**42. 接雨水**

给定 n 个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。

输入: [0,1,0,2,1,0,1,3,2,1,2,1]

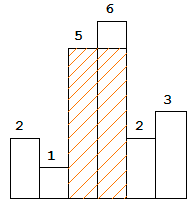
输出: 6



|  |
| --- |
| class Solution {  public:  int trap(vector<int>& height)  {  if (height.size() < 3) {  return 0;  }  int res = 0;  stack<int> s;  for (size\_t i = 0; i < height.size(); i++) {  int right = i;  while ((!s.empty()) && height[s.top()] < height[right]) {  int curt = s.top();  s.pop();  while (!s.empty() && height[s.top()] == height[curt]) {  s.pop();  }  if (!s.empty()) {  int left = s.top();  res += (min(height[left], height[right]) - height[curt]) \* (right - left - 1);  }  }  s.push(i);  }  return res;  }  };  维护递减的单调递减栈。遇到大于栈顶的，就pop，因为当前栈顶元素左右两边的高度都一定比它高，所以可以同时计算它（横向的）蓄水量。 |

**84. 柱状图中最大的矩形**

给定 n 个非负整数，用来表示柱状图中各个柱子的高度。每个柱子彼此相邻，且宽度为 1 。



求在该柱状图中，能够勾勒出来的矩形的最大面积。

输入: [2,1,5,6,2,3]

输出: 10

|  |
| --- |
| class Solution {  public:  int largestRectangleArea(vector<int>& heights)  {  int area = 0;  stack<pair<int, int>> s;  heights.push\_back(0);  for (size\_t i = 0; i < heights.size(); i++) {  if (s.empty()) {  s.push({ heights[i], i });  } else {  if (s.top().first < heights[i]) {  s.push({ heights[i], i });  } else if (s.top().first > heights[i]){  int lastPos = 0;  while (!s.empty() && s.top().first > heights[i]) {  int tmp = s.top().first \* (i - s.top().second);  area = max(area, tmp);  lastPos = s.top().second;  s.pop();  }  s.push({ heights[i], lastPos });  }  }  }  return area;  }  };  想象成锯木板，如果木板都是递增的那我很开心，如果突然遇到一块木板i矮了一截，那我就先找之前最戳出来的一块（其实就是第i-1块），计算一下这个木板单独的面积，然后把它锯成次高的，这是因为我之后的计算都再也用不着这块木板本身的高度了。再然后如果发觉次高的仍然比现在这个i木板高，那我继续单独计算这个次高木板的面积（应该是第i-1和i-2块），再把它俩锯短。直到发觉不需要锯就比第i块矮了，那我继续开开心心往右找更高的。当然为了避免到了最后一直都是递增的，所以可以在最后加一块高度为0的木板。  维护递增的单调栈，当发现当前数字比栈顶要小的时候，此时栈顶元素是最大的（大于栈里的下一个元素，也大于当前元素），所以可以计算以当前栈顶元素为高的矩形面积，并比较是不是最大的。 |

**150. 逆波兰表达式求值**

有效的运算符包括 +, -, \*, / 。每个运算对象可以是整数，也可以是另一个逆波兰表达式。

说明：

整数除法只保留整数部分。

给定逆波兰表达式总是有效的。换句话说，表达式总会得出有效数值且不存在除数为 0 的情况。

示例 1：

输入: ["2", "1", "+", "3", "\*"]

输出: 9

解释: 该算式转化为常见的中缀算术表达式为：((2 + 1) \* 3) = 9

|  |
| --- |
| class Solution {  public:  int evalRPN(vector<string>& tokens) {  stack<int> s;  set<string> ops{ "+", "-", "\*", "/" };  for (auto& str : tokens) {  if (ops.find(str) == ops.end()) {  s.push(stoi(str));  } else {  auto op2 = s.top();  s.pop();  auto op1 = s.top();  s.pop();  if (str == "+") {  s.push(op1 + op2);  }  else if (str == "-") {  s.push(op1 - op2);  }  else if (str == "\*") {  s.push(op1 \* op2);  }  else if (str == "/") {  s.push(op1 / op2);  }  }  }  return s.top();  }  }; |

**316. 去除重复字母**

给你一个仅包含小写字母的字符串，请你去除字符串中重复的字母，使得每个字母只出现一次。需保证返回结果的字典序最小（要求不能打乱其他字符的相对位置）。

示例 1:

输入: "bcabc"

输出: "abc"

示例 2:

输入: "cbacdcbc"

输出: "acdb"

|  |
| --- |
| class Solution {  public:  string removeDuplicateLetters(string s)  {  string res;  for (size\_t i = 0; i < s.size(); i++) {  char c = s[i];  if (res.find(c) != string::npos) {  continue;  }  while (!res.empty() && res.back() > c && s.find(res.back(), i) != string::npos) {  res.pop\_back();  }  res.push\_back(c);  }  return res;  }  };  维护递增的单调递增栈。栈中已存在该字符，无需要重复入栈；循环判断当前字符比栈顶更小的，并且栈顶对应的字符在后面待处理字符中还有出现，弹出栈顶的该字符，考虑之后再入栈这一字符。 |

**331. 验证二叉树的前序序列化**

序列化二叉树的一种方法是使用前序遍历。当我们遇到一个非空节点时，我们可以记录下这个节点的值。如果它是一个空节点，我们可以使用一个标记值记录，例如 #。

\_9\_

/ \

3 2

/ \ / \

4 1 # 6

/ \ / \ / \

# # # # # #

例如，上面的二叉树可以被序列化为字符串 "9,3,4,#,#,1,#,#,2,#,6,#,#"，其中 # 代表一个空节点。

给定一串以逗号分隔的序列，验证它是否是正确的二叉树的前序序列化。编写一个在不重构树的条件下的可行算法。

每个以逗号分隔的字符或为一个整数或为一个表示 null 指针的 '#' 。

你可以认为输入格式总是有效的，例如它永远不会包含两个连续的逗号，比如 "1,,3" 。

示例 1:

输入: "9,3,4,#,#,1,#,#,2,#,6,#,#"

输出: true

示例 2:

输入: "1,#"

输出: false

示例 3:

输入: "9,#,#,1"

输出: false

|  |
| --- |
| class Solution {  public:  bool isValidSerialization(string preorder) {  int n = preorder.size();  int num = 0;//记录#的个数  for(int i = n-1; i>=0;i--){  if(preorder[i] == ',')  continue;  if(preorder[i] == '#')  num++;  else{  while(i>=0 && preorder[i] != ',')//节点数字可能有多位  i--;  if(num >= 2)//#的个数>=2，消除2个#，消除一个节点数字并转换成#，即num-1  num--;  else  return false;//#的个数不足2，证明false  }  }  if(num != 1)//最终#的个数须==1  return false;  return true;  }  };  #方法二  class Solution(object):  def isValidSerialization(self, preorder):  from collections import deque  if preorder == "#":  return True  preorder = preorder.split(",")  if len(preorder) % 2 == 0 or (preorder[0] == "#" and len(preorder) > 0):  return False  stack = deque()  for val in preorder:  if val == "#":  while stack and stack[-1] == "#":  stack.pop()  if not stack:  return False  stack.pop()  stack.append(val)  return len(stack) == 1 and stack[0] == "#" |

**341. 扁平化嵌套列表迭代器**

给你一个嵌套的整型列表。请你设计一个迭代器，使其能够遍历这个整型列表中的所有整数。

列表中的每一项或者为一个整数，或者是另一个列表。其中列表的元素也可能是整数或是其他列表。

示例 1:

输入: [[1,1],2,[1,1]]

输出: [1,1,2,1,1]

解释: 通过重复调用 next 直到 hasNext 返回 false，next 返回的元素的顺序应该是: [1,1,2,1,1]。

示例 2:

输入: [1,[4,[6]]]

输出: [1,4,6]

解释: 通过重复调用 next 直到 hasNext 返回 false，next 返回的元素的顺序应该是: [1,4,6]。

|  |
| --- |
| class NestedIterator {  vector<int> datas\_;  int index\_ = 0;  public:  NestedIterator(vector<NestedInteger> &nestedList) {  Dfs(nestedList);  }  void Dfs(const vector<NestedInteger> &nestedList) {  for (auto& item : nestedList) {  if (item.isInteger()) {  datas\_.push\_back(item.getInteger());  } else {  Dfs(item.getList());  }  }  }  int next() {  return datas\_[index\_++];  }  bool hasNext() {  return index\_ < datas\_.size();  }  }; |

**385. 迷你语法分析器**

给定一个用字符串表示的整数的嵌套列表，实现一个解析它的语法分析器。

列表中的每个元素只可能是整数或整数嵌套列表

提示：你可以假定这些字符串都是格式良好的：

字符串非空

字符串不包含空格

字符串只包含数字0-9、[、-、,、]

示例 1：

给定 s = "324",

你应该返回一个 NestedInteger 对象，其中只包含整数值 324。

示例 2：

给定 s = "[123,[456,[789]]]",

返回一个 NestedInteger 对象包含一个有两个元素的嵌套列表：

1. 一个 integer 包含值 123

2. 一个包含两个元素的嵌套列表：

i. 一个 integer 包含值 456

ii. 一个包含一个元素的嵌套列表

a. 一个 integer 包含值 789

栈模拟递归。解析括号，直接使用递归解析更加简单。

|  |
| --- |
| NestedInteger deserialize(string s) {  if(s.empty()) return NestedInteger();  if(s[0]!='[') return NestedInteger(stoi(s));  if(s.size()<=2) return NestedInteger();  NestedInteger res;  int start=1,cnt=0;  for (int i=1;i<s.size();i++){  if(cnt==0&&(s[i]==','||i==s.size()-1)){  res.add(deserialize(s.substr(start,i-start)));  start=i+1;  }  else if(s[i]=='[') cnt++;  else if(s[i]==']') cnt--;  }  return res;  }  1 .首先判断s是否为空，为空直接返回；  2.s不为空的话看首字符是否为’[’，不是的话说明s为一个整数，我们直接返回结果。  如果s中首字符是’[’，且s长度小于等于2，说明没有内容，直接返回结果。（’[‘或’[]’）  如果s长度大于2，我们从i=1开始遍历，我们需要一个变量start来记录某一层的真实位置，用cnt来记录跟真实位置是否为同一深度，cnt=0表示同一深度，由于中间每段都是由逗号隔开，所以当我们判断当cnt为0，且当前字符是逗号或者已经到字符串末尾了，我们把start到当前位置之间的字符串取出来递归调用函数，把返回结果加入res中，然后start更新为i+1。如果遇到’[’，计数器cnt自增1，若遇到’]’，计数器cnt自减1。 |

**402. 移掉K位数字**

给定一个以字符串表示的非负整数 num，移除这个数中的 k 位数字，使得剩下的数字最小。

注意:

num 的长度小于 10002 且 ≥ k。

num 不会包含任何前导零。

示例 1 :

输入: num = "1432219", k = 3

输出: "1219"

解释: 移除掉三个数字 4, 3, 和 2 形成一个新的最小的数字 1219。

示例 2 :

输入: num = "10200", k = 1

输出: "200"

解释: 移掉首位的 1 剩下的数字为 200. 注意输出不能有任何前导零。

|  |
| --- |
| class Solution {  public:  string removeKdigits(string num, int k) {  vector<int> tmp;  string result;  for (int i = 0; i < num.size(); ++i) {  int current = num[i] - '0';  while (tmp.size() > 0 && tmp[tmp.size() - 1] > current && k > 0) {  tmp.pop\_back();  k--;  }  if (current != 0) {  tmp.push\_back(current);  continue;  }  if (current == 0 && tmp.size() > 0) {  tmp.push\_back(current);  continue;  }  }  while (k > 0 && tmp.size() > 0) {  tmp.pop\_back();  k--;  }  for (auto item : tmp) {  result.push\_back(item + '0');  }  if (result == "") {  return "0";  }  return result;  }  }; |

**456. 132模式**

给定一个整数序列：a1, a2, ..., an，一个132模式的子序列 ai, aj, ak 被定义为：当 i < j < k 时，ai < ak < aj。设计一个算法，当给定有 n 个数字的序列时，验证这个序列中是否含有132模式的子序列。

注意：n 的值小于15000。

示例1:

输入: [1, 2, 3, 4]

输出: False

解释: 序列中不存在132模式的子序列。

示例 2:

输入: [3, 1, 4, 2]

输出: True

解释: 序列中有 1 个132模式的子序列： [1, 4, 2].

|  |
| --- |
| class Solution {  public:  bool find132pattern(vector<int>& nums) {  int n=nums.size();  int third=INT\_MIN;  stack<int> sta;  for(int i=n-1;i>=0;i--){  if(nums[i]<third) return true;  while(!sta.empty() && nums[i]>sta.top()){  third=sta.top();  sta.pop();  }  sta.push(nums[i]);  }  return false;  }  };  维护递减的单调栈。从后往前遍历，相当于寻找231paterrn。遇到更大的数就pop出来，同时记录栈顶的值（作为第二大的数），之后判断接下来的值小于这个第二大的数即可。 |

**503. 下一个更大元素 II**

给定一个循环数组（最后一个元素的下一个元素是数组的第一个元素），输出每个元素的下一个更大元素。数字 x 的下一个更大的元素是按数组遍历顺序，这个数字之后的第一个比它更大的数，这意味着你应该循环地搜索它的下一个更大的数。如果不存在，则输出 -1。

示例 1:

输入: [1,2,1]

输出: [2,-1,2]

解释: 第一个 1 的下一个更大的数是 2；

数字 2 找不到下一个更大的数；

第二个 1 的下一个最大的数需要循环搜索，结果也是 2。

|  |
| --- |
| class Solution {  public:  vector<int> nextGreaterElements(vector<int>& nums)  {  vector<int> res(nums.size(), -1);  stack<int> s;  for (size\_t i = 0; i < 2 \* nums.size(); i++) {  int curt = nums[i % nums.size()];  while (!s.empty() && nums[s.top()] < curt) {  auto index = s.top();  s.pop();  res[index] = curt;  }  s.push(i % nums.size());  }  return res;  }  };  维护递减的单调栈。不过由于题目要求是循环的，需要两个pass，第二个pass处理循环生效的next greater，同时需要把下标已经超出范围的队首数据及时pop出来。 |

**739. 每日温度**

请根据每日气温列表，重新生成一个列表。对应位置的输出为：要想观测到更高的气温，至少需要等待的天数。如果气温在这之后都不会升高，请在该位置用 0 来代替。

例如，给定一个列表 temperatures = [73, 74, 75, 71, 69, 72, 76, 73]，你的输出应该是 [1, 1, 4, 2, 1, 1, 0, 0]。

|  |
| --- |
| class Solution {  public:  vector<int> dailyTemperatures(vector<int>& T)  {  vector<int> res(T.size(), 0);  stack<int> s;  for (size\_t i = 0; i < T.size(); i++) {  int curtTemp = T[i];  while (!s.empty() && T[s.top()] < curtTemp) {  auto topIndex = s.top();  s.pop();  res[topIndex] = i - topIndex;  }  s.push(i);  }  return res;  }  }; |

**1019. 链表中的下一个更大节点**

示例 1：

输入：[2,1,5]

输出：[5,5,0]

示例 2：

输入：[2,7,4,3,5]

输出：[7,0,5,5,0]

|  |
| --- |
| class Solution {  public:  vector<int> nextLargerNodes(ListNode\* head)  {  ListNode\* node = head;  int num = 0;  while (node != nullptr) {  num++;  node = node->next;  }  vector<int> res(num, 0);  stack<pair<int, int>> s;  int index = 0;  node = head;  while (node != nullptr) {  while (!s.empty() && s.top().first < node->val) {  int val = s.top().first;  int i = s.top().second;  s.pop();  res[i] = node->val;  }  s.push({ node->val, index++ });  node = node->next;  }  return res;  }  }; |