# Stack

This topic will discuss the algorithm which leverage stack. In general stack can be used in the following scenarios:

1. Keep the sequence monotone increasing or decreasing.
2. Change the process order.
3. Pair with elements.
4. Keep track on repeated elements.
5. Keep track on nested layers

## 316. Remove Duplicate Letters

Hard

Given a string which contains only lowercase letters, remove duplicate letters so that every letter appears once and only once. You must make sure your result is the smallest in lexicographical order among all possible results.

**Example 1:**

**Input:** "bcabc"

**Output:** "abc"

**Example 2:**

**Input:** "cbacdcbc"

**Output:** "acdb"

### Analysis:

First we count the duplicate characters, then we push every character from the string, if we see the end of stack with a large character and it is duplicate, we pop it up and deduct the count. In another case if the character is already in the result string, we skip it.

/// <summary>

/// Leet code #316. Remove Duplicate Letters

///

/// Given a string which contains only lowercase letters, remove duplicate

/// letters so that every letter appear once and only once. You must make sure

/// your result is the smallest in lexicographical order among all possible

/// results.

///

/// Example:

///

/// Given "bcabc"

/// Return "abc"

///

/// Given "cbacdcbc"

/// Return "acdb"

/// </summary>

string LeetCodeStack::removeDuplicateLetters(string s)

{

// count remaining characters,

vector<int> char\_count(26);

// count in string characters

vector<int> char\_used(26);

string result;

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

{

char\_count[s[i] - 'a']++;

}

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

{

if (char\_used[s[i] - 'a'] == 0)

{

// pop out duplicate and bigger characters.

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

char\_count[result.back() - 'a'] > 0)

{

char\_used[result.back() - 'a']--;

result.pop\_back();

}

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

char\_used[s[i] - 'a']++;

}

char\_count[s[i] - 'a']--;

}

return result;

}

## 339. Nested List Weight Sum

Easy

Given a nested list of integers, return the sum of all integers in the list weighted by their depth.

Each element is either an integer, or a list -- whose elements may also be integers or other lists.

**Example 1:**

**Input:** [[1,1],2,[1,1]]

**Output:** 10

**Explanation:** Four 1's at depth 2, one 2 at depth 1.

**Example 2:**

**Input:** [1,[4,[6]]]

**Output:** 27

**Explanation:** One 1 at depth 1, one 4 at depth 2, and one 6 at depth 3; 1 + 4\*2 + 6\*3 = 27.

### Analysis:

We use the stack to keep track on nested layer, if we see '[', we push to stack, if we see ']', we pop out from stack, the stack size is depth. In this case the stack is implicitly implemented by recursive call.

/// <summary>

/// Leet code #339. Nested List Weight Sum

/// </summary>

int LeetCodeStack::depthSum(vector<NestedInteger>& nestedList, int depth)

{

int sum = 0;

for (NestedInteger ni : nestedList)

{

if (ni.isInteger())

{

sum += ni.getInteger() \* depth;

}

else

{

vector<NestedInteger> nextList = ni.getList();

sum += depthSum(nextList, depth + 1);

}

}

return sum;

}

/// <summary>

/// Leet code #339. Nested List Weight Sum

///

/// Given a nested list of integers, return the sum of all integers in the list weighted by their depth.

/// Each element is either an integer, or a list -- whose elements may also be integers or other lists.

/// Example 1:

/// Given the list [[1,1],2,[1,1]], return 10. (four 1's at depth 2, one 2 at depth 1)

/// Example 2:

/// Given the list [1,[4,[6]]], return 27. (one 1 at depth 1, one 4 at depth 2, and one 6 at depth 3; 1 + 4\*2 + 6\*3 = 27)

/// This is the interface that allows for creating nested lists.

/// You should not implement it, or speculate about its implementation

/// class NestedInteger {

/// public:

/// // Return true if this NestedInteger holds a single integer, rather than a nested list.

/// bool isInteger() const;

///

/// // Return the single integer that this NestedInteger holds, if it holds a single integer

/// // The result is undefined if this NestedInteger holds a nested list

/// int getInteger() const;

///

/// // Return the nested list that this NestedInteger holds, if it holds a nested list

/// // The result is undefined if this NestedInteger holds a single integer

/// const vector<NestedInteger> &getList() const;

/// };

/// </summary>

int LeetCodeStack::depthSum(vector<NestedInteger>& nestedList)

{

return depthSum(nestedList, 1);

}

## 402. Remove K Digits

Medium

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.

### Analysis:

We push the digits into stack and pop up the large digits in the stack, leading 0 is removed.

/// <summary>

/// <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;

}

## 439. Ternary Expression Parser

Medium

Given a string representing arbitrarily nested ternary expressions, calculate the result of the expression. You can always assume that the given expression is valid and only consists of digits 0-9, ?, :, T and F (T and F represent True and False respectively).

**Note:**

1. The length of the given string is ≤ 10000.
2. Each number will contain only one digit.
3. The conditional expressions group right-to-left (as usual in most languages).
4. The condition will always be either T or F. That is, the condition will never be a digit.
5. The result of the expression will always evaluate to either a digit 0-9, T or F.

**Example 1:**

**Input:** "T?2:3"

**Output:** "2"

**Explanation:** If true, then result is 2; otherwise result is 3.

**Example 2:**

**Input:** "F?1:T?4:5"

**Output:** "4"

**Explanation:** The conditional expressions group right-to-left. Using parenthesis, it is read/evaluated as:

"(F ? 1 : (T ? 4 : 5))" "(F ? 1 : (T ? 4 : 5))"

-> "(F ? 1 : 4)" or -> "(T ? 4 : 5)"

-> "4" -> "4"

**Example 3:**

**Input:** "T?T?F:5:3"

**Output:** "F"

**Explanation:** The conditional expressions group right-to-left. Using parenthesis, it is read/evaluated as:

"(T ? (T ? F : 5) : 3)" "(T ? (T ? F : 5) : 3)"

-> "(T ? F : 3)" or -> "(T ? F : 5)"

-> "F" -> "F"

### Analysis:

We parse the expression in nested order.

/// <summary>

/// Leet code #439. Ternary Expression Parser

/// </summary>

void LeetCodeStack::calculateTernary(string& result, string& operators)

{

while (!operators.empty() && operators.back() == ':')

{

char right = result.back();

result.pop\_back();

char left = result.back();

result.pop\_back();

char condition = result.back();

result.pop\_back();

if (condition == 'T') result.push\_back(left);

else result.push\_back(right);

operators.pop\_back();

operators.pop\_back();

}

}

/// <summary>

/// Leet code #439. Ternary Expression Parser

///

/// Given a string representing arbitrarily nested ternary expressions,

/// calculate the result of the expression. You can always assume that

/// the given expression is valid and only consists of digits 0-9, ?, :,

/// T and F (T and F represent True and False respectively).

/// Note:

/// 1.The length of the given string is ≤ 10000.

/// 2.Each number will contain only one digit.

/// 3.The conditional expressions group right-to-left (as usual in

/// most languages).

/// 4.The condition will always be either T or F. That is, the condition

/// will never be a digit.

/// 5.The result of the expression will always evaluate to either a

/// digit 0-9, T or F.

///

/// Example 1:

/// Input: "T?2:3"

/// Output: "2"

/// Explanation: If true, then result is 2; otherwise result is 3.

/// Example 2:

/// Input: "F?1:T?4:5"

/// Output: "4"

/// Explanation: The conditional expressions group right-to-left. Using

/// parenthesis, it is read/evaluated as:

/// "(F ? 1 : (T ? 4 : 5))"                   "(F ? 1 : (T ? 4 : 5))"

/// -> "(F ? 1 : 4)"                 or       -> "(T ? 4 : 5)"

/// -> "4"                                    -> "4"

/// Example 3:

/// Input: "T?T?F:5:3"

/// Output: "F"

/// Explanation: The conditional expressions group right-to-left. Using

/// parenthesis, it is read/evaluated as:

/// "(T ? (T ? F : 5) : 3)"                   "(T ? (T ? F : 5) : 3)"

/// -> "(T ? F : 3)"                 or       -> "(T ? F : 5)"

/// -> "F"                                    -> "F"

/// </summary>

string LeetCodeStack::parseTernaryII(string expression)

{

string result;

string operators;

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

{

if (isdigit(expression[i]) || expression[i] == 'T' ||

expression[i] == 'F')

{

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

}

else if (expression[i] == '?')

{

operators.push\_back(expression[i]);

}

else if (expression[i] == ':')

{

if (operators.back() == '?')

{

operators.push\_back(expression[i]);

}

else

{

calculateTernary(result, operators);

operators.push\_back(expression[i]);

}

}

}

calculateTernary(result, operators);

return result;

}

## 636. Exclusive Time of Functions

Medium

On a **single threaded** CPU, we execute some functions.  Each function has a unique id between 0 and N-1.

We store logs in timestamp order that describe when a function is entered or exited.

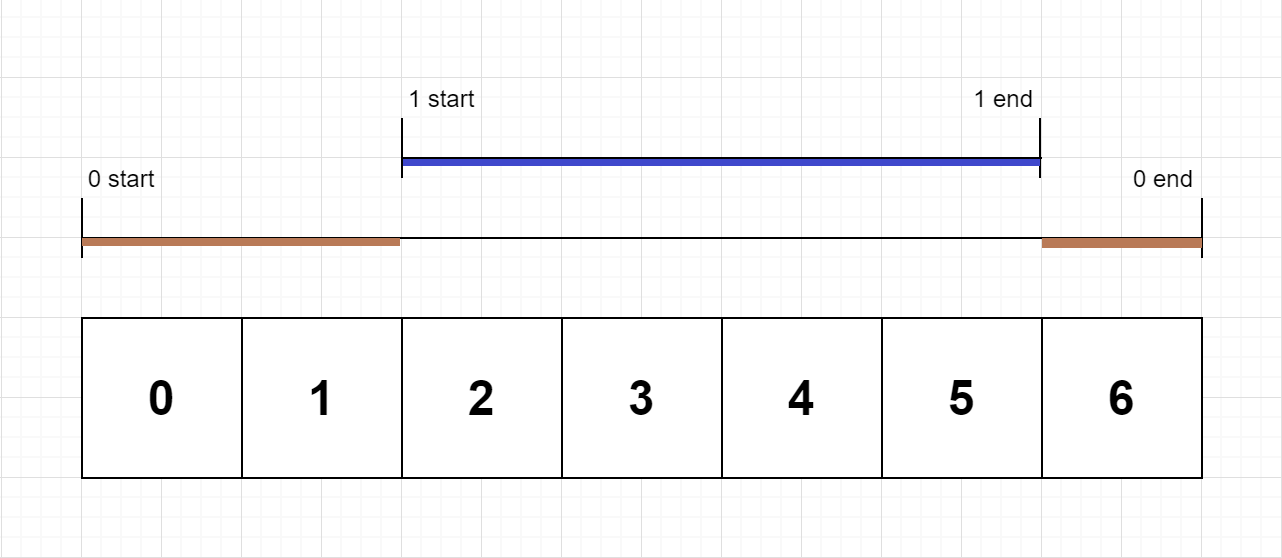
Each log is a string with this format: "{function\_id}:{"start" | "end"}:{timestamp}".  For example, "0:start:3" means the function with id 0 **started at the beginning** of timestamp 3.  "1:end:2" means the function with id 1 **ended at the end** of timestamp 2.

A function's *exclusive time* is the number of units of time spent in this function.  Note that this does **not** include any recursive calls to child functions.

The CPU is **single threaded** which means that only one function is being executed at a given time unit.

Return the exclusive time of each function, sorted by their function id.

**Example 1:**

****

**Input:**

n = 2

logs = ["0:start:0","1:start:2","1:end:5","0:end:6"]

**Output:** [3, 4]

**Explanation:**

Function 0 starts at the beginning of time 0, then it executes 2 units of time and reaches the end of time 1.

Now function 1 starts at the beginning of time 2, executes 4 units of time and ends at time 5.

Function 0 is running again at the beginning of time 6, and also ends at the end of time 6, thus executing for 1 unit of time.

So function 0 spends 2 + 1 = 3 units of total time executing, and function 1 spends 4 units of total time executing.

**Note:**

1. 1 <= n <= 100
2. Two functions won't start or end at the same time.
3. Functions will always log when they exit.

### Analysis:

This is a nested problem, any start will cause a stack push for the new thread and any end will cause a thread pop from the stack, the time period is calculated based on the last time stamp and added to the thread on the top of stack.

/// <summary>

/// Leet code #636. Exclusive Time of Functions

///

/// Given the running logs of n functions that are executed in a

/// nonpreemptive single threaded CPU, find the exclusive time of these

/// functions.

/// Each function has a unique id, start from 0 to n-1. A function may be

/// called recursively or by another function.

/// A log is a string has this format: function\_id:start\_or\_end:timestamp.

/// For example, "0:start:0" means function 0 starts from the very

/// beginning of time 0.

/// "0:end:0" means function 0 ends to the very end of time 0.

/// Exclusive time of a function is defined as the time spent within

/// this function, the time spent by calling other functions should not

/// be considered as this function's exclusive time. You should return the

/// exclusive time of each function sorted by their function id.

///

/// Example 1:

/// Input:

/// n = 2

/// logs =

/// ["0:start:0",

/// "1:start:2",

/// "1:end:5",

/// "0:end:6"]

/// Output:[3, 4]

/// Explanation:

/// Function 0 starts at time 0, then it executes 2 units of time and

/// reaches the end of time 1.

/// Now function 0 calls function 1, function 1 starts at time 2, executes

/// 4 units of time and end at time 5.

/// Function 0 is running again at time 6, and also end at the time 6, thus

/// executes 1 unit of time.

/// So function 0 totally execute 2 + 1 = 3 units of time, and function 1

/// totally execute 4 units of time.

///

/// Note:

/// 1. Input logs will be sorted by timestamp, NOT log id.

/// 2. Your output should be sorted by function id, which means the 0th

/// element of your output corresponds to the exclusive time of function 0.

/// 3. Two functions won't start or end at the same time.

/// 4. Functions could be called recursively, and will always end.

/// 5. 1 <= n <= 100

/// </summary>

vector<int> LeetCodeStack::exclusiveTime(int n, vector<string>& logs)

{

vector<int> result(n, 0);

stack<int> thread\_stack;

int last\_time = 0;

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

{

string log = logs[i];

size\_t index = 0;

int thread;

string state;

int time\_stamp;

string token;

index = log.find(':');

thread = atoi(log.substr(0, index).c\_str());

size\_t prev = index + 1;

index = log.find(':', prev);

state = log.substr(prev, index - prev);

time\_stamp = atoi(log.substr(index + 1).c\_str());

if (state == "start")

{

if (!thread\_stack.empty())

{

result[thread\_stack.top()] += time\_stamp - last\_time;

}

thread\_stack.push(thread);

last\_time = time\_stamp;

}

else if (state == "end")

{

if (!thread\_stack.empty())

{

result[thread\_stack.top()] += time\_stamp - last\_time + 1;

}

thread\_stack.pop();

last\_time = time\_stamp + 1;

}

}

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.

/// <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;

}

## 907. Sum of Subarray Minimums

Medium

Given an array of integers A, find the sum of min(B), where B ranges over every (contiguous) subarray of A.

Since the answer may be large, **return the answer modulo 10^9 + 7.**

**Example 1:**

**Input:** [3,1,2,4]

**Output:** 17

**Explanation:** Subarrays are [3], [1], [2], [4], [3,1], [1,2], [2,4], [3,1,2], [1,2,4], [3,1,2,4].

Minimums are 3, 1, 2, 4, 1, 1, 2, 1, 1, 1.  Sum is 17.

**Note:**

1. 1 <= A.length <= 30000
2. 1 <= A[i] <= 30000

### Analysis:

We iterate the array from left to right, consider the sub array end with current position, the minimum value in all the sub arrays should be the minimum value so far, so we pop up all the greater values in the stack and keep track on all positions. We should also track the existing sum until the current position to be reused as the extended range for the sub array in the future.

/// <summary>

/// Leet code #907. Sum of Subarray Minimums

///

/// Given an array of integers A, find the sum of min(B), where B ranges over

/// every (contiguous) subarray of A.

///

/// Since the answer may be large, return the answer modulo 10^9 + 7.

///

/// Example 1:

///

/// Input: [3,1,2,4]

/// Output: 17

/// Explanation: Subarrays are [3], [1], [2], [4], [3,1], [1,2], [2,4],

/// [3,1,2], [1,2,4], [3,1,2,4].

/// Minimums are 3, 1, 2, 4, 1, 1, 2, 1, 1, 1. Sum is 17.

///

/// Note:

///

/// 1. 1 <= A.length <= 30000

/// 2. 1 <= A[i] <= 30000

/// </summary>

int LeetCode::sumSubarrayMins(vector<int>& A)

{

// A array of pair, dp[i][0] is the position in A, dp[i][1] is the value of

// and dp[i][2] is the partial sum

size\_t M = 1000000007;

vector<vector<int>> dp;

dp.push\_back({ -1, 0, 0 });

int result = 0;

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

{

// kick out all the values greater than current

while (dp.back()[1] > A[i]) dp.pop\_back();

// starting from one position after previous smaller element to current

// all subarry choose current value

int sum = A[i] \* (i - dp.back()[0]);

sum %= M;

// for any subarry ending current but including previous smaller number

// and the more smaller number before previous...

sum += dp.back()[2];

sum %= M;

result += sum;

result %= M;

dp.push\_back({ i, A[i], sum });

}

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;

}

## 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:

Keep track result in stack and pop up the large and duplicate characters.

/// <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]);

}

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;

}

## 1190. Reverse Substrings Between Each Pair of Parentheses

Medium

You are given a string s that consists of lower case English letters and brackets.

Reverse the strings in each pair of matching parentheses, starting from the innermost one.

Your result should **not** contain any brackets.

**Example 1:**

**Input:** s = "(abcd)"

**Output:** "dcba"

**Example 2:**

**Input:** s = "(u(love)i)"

**Output:** "iloveu"

**Explanation:** The substring "love" is reversed first, then the whole string is reversed.

**Example 3:**

**Input:** s = "(ed(et(oc))el)"

**Output:** "leetcode"

**Explanation:** First, we reverse the substring "oc", then "etco", and finally, the whole string.

**Example 4:**

**Input:** s = "a(bcdefghijkl(mno)p)q"

**Output:** "apmnolkjihgfedcbq"

**Constraints:**

* 0 <= s.length <= 2000
* s only contains lower case English characters and parentheses.
* It's guaranteed that all parentheses are balanced.

### Analysis:

Track parentheses in stack but the parentheses itself does not to be in stack, just the string in stack, and when pop up reverse the string, before the beginning parentheses, we can add an empty string.

/// <summary>

/// Leet code #1190. Reverse Substrings Between Each Pair of Parentheses

///

/// Given a string s that consists of lower case English letters and

/// brackets.

/// Reverse the strings in each pair of matching parentheses, starting from

/// the innermost one.

/// Your result should not contain any bracket.

///

/// Example 1:

/// Input: s = "(abcd)"

/// Output: "dcba"

///

/// Example 2:

/// Input: s = "(u(love)i)"

/// Output: "iloveu"

///

/// Example 3:

/// Input: s = "(ed(et(oc))el)"

/// Output: "leetcode"

///

/// Example 4:

/// Input: s = "a(bcdefghijkl(mno)p)q"

/// Output: "apmnolkjihgfedcbq"

///

/// Constraints:

/// 1. 0 <= s.length <= 2000

/// 2. s only contains lower case English characters and parentheses.

/// 3. It's guaranteed that all parentheses are balanced.

/// </summary>

string LeetCodeStack::reverseParentheses(string s)

{

stack<string> str\_stack;

string result;

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

{

if (s[i] == '(')

{

str\_stack.push(result);

result.clear();

}

else if (s[i] == ')')

{

std::reverse(result.begin(), result.end());

if (!str\_stack.empty())

{

result = str\_stack.top() + result;

str\_stack.pop();

}

}

else

{

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

}

}

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;

}