

For item 0 with price[0]=8 you will receive a discount equivalent to prices[1]=4, therefore, the final price you will pay is 8 - 4 = 4.

For item 1 with price[1]=4 you will receive a discount equivalent to prices[3]=2, therefore, the final price you will pay is 4 - 2 = 2.

For item 2 with price[2]=6 you will receive a discount equivalent to prices[3]=2, therefore, the final price you will pay is 6 - 2 = 4.

For items 3 and 4 you will not receive any discount at all.

**Example 2:**

**Input:** prices = [1,2,3,4,5]

**Output:** [1,2,3,4,5]

**Explanation:** In this case, for all items, you will not receive any discount at all.

**Example 3:**

**Input:** prices = [10,1,1,6]

**Output:** [9,0,1,6]

**Constraints:**

1 <= prices.length <= 500

1 <= prices[i] <= 10^3

### Analysis:

For every price, find a higher or equal price before and discount it. Keep all unprocessed price in stack

/// <summary>

/// Leet code #1475. Final Prices With a Special Discount in a Shop

///

/// Easy

///

/// Given the array prices where prices[i] is the price of the ith item

/// in a shop. There is a special discount for items in the shop, if you

/// buy the ith item, then you will receive a discount equivalent to

/// prices[j] where j is the minimum index such that j > i and

/// prices[j] <= prices[i], otherwise, you will not receive any discount

/// at all.

///

/// Return an array where the ith element is the final price you will pay

/// for the ith item of the shop considering the special discount.

///

/// Example 1:

/// Input: prices = [8,4,6,2,3]

/// Output: [4,2,4,2,3]

/// Explanation:

/// For item 0 with price[0]=8 you will receive a discount equivalent to

/// prices[1]=4, therefore, the final price you will pay is 8 - 4 = 4.

/// For item 1 with price[1]=4 you will receive a discount equivalent to

/// prices[3]=2, therefore, the final price you will pay is 4 - 2 = 2.

/// For item 2 with price[2]=6 you will receive a discount equivalent to

/// prices[3]=2, therefore, the final price you will pay is 6 - 2 = 4.

/// For items 3 and 4 you will not receive any discount at all.

///

/// Example 2:

/// Input: prices = [1,2,3,4,5]

/// Output: [1,2,3,4,5]

/// Explanation: In this case, for all items, you will not receive any

/// discount at all.

/// Example 3:

/// Input: prices = [10,1,1,6]

/// Output: [9,0,1,6]

/// Constraints:

/// 1. 1 <= prices.length <= 500

/// 2. 1 <= prices[i] <= 10^3

/// </summary>

vector<int> LeetCodeStack::finalPrices(vector<int>& prices)

{

vector<int> result;

stack<pair<int, int>> stack;

for (size\_t i = 0; i < prices.size(); i++)

{

while ((!stack.empty()) && (stack.top().first >= prices[i]))

{

result[stack.top().second] -= prices[i];

stack.pop();

}

stack.push(make\_pair(prices[i], i));

result.push\_back(prices[i]);

}

return result;

}

## 735. Asteroid Collision

Medium

We are given an array asteroids of integers representing asteroids in a row.

For each asteroid, the absolute value represents its size, and the sign represents its direction (positive meaning right, negative meaning left). Each asteroid moves at the same speed.

Find out the state of the asteroids after all collisions. If two asteroids meet, the smaller one will explode. If both are the same size, both will explode. Two asteroids moving in the same direction will never meet.

**Example 1:**

**Input:**

asteroids = [5, 10, -5]

**Output:** [5, 10]

**Explanation:**

The 10 and -5 collide resulting in 10. The 5 and 10 never collide.

**Example 2:**

**Input:**

asteroids = [8, -8]

**Output:** []

**Explanation:**

The 8 and -8 collide exploding each other.

**Example 3:**

**Input:**

asteroids = [10, 2, -5]

**Output:** [10]

**Explanation:**

The 2 and -5 collide resulting in -5. The 10 and -5 collide resulting in 10.

**Example 4:**

**Input:**

asteroids = [-2, -1, 1, 2]

**Output:** [-2, -1, 1, 2]

**Explanation:**

The -2 and -1 are moving left, while the 1 and 2 are moving right.

Asteroids moving the same direction never meet, so no asteroids will meet each other.

**Note:**

 The length of asteroids will be at most 10000.

 Each asteroid will be a non-zero integer in the range [-1000, 1000]..

### Analysis:

Keep track the asteroids in stack, and always compare with stack top, cancel each other can continue the process until stack top is at same direction of the new ball.

/// <summary>

/// Leet code #735. Asteroid Collision

///

/// We are given an array asteroids of integers representing asteroids in

/// a row.

/// For each asteroid, the absolute value represents its size, and the sign

/// represents its direction (positive meaning right, negative meaning

/// left). Each asteroid moves at the same speed.

///

/// Find out the state of the asteroids after all collisions. If two

/// asteroids meet, the smaller one will explode. If both are the same

/// size, both will explode. Two asteroids moving in the same direction

/// will never meet.

///

/// Example 1:

/// Input:

/// asteroids = [5, 10, -5]

/// Output: [5, 10]

/// Explanation:

/// The 10 and -5 collide resulting in 10. The 5 and 10 never collide.

///

/// Example 2:

/// Input:

/// asteroids = [8, -8]

/// Output: []

/// Explanation:

/// The 8 and -8 collide exploding each other.

///

/// Example 3:

/// Input:

/// asteroids = [10, 2, -5]

/// Output: [10]

/// Explanation:

/// The 2 and -5 collide resulting in -5. The 10 and -5 collide resulting

/// in 10.

///

/// Example 4:

/// Input:

/// asteroids = [-2, -1, 1, 2]

/// Output: [-2, -1, 1, 2]

/// Explanation:

/// The -2 and -1 are moving left, while the 1 and 2 are moving right.

/// Asteroids moving the same direction never meet, so no asteroids will

/// meet each other.

///

/// Note:

/// The length of asteroids will be at most 10000.

/// Each asteroid will be a non-zero integer in the range [-1000, 1000]..

/// </summary>

vector<int> LeetCodeStack::asteroidCollision(vector<int>& asteroids)

{

vector<int> result;

for (size\_t i = 0; i < asteroids.size(); i++)

{

if (result.empty())

{

result.push\_back(asteroids[i]);

}

else if ((result.back() > 0) && (asteroids[i] < 0))

{

while (!result.empty() && (result.back() > 0) && (asteroids[i] < 0))

{

if (abs(result.back()) == abs(asteroids[i]))

{

result.pop\_back();

break;

}

else if (abs(result.back()) > abs(asteroids[i]))

{

break;

}

else

{

result.pop\_back();

}

}

}

else

{

result.push\_back(asteroids[i]);

}

}

return result;

}

## 402. Remove K Digits

Medium

Given string num representing a non-negative integer num, and an integer k, return *the smallest possible integer after removing* k *digits from* num.

**Example 1:**

**Input:** num = "1432219", k = 3

**Output:** "1219"

**Explanation:** Remove the three digits 4, 3, and 2 to form the new number 1219 which is the smallest.

**Example 2:**

**Input:** num = "10200", k = 1

**Output:** "200"

**Explanation:** Remove the leading 1 and the number is 200. Note that the output must not contain leading zeroes.

**Example 3:**

**Input:** num = "10", k = 2

**Output:** "0"

**Explanation:** Remove all the digits from the number and it is left with nothing which is 0.

**Constraints:**

1 <= k <= num.length <= 105

num consists of only digits.

num does not have any leading zeros except for the zero itself.

### Analysis:

Start to pop up all bigger numbers ahead, in the end skip leading 0.

/// <summary>

/// Leet code #402. Remove K Digits

///

/// Given a non-negative integer num represented as a string, remove k digits

/// from the number so that the new number is the smallest possible.

/// Note:

/// The length of num is less than 10002 and will be ≥ k.

/// The given num does not contain any leading zero.

/// Example 1:

/// Input: num = "1432219", k = 3

/// Output: "1219"

/// Explanation: Remove the three digits 4, 3, and 2 to form the new number

/// 1219 which is the smallest.

///

/// Example 2:

/// Input: num = "10200", k = 1

/// Output: "200"

/// Explanation: Remove the leading 1 and the number is 200. Note that the

/// output must not contain leading zeroes.

///

/// Example 3:

/// Input: num = "10", k = 2

/// Output: "0"

/// Explanation: Remove all the digits from the number and it is left with

/// nothing which is 0.

/// </summary>

string LeetCodeStack::removeKdigits(string num, int k)

{

string result;

// pop up big leading digits from front

for (size\_t i = 0; i < num.size(); i++)

{

while (!result.empty() && (num[i] < result.back()) && k > 0)

{

result.pop\_back();

k--;

}

if (!result.empty() || num[i] != '0') result.push\_back(num[i]);

}

// pop up extra digits from end

while (!result.empty() && k > 0)

{

result.pop\_back();

k--;

}

size\_t i = 0;

result = result.substr(i);

if (result.empty()) result = "0";

return result;

}

## 739. Daily Temperatures

Medium

Given an array of integers temperatures represents the daily temperatures, return *an array* answer *such that* answer[i] *is the number of days you have to wait after the* ith *day to get a warmer temperature*. If there is no future day for which this is possible, keep answer[i] == 0 instead.

**Example 1:**

**Input:** temperatures = [73,74,75,71,69,72,76,73]

**Output:** [1,1,4,2,1,1,0,0]

**Example 2:**

**Input:** temperatures = [30,40,50,60]

**Output:** [1,1,1,0]

**Example 3:**

**Input:** temperatures = [30,60,90]

**Output:** [1,1,0]

**Constraints:**

1 <= temperatures.length <= 105

30 <= temperatures[i] <= 100

### Analysis:

On day x, you check all the previous days and see which days with temperature lower than today, and pop them up from stack, mark the result. In the end all the days without processed, are the ones you can not get a day warm than that day.

/// <summary>

/// Leet code #739. Daily Temperatures

///

/// Given a list of daily temperatures, produce a list that, for each day

/// in the input, tells you how many days you would have to wait until a

/// warmer temperature. If there is no future day for which this is

/// possible, put 0 instead.

/// For example, given the list temperatures = [73, 74, 75, 71, 69, 72,

/// 76, 73], your output should be [1, 1, 4, 2, 1, 1, 0, 0].

/// Note: The length of temperatures will be in the range [1, 30000]. Each

/// temperature will be an integer in the range [30, 100].

/// </summary>

vector<int> LeetCodeStack::dailyTemperatures(vector<int>& temperatures)

{

vector<int> result(temperatures.size());

stack<pair<int, int>> temp\_stack;

for (size\_t i = 0; i < temperatures.size(); i++)

{

while (!temp\_stack.empty())

{

auto temperature = temp\_stack.top();

// if new temperature is lower then simply push it.

if (temperature.first >= temperatures[i]) break;

// set the day and pop up

result[temperature.second] = i - temperature.second;

temp\_stack.pop();

}

temp\_stack.push(make\_pair(temperatures[i], i));

}

return result;

}

## 1003. Check If Word Is Valid After Substitutions

Medium

We are given that the string "abc" is valid.

From any valid string V, we may split V into two pieces X and Y such that X + Y (X concatenated with Y) is equal to V.  (X or Y may be empty.)  Then, X + "abc" + Y is also valid.

If for example S = "abc", then examples of valid strings are: "abc", "aabcbc", "abcabc", "abcabcababcc".  Examples of **invalid** strings are: "abccba", "ab", "cababc", "bac".

Return true if and only if the given string S is valid.

**Example 1:**

**Input:** "aabcbc"

**Output:** true

**Explanation:**

We start with the valid string "abc".

Then we can insert another "abc" between "a" and "bc", resulting in "a" + "abc" + "bc" which is "aabcbc".

**Example 2:**

**Input:** "abcabcababcc"

**Output:** true

**Explanation:**

"abcabcabc" is valid after consecutive insertings of "abc".

Then we can insert "abc" before the last letter, resulting in "abcabcab" + "abc" + "c" which is "abcabcababcc".

**Example 3:**

**Input:** "abccba"

**Output:** false

**Example 4:**

**Input:** "cababc"

**Output:** false

**Note:**

1. 1 <= S.length <= 20000
2. S[i] is 'a', 'b', or 'c'

### Analysis:

Keep track the string in stack and pop up "abc"

/// <summary>

/// Leet code #1003. Check If Word Is Valid After Substitutions

///

/// We are given that the string "abc" is valid.

///

/// From any valid string V, we may split V into two pieces X and Y such

/// that X + Y (X concatenated with Y) is equal to V. (X or Y may be empty.)

/// Then, X + "abc" + Y is also valid.

///

/// If for example S = "abc", then examples of valid strings are: "abc",

/// "aabcbc", "abcabc", "abcabcababcc". Examples of invalid strings are:

/// "abccba", "ab", "cababc", "bac".

///

/// Return true if and only if the given string S is valid.

///

///

/// Example 1:

///

/// Input: "aabcbc"

/// Output: true

/// Explanation:

/// We start with the valid string "abc".

/// Then we can insert another "abc" between "a" and "bc", resulting in

/// "a" + "abc" + "bc" which is "aabcbc".

///

/// Example 2:

///

/// Input: "abcabcababcc"

/// Output: true

/// Explanation:

/// "abcabcabc" is valid after consecutive insertings of "abc".

/// Then we can insert "abc" before the last letter, resulting in

/// "abcabcab" + "abc" + "c" which is "abcabcababcc".

///

/// Example 3:

///

/// Input: "abccba"

/// Output: false

///

/// Example 4:

///

/// Input: "cababc"

/// Output: false

///

///

/// Note:

///

/// 1. 1 <= S.length <= 20000

/// 2. S[i] is 'a', 'b', or 'c'

/// </summary>

bool LeetCodeStack::isValidAbc(string S)

{

string result;

for (size\_t i = 0; i < S.size(); i++)

{

result.push\_back(S[i]);

while ((result.size() >= 3) &&

(result.substr(result.size() - 3) == "abc"))

{

result.resize(result.size() - 3);

}

}

if (result.empty())

{

return true;

}

else

{

return false;

}

}

## 1081. Smallest Subsequence of Distinct Characters

Medium

Return the lexicographically smallest subsequence of text that contains all the distinct characters of text exactly once.

**Example 1:**

**Input:** "cdadabcc"

**Output:** "adbc"

**Example 2:**

**Input:** "abcd"

**Output:** "abcd"

**Example 3:**

**Input:** "ecbacba"

**Output:** "eacb"

**Example 4:**

**Input:** "leetcode"

**Output:** "letcod"

**Note:**

1. 1 <= text.length <= 1000
2. text consists of lowercase English letters.

### Analysis:

First you count all the distinct characters, then track the result string in stack, when you see a reverse order, if the previous character will appear again in later sequence, you know you can pop it up.

/// <summary>

/// Leet code #1081. Smallest Subsequence of Distinct Characters

///

/// Return the lexicographically smallest subsequence of text that contains

/// all the distinct characters of text exactly once.

///

/// Example 1:

/// Input: "cdadabcc"

/// Output: "adbc"

///

/// Example 2:

/// Input: "abcd"

/// Output: "abcd"

///

/// Example 3:

/// Input: "ecbacba"

/// Output: "eacb"

///

/// Example 4:

/// Input: "leetcode"

/// Output: "letcod"

///

/// Note:

///

/// 1. 1 <= text.length <= 1000

/// 2. text consists of lowercase English letters.

/// </summary>

string LeetCodeStack::smallestSubsequence(string text)

{

vector<int> count(26), used(26);

for (size\_t i = 0; i < text.size(); i++)

{

count[text[i] - 'a']++;

}

string result;

for (size\_t i = 0; i < text.size(); i++)

{

if (result.empty())

{

result.push\_back(text[i]);

used[x] = 1;

}

else

{

int x = text[i] - 'a';

if (used[x] == 1)

{

count[x]--;

continue;

}

while (!result.empty())

{

char ch = result.back();

if (text[i] > ch) break;

int k = ch - 'a';

if (count[k] == 1) break;

count[k]--;

used[k] = 0;

result.pop\_back();

}

result.push\_back(text[i]);

used[x] = 1;

}

}

return result;

}

## 1209. Remove All Adjacent Duplicates in String II

Medium

Given a string s, a *k* *duplicate removal* consists of choosing k adjacent and equal letters from s and removing them causing the left and the right side of the deleted substring to concatenate together.

We repeatedly make k duplicate removals on s until we no longer can.

Return the final string after all such duplicate removals have been made.

It is guaranteed that the answer is unique.

**Example 1:**

**Input:** s = "abcd", k = 2

**Output:** "abcd"

**Explanation:** There's nothing to delete.

**Example 2:**

**Input:** s = "deeedbbcccbdaa", k = 3

**Output:** "aa"

**Explanation:**

First delete "eee" and "ccc", get "ddbbbdaa"

Then delete "bbb", get "dddaa"

Finally delete "ddd", get "aa"

**Example 3:**

**Input:** s = "pbbcggttciiippooaais", k = 2

**Output:** "ps"

**Constraints:**

* 1 <= s.length <= 10^5
* 2 <= k <= 10^4
* s only contains lower case English letters.

### Analysis:

Keep character count in stack when reach limit, pop all of them out

/// <summary>

/// Leet code #1209. Remove All Adjacent Duplicates in String II

///

/// Given a string s, a k duplicate removal consists of choosing k adjacent

/// and equal letters from s and removing them causing the left and the

/// right side of the deleted substring to concatenate together.

///

/// We repeatedly make k duplicate removals on s until we no longer can.

///

/// Return the final string after all such duplicate removals have been made.

///

/// It is guaranteed that the answer is unique.

///

/// Example 1:

/// Input: s = "abcd", k = 2

/// Output: "abcd"

/// Explanation: There's nothing to delete.

///

/// Example 2:

/// Input: s = "deeedbbcccbdaa", k = 3

/// Output: "aa"

/// Explanation:

/// First delete "eee" and "ccc", get "ddbbbdaa"

/// Then delete "bbb", get "dddaa"

/// Finally delete "ddd", get "aa"

///

/// Example 3:

/// Input: s = "pbbcggttciiippooaais", k = 2

/// Output: "ps"

///

///

/// Constraints:

/// 1. 1 <= s.length <= 10^5

/// 2. 2 <= k <= 10^4

/// 3. s only contains lower case English letters.

/// </summary>

string LeetCodeStack::removeDuplicates(string s, int k)

{

string result;

vector<int> dp;

for (size\_t i = 0; i < s.size(); i++)

{

if (dp.empty())

{

result.push\_back(s[i]);

dp.push\_back(1);

}

else

{

if (result.back() == s[i])

{

dp.push\_back(dp.back()+ 1);

}

else

{

dp.push\_back(1);

}

result.push\_back(s[i]);

}

if (dp.back() == k)

{

result.resize(dp.size() - k);

dp.resize(dp.size() - k);

}

}

return result;

}

## 946. Validate Stack Sequences

Medium

Given two sequences pushed and popped **with distinct values**, return true if and only if this could have been the result of a sequence of push and pop operations on an initially empty stack.

**Example 1:**

**Input:** pushed = [1,2,3,4,5], popped = [4,5,3,2,1]

**Output:** true

**Explanation:** We might do the following sequence:

push(1), push(2), push(3), push(4), pop() -> 4,

push(5), pop() -> 5, pop() -> 3, pop() -> 2, pop() -> 1

**Example 2:**

**Input:** pushed = [1,2,3,4,5], popped = [4,3,5,1,2]

**Output:** false

**Explanation:** 1 cannot be popped before 2.

**Note:**

1. 0 <= pushed.length == popped.length <= 1000
2. 0 <= pushed[i], popped[i] < 1000
3. pushed is a permutation of popped.
4. pushed and popped have distinct values.

### Analysis:

If stack top matches the second sequence, pop it, otherwise push a new one.

/// <summary>

/// Leet code #946. Validate Stack Sequences

///

/// Given two sequences pushed and popped with distinct values, return

/// true if and only if this could have been the result of a sequence

/// of push and pop operations on an initially empty stack.

///

/// Example 1:

/// Input: pushed = [1,2,3,4,5], popped = [4,5,3,2,1]

/// Output: true

/// Explanation: We might do the following sequence:

/// push(1), push(2), push(3), push(4), pop() -> 4,

/// push(5), pop() -> 5, pop() -> 3, pop() -> 2, pop() -> 1

///

/// Example 2:

/// Input: pushed = [1,2,3,4,5], popped = [4,3,5,1,2]

/// Output: false

/// Explanation: 1 cannot be popped before 2.

///

/// Note:

///

/// 1. 0 <= pushed.length == popped.length <= 1000

/// 2. 0 <= pushed[i], popped[i] < 1000

/// 3. pushed is a permutation of popped.

/// 4. pushed and popped have distinct values.

/// </summary>

bool LeetCodeStack::validateStackSequences(vector<int>& pushed, vector<int>& popped)

{

size\_t index1 = 0;

size\_t index2 = 0;

stack<int> work\_stack;

while (index1 < pushed.size() || (!work\_stack.empty() && work\_stack.top() == popped[index2]))

{

if (work\_stack.empty() || work\_stack.top() != popped[index2])

{

work\_stack.push(pushed[index1]);

index1++;

}

else

{

work\_stack.pop();

index2++;

}

}

if (index2 == popped.size()) return true;

else return false;

}

## 1989. Maximum Number of People That Can Be Caught in Tag

Medium

You are playing a game of tag with your friends. In tag, people are divided into two teams: people who are "it", and people who are not "it". The people who are "it" want to catch as many people as possible who are not "it".

You are given a **0-indexed** integer array team containing only zeros (denoting people who are **not** "it") and ones (denoting people who are "it"), and an integer dist. A person who is "it" at index i can catch any **one** person whose index is in the range [i - dist, i + dist] (**inclusive**) and is **not** "it".

Return *the****maximum****number of people that the people who are "it" can catch*.

**Example 1:**

**Input:** team = [0,1,0,1,0], dist = 3

**Output:** 2

**Explanation:**

The person who is "it" at index 1 can catch people in the range [i-dist, i+dist] = [1-3, 1+3] = [-2, 4].

They can catch the person who is not "it" at index 2.

The person who is "it" at index 3 can catch people in the range [i-dist, i+dist] = [3-3, 3+3] = [0, 6].

They can catch the person who is not "it" at index 0.

The person who is not "it" at index 4 will not be caught because the people at indices 1 and 3 are already catching one person.

**Example 2:**

**Input:** team = [1], dist = 1

**Output:** 0

**Explanation:**

There are no people who are not "it" to catch.

**Example 3:**

**Input:** team = [0], dist = 1

**Output:** 0

**Explanation:**

There are no people who are "it" to catch people.

**Constraints:**

1 <= team.length <= 105

0 <= team[i] <= 1

1 <= dist <= team.length

### Analysis:

Push tag 0 people in queue and captured by tag 1 people, when out of range, pop out.

/// <summary>

/// Leet Code 1989. Maximum Number of People That Can Be Caught in Tag

///

/// Medium

///

/// You are playing a game of tag with your friends. In tag, people are

/// divided into two teams: people who are "it", and people who are not

/// "it". The people who are "it" want to catch as many people as

/// possible who are not "it".

///

/// You are given a 0-indexed integer array team containing only zeros

/// (denoting people who are not "it") and ones (denoting people who are

/// "it"), and an integer dist. A person who is "it" at index i can catch

/// any one person whose index is in the range [i - dist, i + dist]

/// (inclusive) and is not "it".

///

/// Return the maximum number of people that the people who are "it" can

/// catch.

///

/// Example 1:

/// Input: team = [0,1,0,1,0], dist = 3

/// Output: 2

/// Explanation:

/// The person who is "it" at index 1 can catch people in the range

/// [i-dist, i+dist] = [1-3, 1+3] = [-2, 4].

/// They can catch the person who is not "it" at index 2.

/// The person who is "it" at index 3 can catch people in the range

/// [i-dist, i+dist] = [3-3, 3+3] = [0, 6].

/// They can catch the person who is not "it" at index 0.

/// The person who is not "it" at index 4 will not be caught because the

/// people at indices 1 and 3 are already catching one person.

///

/// Example 2:

/// Input: team = [1], dist = 1

/// Output: 0

/// Explanation:

/// There are no people who are not "it" to catch.

///

/// Example 3:

/// Input: team = [0], dist = 1

/// Output: 0

/// Explanation:

/// There are no people who are "it" to catch people.

///

/// Constraints:

/// 1. 1 <= team.length <= 10^5

/// 2. 0 <= team[i] <= 1

/// 3. 1 <= dist <= team.length

/// </summary>

int LeetCodeGreedy::catchMaximumAmountofPeople(vector<int>& team, int dist)

{

int result = 0;

vector<queue<int>> queue(2);

for (int i = 0; i < (int)team.size(); i++)

{

while (!queue[1 - team[i]].empty() &&

i - queue[1 - team[i]].front() > dist)

{

queue[1 - team[i]].pop();

}

if (!queue[1 - team[i]].empty())

{

queue[1 - team[i]].pop();

result++;

continue;

}

else

{

queue[team[i]].push(i);

}

}

return result;

}

**1438. Longest Continuous Subarray With Absolute Diff Less Than or Equal to Limit**

Medium

Given an array of integers nums and an integer limit, return the size of the longest **non-empty** subarray such that the absolute difference between any two elements of this subarray is less than or equal to limit*.*

**Example 1:**

**Input:** nums = [8,2,4,7], limit = 4

**Output:** 2

**Explanation:** All subarrays are:

[8] with maximum absolute diff |8-8| = 0 <= 4.

[8,2] with maximum absolute diff |8-2| = 6 > 4.

[8,2,4] with maximum absolute diff |8-2| = 6 > 4.

[8,2,4,7] with maximum absolute diff |8-2| = 6 > 4.

[2] with maximum absolute diff |2-2| = 0 <= 4.

[2,4] with maximum absolute diff |2-4| = 2 <= 4.

[2,4,7] with maximum absolute diff |2-7| = 5 > 4.

[4] with maximum absolute diff |4-4| = 0 <= 4.

[4,7] with maximum absolute diff |4-7| = 3 <= 4.

[7] with maximum absolute diff |7-7| = 0 <= 4.

Therefore, the size of the longest subarray is 2.

**Example 2:**

**Input:** nums = [10,1,2,4,7,2], limit = 5

**Output:** 4

**Explanation:** The subarray [2,4,7,2] is the longest since the maximum absolute diff is |2-7| = 5 <= 5.

**Example 3:**

**Input:** nums = [4,2,2,2,4,4,2,2], limit = 0

**Output:** 3

**Constraints:**

* 1 <= nums.length <= 105
* 1 <= nums[i] <= 109
* 0 <= limit <= 109

### Analysis:

Keep maximum number in decreasing order, and minimum number in increasing order to indicating from current index to end of scanning point what is the maximum number and what is the minimum number in the sub array.

/// <summary>

/// Leet code #1438. Longest Continuous Subarray With Absolute Diff

/// Less Than or Equal to Limit

///

/// Medium

///

/// Given an array of integers nums and an integer limit, return the

/// size of the longest continuous subarray such that the absolute

/// difference between any two elements is less than or equal to limit.

///

/// In case there is no subarray satisfying the given condition return 0.

///

/// Example 1:

/// Input: nums = [8,2,4,7], limit = 4

/// Output: 2

/// Explanation: All subarrays are:

/// [8] with maximum absolute diff |8-8| = 0 <= 4.

/// [8,2] with maximum absolute diff |8-2| = 6 > 4.

/// [8,2,4] with maximum absolute diff |8-2| = 6 > 4.

/// [8,2,4,7] with maximum absolute diff |8-2| = 6 > 4.

/// [2] with maximum absolute diff |2-2| = 0 <= 4.

/// [2,4] with maximum absolute diff |2-4| = 2 <= 4.

/// [2,4,7] with maximum absolute diff |2-7| = 5 > 4.

/// [4] with maximum absolute diff |4-4| = 0 <= 4.

/// [4,7] with maximum absolute diff |4-7| = 3 <= 4.

/// [7] with maximum absolute diff |7-7| = 0 <= 4.

/// Therefore, the size of the longest subarray is 2.

///

/// Example 2:

/// Input: nums = [10,1,2,4,7,2], limit = 5

/// Output: 4

/// Explanation: The subarray [2,4,7,2] is the longest since the maximum

/// absolute diff is |2-7| = 5 <= 5.

///

/// Example 3:

/// Input: nums = [4,2,2,2,4,4,2,2], limit = 0

/// Output: 3

///

/// Constraints:

/// 1. 1 <= nums.length <= 10^5

/// 2. 1 <= nums[i] <= 10^9

/// 3. 0 <= limit <= 10^9

/// </summary>

int LeetCodeArray::longestSubarray(vector<int>& nums, int limit)

{

deque<int> min\_list;

deque<int> max\_list;

int first = 0;

int last = 0;

int result = 0;

min\_list.push\_back(nums[0]);

max\_list.push\_back(nums[0]);

while (last < (int)nums.size())

{

if (max\_list.front() - min\_list.front() <= limit)

{

result = max(result, last - first + 1);

last++;

if (last < (int)nums.size())

{

while (!min\_list.empty() && min\_list.back() > nums[last]) min\_list.pop\_back();

while (!max\_list.empty() && max\_list.back() < nums[last]) max\_list.pop\_back();

min\_list.push\_back(nums[last]);

max\_list.push\_back(nums[last]);

}

}

else

{

if (!min\_list.empty() && min\_list.front() == nums[first]) min\_list.pop\_front();

if (!max\_list.empty() && max\_list.front() == nums[first]) max\_list.pop\_front();

first++;

}

}

return result;

}

# Advanced Problems

## 239. Sliding Window Maximum

Hard

You are given an array of integers nums, there is a sliding window of size k which is moving from the very left of the array to the very right. You can only see the k numbers in the window. Each time the sliding window moves right by one position.

Return *the max sliding window*.

**Example 1:**

**Input:** nums = [1,3,-1,-3,5,3,6,7], k = 3

**Output:** [3,3,5,5,6,7]

**Explanation:**

Window position Max

--------------- -----

[1 3 -1] -3 5 3 6 7 **3**

1 [3 -1 -3] 5 3 6 7 **3**

1 3 [-1 -3 5] 3 6 7  **5**

1 3 -1 [-3 5 3] 6 7 **5**

1 3 -1 -3 [5 3 6] 7 **6**

1 3 -1 -3 5 [3 6 7] **7**

**Example 2:**

**Input:** nums = [1], k = 1

**Output:** [1]

**Example 3:**

**Input:** nums = [1,-1], k = 1

**Output:** [1,-1]

**Example 4:**

**Input:** nums = [9,11], k = 2

**Output:** [11]

**Example 5:**

**Input:** nums = [4,-2], k = 2

**Output:** [4]

**Constraints:**

1 <= nums.length <= 105

-104 <= nums[i] <= 104

1 <= k <= nums.length

### Analysis:

Keep monotone descrease stack in a slide window queue, for this purpose you need to use deque, which you can pop up from both ends. If there is no deque in the programming language, you may need to implement your double linked list.

/// <summary>

/// Leet code #239. Sliding Window Maximum

/// Given an array nums, there is a sliding window of size k which is moving

/// from the very left of the array to the very right.

/// You can only see the k numbers in the window. Each time the sliding window

/// moves right by one position.

/// For example,

/// Given nums = [1,3,-1,-3,5,3,6,7], and k = 3.

/// Window position Max

/// --------------- -----

/// [1 3 -1] -3 5 3 6 7 3

/// 1 [3 -1 -3] 5 3 6 7 3

/// 1 3 [-1 -3 5] 3 6 7 5

/// 1 3 -1 [-3 5 3] 6 7 5

/// 1 3 -1 -3 [5 3 6] 7 6

/// 1 3 -1 -3 5 [3 6 7] 7

/// Therefore, return the max sliding window as [3,3,5,5,6,7].

/// Note:

/// You may assume k is always valid, ie: 1 ¡Ü k ¡Ü input array's size for

/// non-empty array.

/// Follow up:

/// Could you solve it in linear time?

/// Hint:

/// 1.How about using a data structure such as deque (double-ended queue)?

/// 2.The queue size need not be the same as the window¡¯s size.

/// 3.Remove redundant elements and the queue should store only elements

/// that need to be considered.

/// </summary>

vector<int> LeetCodeStack::maxSlidingWindow(vector<int>& nums, int k)

{

vector<int> result;

deque<int> max\_window;

for (size\_t i = 0; i < nums.size(); i++)

{

if (max\_window.empty())

{

max\_window.push\_back(nums[i]);

}

else

{

if (max\_window.size() == k) max\_window.pop\_front();

size\_t count = 0;

while (!max\_window.empty() && max\_window.back() < nums[i])

{

max\_window.pop\_back();

count++;

}

for (size\_t j = 0; j < count; j++)

{

max\_window.push\_back(nums[i]);

}

max\_window.push\_back(nums[i]);

}

if (max\_window.size() == k)

{

result.push\_back(max\_window.front());

}

}

return result;

}

**2030. Smallest K-Length Subsequence With Occurrences of a Letter**

Hard

You are given a string s, an integer k, a letter letter, and an integer repetition.

Return *the****lexicographically smallest****subsequence of* s*of length* k *that has the letter* letter *appear****at least*** repetition *times*. The test cases are generated so that the letter appears in s **at least** repetition times.

A **subsequence** is a string that can be derived from another string by deleting some or no characters without changing the order of the remaining characters.

A string a is **lexicographically smaller** than a string b if in the first position where a and b differ, string a has a letter that appears earlier in the alphabet than the corresponding letter in b.

**Example 1:**

**Input:** s = "leet", k = 3, letter = "e", repetition = 1

**Output:** "eet"

**Explanation:** There are four subsequences of length 3 that have the letter 'e' appear at least 1 time:

- "lee" (from "**lee**t")

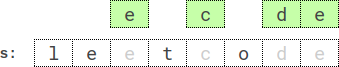
- "let" (from "**le**e**t**")

- "let" (from "**l**e**et**")

- "eet" (from "l**eet**")

The lexicographically smallest subsequence among them is "eet".

**Example 2:**



**Input:** s = "leetcode", k = 4, letter = "e", repetition = 2

**Output:** "ecde"

**Explanation:** "ecde" is the lexicographically smallest subsequence of length 4 that has the letter "e" appear at least 2 times.

**Example 3:**

**Input:** s = "bb", k = 2, letter = "b", repetition = 2

**Output:** "bb"

**Explanation:** "bb" is the only subsequence of length 2 that has the letter "b" appear at least 2 times.

**Constraints:**

* 1 <= repetition <= k <= s.length <= 5 \* 104
* s consists of lowercase English letters.
* letter is a lowercase English letter, and appears in s at least repetition times.

### Analysis:

Pop up decreasing order characters if we have more than k occurrence for that letter.

/// <summary>

/// Leet Code 2030. Smallest K-Length Subsequence With Occurrences

/// of a Letter

///

/// Hard

///

/// You are given a string s, an integer k, a letter letter, and an

/// integer repetition.

///

/// Return the lexicographically smallest subsequence of s of length

/// k that has the letter letter appear at least repetition times.

/// The test cases are generated so that the letter appears in s at

/// least repetition times.

///

/// A subsequence is a string that can be derived from another string

/// by deleting some or no characters without changing the order of

/// the remaining characters.

///

/// A string a is lexicographically smaller than a string b if in

/// the first position where a and b differ, string a has a letter

/// that appears earlier in the alphabet than the corresponding letter

/// in b.

///

/// Example 1:

/// Input: s = "leet", k = 3, letter = "e", repetition = 1

/// Output: "eet"

/// Explanation: There are four subsequences of length 3 that have

/// the letter 'e' appear at least 1 time:

/// - "lee" (from "leet")

/// - "let" (from "leet")

/// - "let" (from "leet")

/// - "eet" (from "leet")

/// The lexicographically smallest subsequence among them is "eet".

///

/// Example 2:

/// Input: s = "leetcode", k = 4, letter = "e", repetition = 2

/// Output: "ecde"

/// Explanation: "ecde" is the lexicographically smallest

/// subsequence of length 4 that has the letter "e" appear at

/// least 2 times.

///

/// Example 3:

/// Input: s = "bb", k = 2, letter = "b", repetition = 2

/// Output: "bb"

/// Explanation: "bb" is the only subsequence of length 2

/// that has the letter "b" appear at least 2 times.

///

/// Constraints:

/// 1. 1 <= repetition <= k <= s.length <= 5 \* 10^4

/// 2. s consists of lowercase English letters.

/// 3. letter is a lowercase English letter, and appears in s at

/// least repetition times.

/// </summary>

string LeetCodeStack::smallestSubsequence(string s, int k, char letter, int repetition)

{

vector<int> dp(s.size());

int count = 0;

for (int i = s.size() - 1; i >= 0; i--)

{

if (s[i] == letter)

{

count++;

}

dp[i] = count;

}

string result;

int letter\_count = 0;

for (size\_t i = 0; i < s.size(); i++)

{

if (!result.empty() && result.back() > s[i])

{

if ((result.size() + (s.size() - i) > k) &&

((letter\_count + dp[i] > repetition) ||

(letter\_count + dp[i] == repetition && result.back() != letter)))

{

if (result.back() == letter) letter\_count--;

result.pop\_back();

i--;

continue;

}

}

result.push\_back(s[i]);

if (s[i] == letter) letter\_count++;

}

string tail;

while (result.size() + tail.size() > k)

{

if (result.back() == letter)

{

if (letter\_count == repetition)

{

tail.push\_back(result.back());

}

else

{

letter\_count--;

}

}

result.pop\_back();

}

result.append(tail);

return result;

}

## 84. Largest Rectangle in Histogram

Hard

Given an array of integers heights representing the histogram's bar height where the width of each bar is 1, return *the area of the largest rectangle in the histogram*.

**Example 1:**

A picture containing text, clock

Description automatically generated

**Input:** heights = [2,1,5,6,2,3]

**Output:** 10

**Explanation:** The above is a histogram where width of each bar is 1.

The largest rectangle is shown in the red area, which has an area = 10 units.

**Example 2:**

A picture containing text, clock, sign, clipart

Description automatically generated

**Input:** heights = [2,4]

**Output:** 4

**Constraints:**

* 1 <= heights.length <= 105
* 0 <= heights[i] <= 104

### Analysis:

If a bar is higher than previous, push to stack, if the bar is lower than previous one, keep pop up all bars higher than this one and calculate area, then push the new bar. In the end pop up all bars and calculate result.

/// <summary>

/// Leet Code 84. Largest Rectangle in Histogram

///

/// Hard

///

/// Given an array of integers heights representing the histogram's bar

/// height where the width of each bar is 1, return the area of the

/// largest rectangle in the histogram.

///

/// Example 1:

/// Input: heights = [2,1,5,6,2,3]

/// Output: 10

/// Explanation: The above is a histogram where width of each bar is 1.

/// The largest rectangle is shown in the red area, which has an

/// area = 10 units.

///

/// Example 2:

/// Input: heights = [2,4]

/// Output: 4

///

/// Constraints:

/// 1. 1 <= heights.length <= 10^5

/// 2. 0 <= heights[i] <= 10^4

/// </summary>

int LeetCodeStack::largestRectangleArea(vector<int>& heights)

{

int max\_area = 0;

stack<pair<int, int>> height\_stack;

for (size\_t i = 0; i <= heights.size(); i++)

{

int height = (i == heights.size()) ? 0 : heights[i];

if (height\_stack.empty() || (height >= height\_stack.top().second))

{

height\_stack.push(make\_pair(i, height));

}

else

{

int end = height\_stack.top().first;

pair<int, int> pair;

while ((!height\_stack.empty()) &&

(height < height\_stack.top().second))

{

pair = height\_stack.top();

height\_stack.pop();

max\_area = max(max\_area, (end - pair.first + 1) \* pair.second);

}

height\_stack.push(make\_pair(pair.first, height));

height\_stack.push(make\_pair(i, height));

}

}

return max\_area;

}

## 1425. Constrained Subsequence Sum

Hard

Given an integer array nums and an integer k, return the maximum sum of a **non-empty** subsequence of that array such that for every two **consecutive** integers in the subsequence, nums[i] and nums[j], where i < j, the condition j - i <= k is satisfied.

A *subsequence* of an array is obtained by deleting some number of elements (can be zero) from the array, leaving the remaining elements in their original order.

**Example 1:**

**Input:** nums = [10,2,-10,5,20], k = 2

**Output:** 37

**Explanation:** The subsequence is [10, 2, 5, 20].

**Example 2:**

**Input:** nums = [-1,-2,-3], k = 1

**Output:** -1

**Explanation:** The subsequence must be non-empty, so we choose the largest number.

**Example 3:**

**Input:** nums = [10,-2,-10,-5,20], k = 2

**Output:** 23

**Explanation:** The subsequence is [10, -2, -5, 20].

**Constraints:**

* 1 <= k <= nums.length <= 105
* -104 <= nums[i] <= 104

### Analysis:

The target is to get maximum sub-sequence sum but keep the gap is less than k, so at each scanning position, we just care all previous answer which is within range K, if we can not get a positive one, we can start a sequence again. Keep the accumulated sum in decreasing order with end index, because if later you get a greater result later, you do not need previous one. You can also discard all negative accumulated sum if you want.

/// <summary>

/// Leet code #1425. Constrained Subset Sum

///

/// Hard

///

/// Given an integer array nums and an integer k, return the maximum sum

/// of a non-empty subset of that array such that for every two

/// consecutive integers in the subset, nums[i] and nums[j], where i < j,

/// the condition j - i <= k is satisfied.

///

/// A subset of an array is obtained by deleting some number of elements

/// (can be zero) from the array, leaving the remaining elements in their

/// original order.

/// Example 1:

///

/// Input: nums = [10,2,-10,5,20], k = 2

/// Output: 37

/// Explanation: The subset is [10, 2, 5, 20].

/// Example 2:

///

/// Input: nums = [-1,-2,-3], k = 1

/// Output: -1

/// Explanation: The subset must be non-empty, so we choose the largest

/// number.

///

/// Example 3:

///

/// Input: nums = [10,-2,-10,-5,20], k = 2

/// Output: 23

/// Explanation: The subset is [10, -2, -5, 20].

///

/// Constraints:

/// 1. 1 <= k <= nums.length <= 10^5

/// 2. -10^4 <= nums[i] <= 10^4

/// </summary>

int LeetCodeStack::constrainedSubsetSum(vector<int>& nums, int k)

{

deque<pair<int, int>> sum\_chain;

int result = INT\_MIN;

for (size\_t i = 0; i < nums.size(); i++)

{

int sum;

if (sum\_chain.empty())

{

sum = nums[i];

}

else

{

if ((int)i - sum\_chain.front().first > k) sum\_chain.pop\_front();

sum = max(nums[i], nums[i] + sum\_chain.front().second);

while (!sum\_chain.empty() && sum >= sum\_chain.back().second)

{

sum\_chain.pop\_back();

}

}

sum\_chain.push\_back(make\_pair(i, sum));

result = max(sum, result);

}

return result;

}

## 862. Shortest Subarray with Sum at Least K

Hard

Given an integer array nums and an integer k, return *the length of the shortest non-empty****subarray****of*nums*with a sum of at least*k. If there is no such **subarray**, return -1.

A **subarray** is a **contiguous** part of an array.

**Example 1:**

**Input:** nums = [1], k = 1

**Output:** 1

**Example 2:**

**Input:** nums = [1,2], k = 4

**Output:** -1

**Example 3:**

**Input:** nums = [2,-1,2], k = 3

**Output:** 3

**Constraints:**

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

### Analysis:

The target is to get maximum sub-sequence sum but keep the gap is less than k, so at each scanning position, we just care all previous answer which is within range K, if we can not get a positive one, we can start a sequence again. Keep the accumulated sum in decreasing order with end index, because if later you get a greater result later, you do not need previous one. You can also discard all negative accumulated sum if you want.

/// <summary>

/// Leet code #862. Shortest Subarray with Sum at Least K

///

/// Return the length of the shortest, non-empty, contiguous subarray of A

/// with sum at least K.

///

/// If there is no non-empty subarray with sum at least K, return -1.

///

/// Example 1:

/// Input: A = [1], K = 1

/// Output: 1

///

/// Example 2:

/// Input: A = [1,2], K = 4

/// Output: -1

///

/// Example 3:

/// Input: A = [2,-1,2], K = 3

/// Output: 3

///

/// Note:

/// 1. 1 <= A.length <= 50000

/// 2. -10 ^ 5 <= A[i] <= 10 ^ 5

/// 3. 1 <= K <= 10 ^ 9

/// </summary>

int LeetCodeStack::shortestSubarray(vector<int>& nums, int k)

{

int result = INT\_MAX;

deque<pair<long long, int>> window;

long long sum = 0;

window.push\_back(make\_pair(0, -1));

for (int i = 0; i < (int)nums.size(); i++)

{

if (nums[i] >= k) return 1;

sum += nums[i];

while (!window.empty() && sum <= window.back().first)

{

window.pop\_back();

}

while (!window.empty() && sum - window.front().first >= k)

{

result = min(result, i - window.front().second);

window.pop\_front();

}

window.push\_back(make\_pair(sum, i));

}

if (result == INT\_MAX) return -1;

return result;

}