# 1 两数之和(简单)

题目描述: 给定一个整数数组nums和一个整数目标值target，请你在该数组中找出和为目标值的那两个整数，并返回它们的数组下标。你可以假设每种输入只会对应一个答案。但是，数组中同一个元素在答案里不能重复出现。你可以按任意顺序返回答案。

思路: 使用map, 键为数组元素的值, 值为数组元素的下标, 即mp[nums[i]] = i. 如果target-nums[i]这个数在映射里, 则返回{nums[i], target-nums[i]的下标}, 否则建立nums[i]的映射.

|  |
| --- |
| class Solution {  public:  vector<int> twoSum(vector<int>& nums, int target) {  map<int, int> mp;  for(int i = 0; i < nums.size(); i++) {  map<int, int>::iterator it = mp.find(target - nums[i]);  if(it != mp.end()) return {it->second, i};  else mp[nums[i]] = i;  }  return {};  }  }; |

也可以使用双指针法.

将nums的元素及下标记录在vector<pair<int, int>> pr里, 对pr排序.

数组A[]从小到大排序, 指针i从左开始, j从右开始.

如果A[i]+A[j]=M, 那么{i, j}保存到结果中.

如果A[i]+A[j]<M, 那么A[i-1]+A[j]<M, A[i+1]+A[j]未知, i应该往右移.

如果A[i]+A[j]>M, 那么A[i]+A[j+1]<M, A[i]+A[j-1]未知, j应该往左移.

现在来论证A[i]+A[j]<M时, 为什么只能通过i往右移来搜寻答案, 而不可以尝试同时i往左移, j往右移来查找.

假设一开始时A[i]+A[j]<M, 此时i=0, 记last=j.

因为在开端, 所以只能通过i右移来查找.

i不断右移, 突然A[i]+A[j]>M, 此时记u=i, j还是last.

现在有(A[0]~A[u-1])+A[j]<M.

现在只能通过左移j来靠拢目标M了.

(因为此时j在最右端了, 只能左移, i往如果左移, 又会使得A[i]+A[j]<M)

j不断左移, 突然A[i]+A[j]<M, 此时记v=j.

按照正确的算法, i要往右移动了. 那么为什么不可以尝试同时i往左移, j往右移来靠拢M呢?

因为一开始是因为和>M才会令j左移, 所以A[u]+(A[v+1]~A[last])都会使和>M.

所以j不能单独右移, 那么i不能单独左移也是显然的.

对于A[0]~A[u-1], 前面已经验证了(A[0]~A[u-1])+A[last]<M, 所以

(A[0]~A[u-1])+(A[v+1]~A[last])<M是必然的, 即i也不用左移, 同时j也不用右移了.

最后只剩下j往右移这种方法了.

|  |
| --- |
| class Solution {  public:  vector<int> twoSum(vector<int>& nums, int target) {  int i = 0, j = nums.size() - 1;  vector<pair<int, int>> pr;  for(int k = 0; k < nums.size(); k++) {  pr.push\_back(make\_pair(nums[k], k));  }  sort(pr.begin(), pr.end());  while(i < j) {  if(pr[i].first + pr[j].first == target)  return {pr[i].second, pr[j].second};  while(i < j && pr[i].first + pr[j].first < target)  i++;  while(i < j && pr[i].first + pr[j].first > target)  j--;  }  return {};  }  }; |

C++编程要点:

sort()对pair排序, 默认对first升序再对second升序.

快速生成pair的方法:

p = pair<string, int>("haha", 5);

p = make\_pair("haha", 5);

# 2 两数相加(中等)

题目描述:

给你两个 非空 的链表，表示两个非负的整数。它们每位数字都是按照 逆序 的方式存储的，并且每个节点只能存储 一位 数字。

请你将两个数相加，并以相同形式返回一个表示和的链表。

你可以假设除了数字 0 之外，这两个数都不会以 0 开头。

|  |
| --- |
| /\*\*  \* Definition for singly-linked list.  \* struct ListNode {  \* int val;  \* ListNode \*next;  \* ListNode() : val(0), next(nullptr) {}  \* ListNode(int x) : val(x), next(nullptr) {}  \* ListNode(int x, ListNode \*next) : val(x), next(next) {}  \* };  \*/  class Solution {  public:  ListNode\* addTwoNumbers(ListNode\* l1, ListNode\* l2) {  ListNode\* head = NULL;  ListNode\* tail = head;  int carry = 0;  while(l1 || l2) {  ListNode\* p = new ListNode;  int val1 = l1 ? l1->val : 0;  int val2 = l2 ? l2->val : 0;  int t = val1 + val2;  p->val = (t + carry) % 10;  carry = t + carry > 9 ? 1 : 0;  if(head == NULL) head = p;  if(tail) tail->next = p;  tail = p;  l1 = l1 ? l1->next : NULL;  l2 = l2 ? l2->next : NULL;  }  if(carry) {  ListNode\* p = new ListNode(1, NULL);  tail->next = p;  tail = p;  }  tail->next = NULL;  return head;  }  }; |

# 3 无重复字符的最长字串(中等)

题目描述:

给定一个字符串，请你找出其中不含有重复字符的 **最长子串**的长度。

使用滑动窗口和哈希表.

i, j从0开始, 直到j==len.

j一直向右移动, j经过的地方其哈希值变为1, 直到再次遇到某个字符.

此时可以计算最长不重复字串.

i一直向右移动,直到遇到和s[j]相同的字符, i经过的字符其哈希值变为0.

使用rslt=max(rslt, j-i+1)动态记录最长的字符串.

|  |
| --- |
| class Solution {  public:  int lengthOfLongestSubstring(string s) {  int hash[200] = {0};  int rslt = 0;  int i =0, j = 0;  if(s == "") return 0;  while(j < s.size()) {  // j右移直到遇到重复字符  while(j < s.size() && hash[s[j]] == 0) {  hash[s[j++]] = 1;  }  rslt = max(rslt, j - i);  // i右移直到遇到s[j]相同的字符  while(j < s.size() && s[i] != s[j]) {  hash[s[i++]] = 0;  }  hash[s[i++]] = 0;  }  return rslt;  }  }; |

# 5 最长回文子串(中等)

给你一个字符串 s. 找到 s 中最长的回文子串.

思路:

dp[i][j] 字符串s[i:j]是否为回文串

只有一个字符的情况: dp[i][i]=true;

只有两个字符的情况: dp[i][i+1]= (s[i]==s[i+1])

对于dp[i][j]

(1) 如果s[i]==s[j], 则dp[i][j]=dp[i+1][j-1]

(2) 如果s[i]!=s[j], 则dp[i][j]=false

按照什么样的顺序遍历?

dp数组只需要管理上三角部分. 初始时对角线为Y, 第二条对角线值为x, 其后的对角线就可以根据这两条对角线算出来.

对下面表格要进行7重大循环step,

step从1到7

i从0到7-step, 计算(i, step + i - 1)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | j=0 | j=1 | j=2 | j=3 | j=4 | j=5 | j=6 |
| i=0 | Y | X |  |  |  |  |  |
|  | i=1 | Y | X |  |  |  |  |
|  |  | i=2 | Y | X |  |  |  |
|  |  |  | i=3 | Y | X |  |  |
|  |  |  |  | i=4 | Y | X |  |
|  |  |  |  |  | i=5 | Y | X |
|  |  |  |  |  |  | i=6 | Y |

计算顺序应该为按对角线方向向右上角逼近.

|  |
| --- |
| class Solution {  public:  string longestPalindrome(string s) {  int len = s.size();  vector<vector<bool>> dp(len, vector<bool>(len));  int rsltLen = 0;  int index;  for(int step = 1; step <= len; step++) {  for(int i = 0; i <= len - step; i++) {  int j = step + i - 1;  if(i == j) dp[i][j] = true;  else if(i + 1 == j) dp[i][j] = (s[i] == s[i + 1]);  else if(s[i]==s[j]) dp[i][j] = dp[i + 1][j - 1];  else if(s[i] != s[j]) dp[i][j] = false;  if(dp[i][j] && rsltLen < j - i + 1) {  rsltLen = j - i + 1;  index = i;  }  }  }  return s.substr(index, rsltLen);  }  }; |

性能: 764ms(32.00%), 29.3MB(55.43%)

最长回文字串还有更好的方法:

二分+字符串hash, 复杂度O(nlogn)

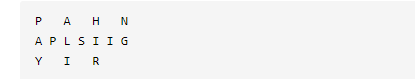
Manacher算法, 复杂度O(nlogn)

# 6. Z字形变换

题目描述:

将一个给定字符串 s 根据给定的行数 numRows ，以从上往下、从左到右进行 Z 字形排列。

比如输入字符串为 "PAYPALISHIRING" 行数为 3 时，排列如下：



# 7 整数反转(简单)

|  |
| --- |
| int reverse(int x){  long rslt = 0;  while(x && !(x % 10)) {  x /= 10;  }  while(x) {  rslt = rslt \* 10 + (x % 10);  x /= 10;  }  if((int)rslt != rslt) return 0;  return rslt;  } |

注意: 因为题目要求超过32位的有符号整数范围就返回0, 所以先将rslt定义为long型, 然后使用((int)rslt != rslt)就可以判断是否超出范围.

# 8 字符串转换整数(atoi)(中等)

|  |
| --- |
| int myAtoi(char \* s){  long rslt = 0;  char \*pStart = s;  long sign = 1;  while(\*pStart == ' ') pStart++;  if(\*pStart == '\0') return 0;  if(\*pStart != '-' && \*pStart != '+' && (\*pStart < '0' || \*pStart > '9')) {  return 0;  }  if(\*pStart == '-') sign = -1, pStart++;  else if(\*pStart == '+') pStart++;  while('0' <= \*pStart && \*pStart <= '9') {  rslt = rslt \* 10 + \*pStart - '0';  if((int)rslt != rslt) break;  pStart++;  }  rslt = sign \* rslt;  if((int)rslt != rslt) {  if(rslt > 2147483647) return 2147483647;  else return -2147483648;  }  return (int)rslt;  } |

在计算rslt的过程中就可能超出了int的范围, 但是不知道接下来还有多少个数字, 所以不如直接跳出.

# 13 罗马数字转整数(简单)

|  |
| --- |
| int romanToInt(char \* s){  int rslt = 0;  char cur, pre = 0;  while(\*s) {  cur = \*s++;  switch(cur) {  case 'I': rslt += 1; break;  case 'V': rslt += 5; if(pre == 'I') rslt -= 2; break;  case 'X': rslt += 10; if(pre == 'I') rslt -= 2; break;  case 'L': rslt += 50; if(pre == 'X') rslt -= 20; break;  case 'C': rslt += 100; if(pre == 'X') rslt -= 20; break;  case 'D': rslt += 500; if(pre == 'C') rslt -= 200; break;  case 'M': rslt += 1000; if(pre == 'C') rslt -= 200; break;  }  pre = cur;  }  return rslt;  } |

注意要减回两倍pre所代表的数字.

# 14 最长公共前缀(简单)

思路: 记最长公共前缀pref. 初始时pref为字符串[0], 然后找pref与字符串[1]的最长公共前缀p, 令pref=p, 然后依次处理字符串[2][3]...[最后], pref就是所有字符串的最长公共前缀.

|  |
| --- |
| class Solution {  public:  string helper(string str1, string str2) {  string rslt = "";  int len = min(str1.size(), str2.size());  for(int i = 0; i < len; i++) {  if(str1[i] != str2[i])  break;  rslt += str1[i];  }  return rslt;  }  string longestCommonPrefix(vector<string>& strs) {  if(strs.size() == 0) return "";  string prefix = strs[0];  for(int i = 1; i < strs.size(); i++) {  prefix = helper(prefix, strs[i]);  if(prefix == "") break;  }  return prefix;  }  }; |

# 15 三数之和(中等)

题目描述: 给你一个包含n个整数的数组nums，判断nums中是否存在三个元素a，b，c，使得a + b + c =0? 请你找出所有和为0且不重复的三元组.

题目要求不可以包含重复的三元组

思路: 借用第1题两数之和的写法, 依次对(-nums[i])在nums[i+1:last]中求两数之和, 使得nums[x]+nums[y]=-nums[i], 还要考虑到组合去重, 详细做法见代码.

|  |
| --- |
| class Solution {  public:  void towSum(vector<vector<int>> &rslt, vector<int>& ary, int start, int end, int target) {  int i = start, j = end;  while(i < j) {  if(ary[i] + ary[j] == target) {  rslt.push\_back({-target, ary[i], ary[j]});  while(i < j && ary[i] == ary[i + 1]) i++;  i++;  }  while(i < j && ary[i] + ary[j] < target) i++;  while(i < j && ary[i] + ary[j] > target) j--;  }  }  vector<vector<int>> threeSum(vector<int>& nums) {  if(nums.size() < 3) return {};  sort(nums.begin(), nums.end());  vector<vector<int>> rslt;  int target = -nums[0];  for(int i = 0; i < nums.size() - 2; i++) {  if(i != 0 && nums[i] == nums[i - 1]) continue;  target = -nums[i];  towSum(rslt, nums, i + 1, nums.size() - 1, target);  }  return rslt;  }  }; |

# 19 删除链表的倒数第N个节点(中等)

使用双指针法, fast指针先跑n步, 然后slow和fast一直跑, 直到fast跑到结尾.

|  |
| --- |
| class Solution {  public:  ListNode\* removeNthFromEnd(ListNode\* head, int n) {  ListNode\* pre;  ListNode\* slow = head;  ListNode\* fast = head;  for(int i = 0; i < n; i++) {  fast = fast->next;  }    if(fast == NULL) return head->next;  while(fast) {  pre = slow;  slow = slow->next;  fast = fast->next;  }  pre->next = slow->next;  return head;  }  }; |

# 20 有效的括号(简单)

用栈就好了.

|  |
| --- |
| class Solution {  public:  bool charMatch(char a, char b) {  switch (a) {  case '(': return b == ')'; break;  case '[': return b == ']'; break;  case '{': return b == '}'; break;  default: break;  }  return false;  }  bool isValid(string s) {  stack<char> st;  st.push(s[0]);  for(int i = 1; i < s.size(); i++) {  if(!st.empty() && charMatch(st.top(), s[i])) {  st.pop();  }  else {  st.push(s[i]);  }  }  return st.empty();  }  }; |

# 21 合并两个有序链表(简单)

将初始值大的链表插入初始值小的链表

|  |
| --- |
| /\*\*  \* Definition for singly-linked list.  \* struct ListNode {  \* int val;  \* ListNode \*next;  \* ListNode() : val(0), next(nullptr) {}  \* ListNode(int x) : val(x), next(nullptr) {}  \* ListNode(int x, ListNode \*next) : val(x), next(next) {}  \* };  \*/  class Solution {  public:  ListNode\* mergeTwoLists(ListNode\* l1, ListNode\* l2) {  ListNode \*p, \*q, \*r;  ListNode \*rslt;  if(!l1 && !l2) return NULL;  if(l1 && !l2) return l1;  if(!l1 && l2) return l2;  rslt = p = l1;  q = l1->next;  r = l2;  if(l2->val < l1->val) {  rslt = p = l2;  q = l2->next;  r = l1;  }  while(r) {  ListNode \*rNext = r->next;  if(q == NULL || r->val <= q->val) {  p->next = r;  r->next = q;  p = p->next;  r = rNext;  }  else {  p = p->next;  q = q->next;  }  }  return rslt;  }  }; |

q == NULL就相当于q->val为∞, 满足r->val <= q->val.

# 26 删除有序数组中的重复项(简单)

双指针

|  |
| --- |
| int removeDuplicates(int\* nums, int size){  if(size == 0) return 0;  int i = 0, j = 1;  while(j < size) {  if(nums[j] == nums[i]) j++;  else nums[++i] = nums[j++];  }  return i + 1;  } |

# 27 移除元素 04.19

题目描述:

给你一个数组 nums 和一个值 val，你需要 原地 移除所有数值等于 val 的元素，并返回移除后数组的新长度。

不要使用额外的数组空间，你必须仅使用 O(1) 额外空间并 原地 修改输入数组。

元素的顺序可以改变。你不需要考虑数组中超出新长度后面的元素。

|  |
| --- |
| int removeElement(int\* nums, int numsSize, int val){  int k = 0, i = 0;  while(i < numsSize) {  if(nums[i] == val) k++;  else nums[i - k] = nums[i];  i++;  }  return numsSize - k;  } |

# 28 实现strStr()(简单)

|  |
| --- |
| class Solution {  public:  int strStr(string haystack, string needle) {  // if(needle == "") return 0;  // if(haystack == "") return -1;  for(int i = 0; i + needle.size() <= haystack.size(); i++) {  int j = 0;  while(j < needle.size() && needle[j] == haystack[i + j]) j++;  if(j == needle.size()) return i;  }  return -1;  }  }; |

下面这种写法通过全部测试用例却超时了.

|  |
| --- |
| int strStr(char \* haystack, char \* needle){  if(needle[0] == '\0') return 0;  for(int i = 0; haystack[i]; i++) {  for(int j = 0; needle[j]; j++) {  if(needle[j] != haystack[i + j]) break;  if(needle[j + 1] == '\0') return i;  if(haystack[i + j] == '\0') return -1;  }  }  return -1;  } |

# 31 下一个排列(中等)

题目描述:

实现获取 下一个排列 的函数，算法需要将给定数字序列重新排列成字典序中下一个更大的排列。

如果不存在下一个更大的排列，则将数字重新排列成最小的排列（即升序排列）。

必须 原地 修改，只允许使用额外常数空间。

输入：nums = [1,2,3]

输出：[1,3,2]

输入：nums = [3,2,1]

输出：[1,2,3]

输入：nums = [1,1,5]

输出：[1,5,1]

输入：nums = [1]

输出：[1]

思路:

大的数字越靠左, 小的数字越靠右, 则num越大;

大的数字越靠右, 小的数字越靠左, 则num越小.

如果ai<aj, 且i<j则说(ai, aj)是顺序对.

如果ai>aj, 且i<j则说(ai, aj)是逆序对.

对换顺序对的位置会使排列变大;

对换逆序对的位置会使排列变小.

因此我们要对换的是顺序对, 即要满足A[i]<A[j]且i<j.

下一步: 怎么使排序变大的同时, 使得变大的幅度尽量小?

首先要对换的顺序对(A[i], A[j])越靠右, 变大的幅度会越小, 这是主要因素.

其次, 要对换的顺序对越接近, 变大的幅度会越小, 即A[j]>A[i]但是A[j]又要尽量小.

第三, 顺序对的位置交换以后, 对于i位置以后的数, 要保持降序排列.

步骤:

(1)从右往左找, 找到第一个相邻的顺序对(A[i], A[x]).

那么肯定有A[i]<A[x]>A[x+1]~A[last]

(2) 从last往左开始找, 找打第一个大于A[i]的数A[j], swap(A[i], A[j]).

(3) 要保持A[i+1]到A[last]降序.

|  |
| --- |
| class Solution {  public:  void nextPermutation(vector<int>& nums) {  int len = nums.size();  int i = len - 2, j = len - 1;  while(i >= 0 && j >= 1 && nums[i] >= nums[j]) i--, j--;  if(i >= 0) {  for(j = len - 1; nums[j] <= nums[i]; j--);  nums[i] ^= nums[j];  nums[j] ^= nums[i];  nums[i] ^= nums[j];  }  sort(nums.begin() + i + 1, nums.end());  }  }; |

# 36 有效的数独(中等)

题目描述:

判断一个 9x9 的数独是否有效。只需要根据以下规则，验证已经填入的数字是否有效即可。

数字 1-9 在每一行只能出现一次。

数字 1-9 在每一列只能出现一次。

数字 1-9 在每一个以粗实线分隔的 3x3 宫内只能出现一次。

思路:

row[x][y] = 1, 第x行数字y出现了.

col[x][y] = 1, 第x列数字y出现了.

bank[x][y] = 1, 第x块数字y出现了.

遍历到元素A[i][j], 令t= A[i][j], 块号b=i/3\*3+j/3.

row[i][t]++;

col[j][t]++;

bank[b][t]++;

if(row[i][t] == 2 || col[j][t] == 2 || bank[b][t]) return fasle;

|  |
| --- |
| bool isValidSudoku(char\*\* board, int boardSize, int\* boardColSize){  int row[9][9] = {0};  int col[9][9] = {0};  int bank[9][9] = {0};  for(int i = 0; i < 9; i++) {  for(int j = 0; j < 9; j++) {  if(board[i][j] == '.')  continue;  int t = board[i][j], b = i / 3 \* 3 + j / 3;  t = t - '0' - 1;  row[i][t]++;  col[j][t]++;  bank[b][t]++;  if(row[i][t] == 2 || col[j][t] == 2 || bank[b][t] == 2)  return false;  }  }  return true;  } |

# 38 外观数列(简单)

注意理解题意.

|  |
| --- |
| class Solution {  public:  string countAndSay(int n) {  string str = "1";  string rslt = "";  for(int i = 1; i < n; i++) {  for(int j = 0; j < str.size(); ) {  int cnt = 0;  char cur = str[j];  while(j < str.size() && cur == str[j]) {  j++;  cnt++;  }  rslt += to\_string(cnt) + cur;  }  str = rslt;  rslt = "";  }  return str;  }  }; |

# 44 通配符匹配(困难)

使用双指针, i指向普通字符串, j指向模式字符串.

s[i]和p[j]的所有组合如下

p[j] == '?', j移动直到p[j]!='?', i移动j移过的步数.

p[j] == '\*', j移动直到p[j]!='\*', 如果p[j]=='?', continue, 否则, i移动直到s[i]==p[j]

s[i]==p[j]? s[i] == '\0'?

dp[i][j]表示s的前i个字符组成的字符串与p的前j个字符组成的字符串能否匹配.

0≤i≤s.size, 0≤j≤p.size

对于dp[i][j], p[j-1]有3种情况:

p[j-1] == 'x':

如果i==0, 则dp[i][j]=false

否则如果s[i-1] == p[j-1], 当前能够匹配取决于前面的串是否匹配

则dp[i][j]=dp[i-1][j-1]

p[j-1] == '?':

'?'可以和任意一个字符匹配, 当然不能和空字符匹配.

若i==0, 则dp[i][j]=false;

否则当前能够匹配取决于前面的串是否匹配

则dp[i][j]=dp[i-1][j-1]

p[j-1] == '\*':

分两种情况讨论

(1) '\*'与s[i-1]匹配, 保留'\*', i往前移动.

当前是否能匹配可表示成dp[i-1][j]

(2) '\*'不与s[i-1]匹配

当前是否能匹配可表示成dp[i][j-1]

所以dp[i][j]= dp[i-1][j]||dp[i][j-1].

初始情况:

空串""和空串""匹配, dp[0][0]=true;

任意非空串和空串""不匹配dp[i][0]=false; (i!=0)

|  |
| --- |
| class Solution {  public:  bool isMatch(string s, string p) {    vector<vector<bool>> dp(s.size() + 1, vector<bool>(p.size() + 1, false));  dp[0][0] = true;  for(int i = 1; i < s.size(); i++) {  dp[i][0] = false;  }    for(int i = 0; i <= s.size(); i++) {  for(int j = 1; j <= p.size(); j++) {  if(p[j - 1] == '?') {  if(i == 0)  dp[i][j] = false;  else  dp[i][j] = dp[i - 1][j - 1];  }  else if(p[j - 1] == '\*') {  dp[i][j] = (i > 0 && dp[i - 1][j]) || dp[i][j - 1];  }  else {  if(i > 0 && s[i - 1] == p[j - 1])  dp[i][j] = dp[i - 1][j - 1];  }  }  }  return dp[s.size()][p.size()];  }  }; |

# 48 旋转图像(中等)

给定一个 n×n 的二维矩阵matrix 表示一个图像. 请你将图像顺时针旋转 90 度.

你必须在 原地 旋转图像, 这意味着你需要直接修改输入的二维矩阵. 请不要使用另一个矩阵来旋转图像。

先上下对称交换, 再左下角, 右上角交换.

|  |
| --- |
| class Solution {  public:  void rotate(vector<vector<int>>& matrix) {  vector<int> tmp;  int row = matrix.size();  for(int i = 0; i < row / 2; i++) {  tmp = matrix[i];  matrix[i] = matrix[row - i - 1];  matrix[row - i - 1] = tmp;  }  for(int i = 1; i < row; i++) {  for(int j = 0; j < i; j++) {  int t = matrix[i][j];  matrix[i][j] = matrix[j][i];  matrix[j][i] = t;  }  }  }  }; |

# 51 N皇后(困难) 04.18

N个皇后放置在N×N的棋盘上, 输出每一种放置方法.

皇后两两不能在同一条直线上, 不能在同一条对角线上.

本质上是求1~N的全排列, 但是加了一些约束.

使用递归,

求1开头的全排列就是要求2~N的全排列,

求2~N的全排列就是要求3~N的全排列

以此类推

使用数组isUsed[x]表示数字x是否已经使用过了.

# 53 最大子序和(简单)

题目描述: 给定一个整数数组 nums, 找到一个具有最大和的连续子数组(子数组最少包含一个元素), 返回其最大和.

动态规划: dp[i]表示遍历到数组下标i时最大的连续子序和. 当到达下标i时，有两种情况:

(1) 将a[i]加入前面的连续子序列, 形成长度增长1的子序列, 得到一个子序列和=旧子序和+a[i].

(2) a[i]自己从新开始新的子序列, 新的子序和=a[i].

那么dp[i]=max(旧子序和+a[i], a[i]), 最后还要挑选出最大的dp[i].

|  |
| --- |
| int maxSubArray(int\* nums, int len){  int \*dp;  dp = (int \*)malloc(sizeof(int) \* len);  dp[0] = nums[0];  for(int i = 1; i < len; i++) {  dp[i] = fmax(nums[i], dp[i - 1] + nums[i]);  }  int rslt = dp[0];  for(int i = 0; i < len; i++) {  rslt = fmax(rslt, dp[i]);  }  return rslt;  } |

简化: 其实没必要保留dp[]数组, 要的只是最大的数, 只需要将第二个循环合并到第一个循环, 动态保留最大的值即可.

|  |
| --- |
| int maxSubArray(int\* nums, int len){  int sum = nums[0];  int rslt = nums[0];  for(int i = 1; i < len; i++) {  sum = fmax(nums[i], sum + nums[i]);  rslt = fmax(rslt, sum);  }  return rslt;  } |

# 62 不同路径（中等）

题目描述：

一个机器人位于一个 m x n 网格的左上角 （起始点在下图中标记为 “Start” ）。

机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为 “Finish” ）。

问总共有多少条不同的路径？

使用动态规划:

dp[i][j]表示从坐标(0,0)达到坐标(i,j)的不同路径的数量.

一般情况下, 坐标(i,j)可以从上方到达, 也可以从左方到达, 即  
dp[i][j]=dp[i-1][j]+dp[i][j-1],

考虑到坐标的合法性, 当在左边界时  
dp[i][0]=dp[i-1][0],

当在上边界时  
dp[o][j]=dp[0][j-1].

从坐标(0,0)到达坐标(0,0)只有一条路径,所以dp[0][0]=1.

|  |
| --- |
| class Solution {  public:  int uniquePaths(int m, int n) {  vector<vector<int>> dp(m, vector<int>(n));  dp[0][0] = 1;  for(int i = 1; i < m; i++)  dp[i][0] = dp[i - 1][0];  for(int j = 1; j < n; j++)  dp[0][j] = dp[0][j - 1];  for(int i = 1; i < m; i++) {  for(int j = 1; j < n; j++) {  dp[i][j] = dp[i - 1][j] + dp[i][j - 1];  }  }  return dp[m - 1][n - 1];  }  }; |

也可以采用数学方法排列组合.

按照题目给定的走法, 从坐标(0,0)到坐标(x,y)必须要走(x+y)步, 并且必须向右走y步, 向下走x步, 题目等价于求, 本题是从坐标(0,0)到坐标(m-1,n-1).

|  |
| --- |
| class Solution {  public:  int uniquePaths(int m, int n) {  m = m - 1;  n = n - 1;  long long rslt = 1;  for(int y = 1, x = m + 1; y <= n; x++, y++) {  rslt = rslt \* x / y;  }  return rslt;  }  }; |

注意, 一定要写成rslt = rslt \* x / y; 而不是rslt \*= x / y;

\*=的优先级比/低, 导致先运算除法再运算乘法, 再计算的过程中误差累积导致最后的结果不正确.

# 66 加一

|  |
| --- |
| class Solution {  public:  vector<int> plusOne(vector<int>& digits) {  int i = digits.size() - 1;  vector<int> rslt = digits;  int t;  do{  t = digits[i] + 1;  rslt[i--] = t > 9 ? 0 : t;  }while(i >= 0 && t > 9);  if(t > 9)  rslt.insert(rslt.begin(), 1);  return rslt;  }  }; |

# 70 爬楼梯(简单)

使用动态规划:

dp[i]表示爬到第i阶楼梯的方法数, 最后一步要么跨1步, 要么跨2步, 所以

dp[i] = dp[i–1] + dp[i-2].

显然dp[1]=1, dp[2]=2.

因为每次都只用到前两个结果, 所以可以压缩存储空间.

|  |
| --- |
| class Solution {  public:  int climbStairs(int n) {  if(n < 3) return n;  int a = 1, b = 2, c;  for(int i = 3; i <= n; i++) {  c = a + b;  a = b;  b = c;  }  return c;  }  }; |

# 88 合并两个有序数组(简单)

题目描述：

给你两个有序整数数组 nums1 和 nums2，请你将 nums2 合并到 nums1 中，使 nums1 成为一个有序数组。

初始化 nums1 和 nums2 的元素数量分别为 m 和 n 。你可以假设 nums1 的空间大小等于 m + n，这样它就有足够的空间保存来自 nums2 的元素。

由于nums1空间大小等于m+n, 所以可以进行原地合并.

使用指针i和j, 从nums1有效数字和nums2的尾部开始遍历, 指针k指向nums1的尾部, 用来表示下一个数据存放的位置.

|  |
| --- |
| class Solution {  public:  void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {  int i = m - 1, j = n - 1, k = m + n - 1;  while(i >= 0 && j >= 0) {  if(nums1[i] > nums2[j]) nums1[k--] = nums1[i--];  else nums1[k--] = nums2[j--];  }  while(i >= 0) nums1[k--] = nums1[i--];  while(j >= 0) nums1[k--] = nums2[j--];  }  }; |

# 91 解码方法(中等)

一条包含字母 A-Z 的消息通过以下映射进行了 编码 ：

'A' -> 1

'B' -> 2

...

'Z' -> 26

要 解码 已编码的消息，所有数字必须基于上述映射的方法，反向映射回字母（可能有多种方法）。例如，"11106" 可以映射为：

"AAJF" ，将消息分组为 (1 1 10 6)

"KJF" ，将消息分组为 (11 10 6)

注意，消息不能分组为  (1 11 06) ，因为 "06" 不能映射为 "F" ，这是由于 "6" 和 "06" 在映射中并不等价。

给你一个只含数字的 非空 字符串 s ，请计算并返回 解码 方法的 总数 。

题目数据保证答案肯定是一个 32 位 的整数。

思路:

前dp[i]前i个字符的解码方法.

前i个字符的最后一个字符是s[i-1]

前i个字符的倒数第二个字符是s[i-2]

边界:

if(s[0] == '0') return false;

dp[0]=1

dp[1]=1

dp[i-2], 令x=s[i-2], y=s[i-1].

(1) x=='0' && y=='0', return 0;

(2) x=='0' && y!='0', dp[i]=dp[i-1]

(3) x!='0' && y=='0' && "xy"≤"26", dp[i] = dp[i-2]

(4) x!='0' && y=='0' && "xy">"26", return 0;

(5) x!='0' && y!='0' && "xy"≤"26", dp[i]=dp[i-1]+dp[i-2]

(6) x!='0' && y!='0' && "xy">"26", dp[i]=dp[i-1]

|  |
| --- |
| class Solution {  public:  int numDecodings(string s) {  int len = s.size();  vector<int> dp(len + 1, 0);  if(s[0] == '0')  return 0;  dp[0] = 1;  dp[1] = 1;  for(int i = 2; i <= len; i++) {  char x = s[i - 2], y = s[i - 1];  string xy = s.substr(i - 2, 2);  if(x == '0' && y == '0') return 0;  else if(x == '0' && y != '0')  dp[i] = dp[i - 1];  else if(x != '0' && y == '0' && xy <= "26")  dp[i] = dp[i - 2];  else if(x != '0' && y == '0' && xy > "26")  return 0;  else if(x != '0' && y != '0' && xy <= "26")  dp[i] = dp[i - 1] + dp[i - 2];  else if(x != '0' && y != '0' && xy > "26")  dp[i] = dp[i - 1];  }  return dp[len];  }  }; |

压缩空间版本

|  |
| --- |
| class Solution {  public:  int numDecodings(string s) {  if(s[0] == '0')  return 0;  int a = 1, b = 1, c = 1;  for(int i = 2; i <= s.size(); i++) {  char x = s[i - 2], y = s[i - 1];  string xy = s.substr(i - 2, 2);  if(x == '0' && y == '0') return 0;  else if(x == '0' && y != '0')  c = b;  else if(x != '0' && y == '0' && xy <= "26")  c = a;  else if(x != '0' && y == '0' && xy > "26")  return 0;  else if(x != '0' && y != '0' && xy <= "26")  c = a + b;  else if(x != '0' && y != '0' && xy > "26")  c = b;  a = b;  b = c;  }  return c;  }  }; |

# 101 对称二叉树(简单)

题目描述:

给定一个二叉树，检查它是否是镜像对称的.

对每颗子树, 设置两对指针

第一队指针, 一个一直往左走, 一个一直往右走, 所走之处结点都相等

第二队指针, 一个一直往右走, 一个一直往左走, 所走之处结点都相等

如果上面两条都为真, 则树是镜像对称的.

|  |
| --- |
| /\*\*  \* Definition for a binary tree node.  \* struct TreeNode {  \* int val;  \* TreeNode \*left;  \* TreeNode \*right;  \* TreeNode() : val(0), left(nullptr), right(nullptr) {}  \* TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}  \* TreeNode(int x, TreeNode \*left, TreeNode \*right) : val(x), left(left), right(right) {}  \* };  \*/  class Solution {  public:  bool helper(TreeNode\* p, TreeNode\* q) {  if(!p && !q) return true;  if(!p && q) return false;  if(p && !q) return false;  if(p->val != q->val) return false;  return helper(p->left, q->right) && helper(p->right, q->left);  }  bool isSymmetric(TreeNode\* root) {  return helper(root->left, root->right);  }  }; |

性能: 0ms, 15.9MB

# 102 二叉树的层序遍历(中等)

直接上

|  |
| --- |
| /\*\*  \* Definition for a binary tree node.  \* struct TreeNode {  \* int val;  \* TreeNode \*left;  \* TreeNode \*right;  \* TreeNode() : val(0), left(nullptr), right(nullptr) {}  \* TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}  \* TreeNode(int x, TreeNode \*left, TreeNode \*right) : val(x), left(left), right(right) {}  \* };  \*/  class Solution {  public:  vector<vector<int>> levelOrder(TreeNode\* root) {  if(!root) return {};  vector<vector<int>> rslt;  queue<TreeNode\*> qe;  int cnt = 1; // 当前层的节点数  int lev = 0; // 层号  qe.push(root);  while(!qe.empty()) {  int nextCnt = 0;  rslt.push\_back({});  while(cnt) {  TreeNode\* tNode = qe.front();  qe.pop();  rslt[lev].push\_back(tNode->val);  if(tNode->left) qe.push(tNode->left), nextCnt++;  if(tNode->right) qe.push(tNode->right), nextCnt++;  cnt--;  }  lev++;  cnt = nextCnt;  }  return rslt;  }  }; |

性能: 0ms(100%), 12.1MB(63.15%)

# 104 二叉树的最大深度(简单)

直接递归

|  |
| --- |
| /\*\*  \* Definition for a binary tree node.  \* struct TreeNode {  \* int val;  \* TreeNode \*left;  \* TreeNode \*right;  \* TreeNode() : val(0), left(nullptr), right(nullptr) {}  \* TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}  \* TreeNode(int x, TreeNode \*left, TreeNode \*right) : val(x), left(left), right(right) {}  \* };  \*/  class Solution {  public:  int maxDepth(TreeNode\* root) {  if(!root) return 0;  if(root->left == NULL && root->right == NULL)  return 1;  return max(1 + maxDepth(root->left), 1 + maxDepth(root->right));  }  }; |

# 108 将有序数组转换为二叉搜索树(简单)

二叉查找树（Binary Search Tree），（又：二叉搜索树，二叉排序树）它或者是一棵空树，或者是具有下列性质的二叉树： 若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值； 若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值； 它的左、右子树也分别为二叉排序树。

题目描述:

给你一个整数数组 nums ，其中元素已经按 升序 排列，请你将其转换为一棵 高度平衡 二叉搜索树。

高度平衡 二叉树是一棵满足「每个节点的左右两个子树的高度差的绝对值不超过 1 」的二叉树。

思路:

由于nums已经有序, 为了满足二叉排序树的性质, 以nums[mid]作为根节点,

左指针指向[0, mid-1]的根, 右指针指向[mid+1, right]的根.

|  |
| --- |
| class Solution {  public:  TreeNode\* helper(vector<int>& nums, int left, int right) {  if(left > right) return NULL;  TreeNode\* root = new TreeNode;  int mid = (right - left) / 2 + left;  root->val = nums[mid];  root->left = helper(nums, left, mid - 1);  root->right = helper(nums, mid + 1, right);  return root;  }  TreeNode\* sortedArrayToBST(vector<int>& nums) {  int right = nums.size() - 1;  return helper(nums, 0, right);  }  }; |

性能: 24ms(33.30%), 20.9MB(63.17%)

# 121 买卖股票的最佳时机(简单)

在历史最低点买入, 在历史最高点卖出, 获得的利益最高. 约束, 一定要最低点在前, 最高点在后.

|  |
| --- |
| int maxProfit(int\* prices, int pricesSize){  int minPrice = prices[0];  int rslt;  for(int i = 0; i < pricesSize; i++) {  minPrice =fmin(minPrice, prices[i]);  rslt = fmax(prices[i] - minPrice, rslt);  }  return rslt;  } |

# 122 买卖股票的最佳时机II(简单)

方法一 本质上是求增量的累积.

|  |
| --- |
| int maxProfit(int\* prices, int pricesSize){  int minPrice = prices[0];  int sum = 0;  for(int i = 1; i < pricesSize; i++) {  int t = prices[i] - prices[i - 1];  if(t > 0) sum += t;  }  return sum;  } |

方法二: 使用动态规划.

dp[0][i] 表示在第i天末尾不持有股票的最大收益.

dp[1][i] 表示在第i天末尾持有股票的最大收益.

(1)dp[0][i]. 第i天末尾不持有股票, 可能是:

a. 第i天把股票卖了, 这天的股票价格是p[i], 也就是第i-1天是持有股票的, 则

dp[0][i]=dp[1][i-1]+p[i].

b. 第i天本来就没有股票, 即第i天即不赚也不亏, 最大收益和前一天一样, 则

dp[0][i]=dp[0][i-1].

(2) dp[1][i]. 第i天末尾持有股票, 可能是:

a. 当天买入的, 则dp[1][i]=dp[0][i-1]-p[i].

b. 前一天就有股票, dp[1][i]=dp[1][i-1]

dp[0][i]=max(dp[1][i-1]+p[i], dp[0][i-1]);

dp[1][i]=max(dp[0][i-1]-p[i], dp[1][i-1]);

例子[7,1,5,3,6,4]

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| i | 0 | 1 | 2 | 3 | 4 | 5 |
| p[i] | 7 | 1 | 5 | 3 | 6 | 4 |
| dp[0][i] | 0 | -6,0 0 | 4,0 4 | 2,4 4 | 7,4 7 | 5,7 7 |
| dp[1][i] | -7 | -1,-7, -1 | -5,-1 -1 | 1,-1 1 | -2,1 1 | 3,1 3 |

最大值是7

|  |
| --- |
| class Solution {  public:  int maxProfit(vector<int>& prices) {  vector<vector<int>> dp(2, vector<int>(prices.size()));  dp[0][0] = 0;  dp[1][0] = -prices[0];  for(int i = 1; i < prices.size(); i++) {  dp[0][i] = max(dp[1][i - 1] + prices[i], dp[0][i - 1]);  dp[1][i] = max(dp[0][i - 1] - prices[i], dp[1][i - 1]);  }  return dp[0][prices.size() - 1];  }  }; |

# 129 求根节点到叶节点数字之和（中等）

题目描述:

给你一个二叉树的根节点 root ，树中每个节点都存放有一个 0 到 9 之间的数字。

每条从根节点到叶节点的路径都代表一个数字：

例如，从根节点到叶节点的路径 1 -> 2 -> 3 表示数字 123 。

计算从根节点到叶节点生成的 所有数字之和 。

叶节点 是指没有子节点的节点。

采用先序遍历, 求出每一条路径上的数值, 然后求和.

|  |
| --- |
| 先序遍某结点  处理该结点  先序遍历左子树  先序遍历右子树 |

|  |
| --- |
| int sum, num;  void helper(struct TreeNode\* root) {  num = num \* 10 + root->val;  if(root->left == NULL && root->right == NULL) {  sum += num;  }  if(root->left) {  helper(root->left);  num /= 10;  }  if(root->right) {  helper(root->right);  num /= 10;  }  }  int sumNumbers(struct TreeNode\* root) {  sum = 0, num = 0;  helper(root);  return sum;  } |

# 136 只出现一次的数字(简单)

题目描述:

给定一个非空整数数组，除了某个元素只出现一次以外，其余每个元素均出现两次。找出那个只出现了一次的元素。

说明：你的算法应该具有线性时间复杂度。 你可以不使用额外空间来实现吗？

使用异或运算

a^b=b^a;

a^a^c = a^c^a = c^a^a = c;

0^a=a;

|  |
| --- |
| int singleNumber(int\* nums, int numsSize){  int rslt = 0;  for(int i = 0; i < numsSize; i++) {  rslt ^= nums[i];  }  return rslt;  } |

# 141 环形链表(简单)

使用快指针和慢指针.

|  |
| --- |
| bool hasCycle(struct ListNode \*head) {  struct ListNode \*slow = head, \*fast = head;  while(fast && fast->next) {  slow = slow->next;  fast = fast->next->next;  if(slow == fast)  return true;  }  return false;  } |

# 151 翻转字符串里的单词(中等)

用栈

|  |
| --- |
| class Solution {  public:  string reverseWords(string s) {  stack<string> st;  int i = 0;  string rslt;  while(i < s.size()) {  while(i < s.size() && s[i] == ' ') i++;    while(i < s.size() && s[i] != ' ') {  rslt += s[i++];  }  if(!rslt.empty()) st.push(rslt);  rslt.clear();  }  while(!st.empty()) {  rslt += st.top();  st.pop();  rslt += ' ';  }  rslt.erase(rslt.size() - 1, 1);  return rslt;  }  }; |

# 155 最小栈(简单)

题目描述:

设计一个支持 push ，pop ，top 操作，并能在常数时间内检索到最小元素的栈。

push(x) —— 将元素 x 推入栈中。

pop() —— 删除栈顶的元素。

top() —— 获取栈顶元素。

getMin() —— 检索栈中的最小元素。

|  |
| --- |
| class MinStack {  public:  struct Num {  int data;  int minData;  };  stack<Num> st;  /\*\* initialize your data structure here. \*/  MinStack() {  }    void push(int val) {  if(st.empty()) {  st.push({val, val});  return;  }  int t = st.top().minData;  int minData = val < t ? val : t;  st.push({val, minData});  }    void pop() {  st.pop();  }    int top() {  return st.top().data;  }    int getMin() {  return st.top().minData;  }  };  /\*\*  \* Your MinStack object will be instantiated and called as such:  \* MinStack\* obj = new MinStack();  \* obj->push(val);  \* obj->pop();  \* int param\_3 = obj->top();  \* int param\_4 = obj->getMin();  \*/ |

# 179 最大数(中等) 04.13

题目描述:

给定一组非负整数 nums，重新排列每个数的顺序（每个数不可拆分）使之组成一个最大的整数。

注意：输出结果可能非常大，所以你需要返回一个字符串而不是整数。

对于两个正整数x和y, 只有两种拼接方法{x,y}和{y,x}

由于{(to\_str(x), to\_str(y)}和{(to\_str(y), to\_str(x)}长度一样, 所以字典序大的所对应的整数也大.

|  |
| --- |
| class Solution {  public:  string largestNumber(vector<int>& nums) {  sort(nums.begin(), nums.end(), [] (int x, int y) -> bool {  string a = to\_string(x), b = to\_string(y);  return a + b > b + a;  });  string rslt = "";  for(int i = 0; i < nums.size(); i++) {  rslt += to\_string(nums[i]);  }  return rslt[0] == '0' ? "0" : rslt;  }  }; |

lambda表达式:

[函数对象参数] (操作符重载函数参数) -> 返回类型 {函数体}

# 190 颠倒二进制位(简单)

颠倒给定的 32 位无符号整数的二进制位。

|  |
| --- |
| class Solution {  public:  uint32\_t reverseBits(uint32\_t n) {  uint32\_t x = 0x80000000;  uint32\_t y = 0x00000001;  while(x > y) {  if((bool)(x & n) != (bool)(y & n)) {  n ^= x;  n ^= y;  }  x >>= 1;  y <<= 1;  }  return n;  }  }; |

# 198 打家劫舍(中等)

题目描述:

你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。

给定一个代表每个房屋存放金额的非负整数数组，计算你 不触动警报装置的情况下 ，一夜之内能够偷窃到的最高金额。

dp[0][i] 不偷第i家的最大收益

dp[1][i] 偷第i家的最大收益

p[i]第i家的金钱.

对于第i家

(1) 不偷, 上一家可偷可不偷. dp[0][i] = max{dp[0][i - 1], dp[1][i - 1]}.

(2) 偷, 上一家只能不偷. dp[1][i] = dp[0][i - 1] + p[i]

最后返回max{dp[0][size-1], dp[1][size – 1]}

边界:

dp[0][0] = 0;

dp[1][0] = p[0];

|  |
| --- |
| class Solution {  public:  int rob(vector<int>& nums) {  int size = nums.size();  vector<vector<int>> dp(2, vector<int>(size));  dp[0][0] = 0;  dp[1][0] = nums[0];  for(int i = 1; i < size; i++) {  dp[0][i] = max(dp[0][i - 1], dp[1][i - 1]);  dp[1][i] = dp[0][i - 1] + nums[i];  }  return max(dp[0][size - 1], dp[1][size - 1]);  }  }; |

压缩空间

|  |
| --- |
| class Solution {  public:  int rob(vector<int>& nums) {  int a = 0;  int b = nums[0];  for(int i = 1; i < nums.size(); i++) {  int t = a;  a = max(a, b);  b = t + nums[i];  }  return max(a, b);  }  }; |

注意, 在每个循环开始时, 要先用一个临时变量保存a的值.

# 204 计数质数(简单)

统计所有小于非负整数n的质数的数量.

思路:

一个合数一定可以分解为若干个质数因子.

因此一个质数的整数倍一定是合数.

可以采用筛选法, 一个个划掉质数的倍数, 剩下的就是合数了.

注意, 当处理到质数x时, 不需要考虑小于x²的数, 因为这些数一定被小于x的质数干掉了.

因为要验证n是否为质数, 只需要验证n能否被[2, sqrt(n)]整除.

|  |
| --- |
| class Solution {  public:  int countPrimes(int n) {  vector<bool> isPrime(n + 1, true);  for(int i = 2; i < n; i++) {  if(i \* i >= n) break;  for(int j = i \* i; j < n; j += i) {  isPrime[j] = false;  }  }  int rslt = 0;  for(int i = 2; i < n; i++) {  rslt += isPrime[i];  }  return rslt;  }  }; |

# 206 反转链表(简单)

采用头插法

|  |
| --- |
| struct ListNode\* reverseList(struct ListNode\* head){  struct ListNode\* p = (struct ListNode \*)malloc(sizeof(struct ListNode));  p->next = NULL;  while(head) {  struct ListNode \*hNext = head->next;  head->next = p->next;  p->next = head;  head = hNext;  }  return p->next;  } |

# 209 长度最小的子数组(中等)

题目描述:

给定一个含有 n 个正整数的数组和一个正整数 target 。

找出该数组中满足其和 ≥ target 的长度最小的 连续子数组 [numsl, numsl+1, ..., numsr-1, numsr] ，并返回其长度。如果不存在符合条件的子数组，返回 0 。

思路:

数组元素由正整数组成, 可以如果计算前n项的和Sn, 则Sn是一个严格递增的序列.

那么可以找出i<j, Sj-Si≥target, 并且Sj-Si-target要尽可能小.

使用滑动窗口.

i在后面跑, j在前面跑, sum=A[i]+A[i+1]+...+A[j]

j向后移动, 直到sum≥target;

i向后移动, 直到sum＜target, 移动的过程中计算j和i的差值;

|  |
| --- |
| int minSubArrayLen(int target, int\* nums, int numsSize){  int i = 0, j = 0;  int sum = 0, rslt = **INT\_MAX**;  while(j < numsSize) {  while(j < numsSize && sum < target) {  sum += nums[j++];  }  while(sum >= target) {  rslt = fmin(rslt, j - i);  sum -= nums[i++];  }  }  return rslt == INT\_MAX ? 0 : rslt;  } |

编程知识补充(<<C和指针>>P30):

头文件limits.h说明了各种不同整数类型的特点. 下表为变量范围的限制.

|  |  |  |  |
| --- | --- | --- | --- |
|  | signed | | unsigned |
| 类型 | 最小值 | 最大值 | 最大值 |
| 字符 | SCHAR\_MIN | SCHAR\_MIN | UCHAR\_MAX |
| 短整型 | SHRT\_MIN | SHRT\_MAX | USHRT\_MAX |
| 整型 | INT\_MIN | INT\_MAX | UINT\_MAX |
| 长整型 | LONG\_MIN | LONG\_MAX | ULONG\_MAX |

# 217 存在重复元素(简单)

题目描述

给定一个整数数组，判断是否存在重复元素。

如果存在一值在数组中出现至少两次，函数返回 true 。如果数组中每个元素都不相同，则返回 false 。

使用hash表(unordered\_set)

|  |
| --- |
| class Solution {  public:  bool containsDuplicate(vector<int>& nums) {  unordered\_set<int> st;  for(int i = 0; i < nums.size(); i++) {  if(st.find(nums[i]) != st.end())  return true;  st.insert(nums[i]);  }  return false;  }  }; |

# 234 回文链表(简单)

使用三个指针b, m, e.

b指向表头固定不动, m和e从表头开始, m一步一步地走, e两步两步地走, 直到e走到末尾(e==NULL).

因为是验证回文, m最终要指向后一半的开头.

下面代码要注意vt.push\_back(m->val);和m = m->next;的语句位置.

|  |
| --- |
| class Solution {  public:  bool isPalindrome(ListNode\* head) {  ListNode \*b = head, \*m = head, \*e = head;  vector<int> vt;  while(e) {  e = e->next;  if(e) {  e = e->next;  vt.push\_back(m->val);  }  m = m->next;  }  int i = vt.size() - 1;  while(m) {  if(m->val != vt[i--])  return false;  m = m->next;  vt.pop\_back();  }  return true;  }  }; |

链表的结点有偶数个或奇数个, 对于m最终停留的位置

(1) 按照下面这种写法, m最终在后一半的开头.

解释: e最终一定指向NULL, e每个循环移动一步或两步, m一定移动一步, 所以m最终在后一半的开头.

|  |
| --- |
| while(e) {  e = e->next;  if(e) {  e = e->next;  }  m = m->next;  } |

如{1,2,3,4,5,6,7}, m最终指向5.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| e |  | e |  | e |  | e | e |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | NULL |
| m | m | m | m | m |  |  |  |

如{1,2,3,4,5,6}, m最终指向4.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| e |  | e |  | e |  | e |
| 1 | 2 | 3 | 4 | 5 | 6 | NULL |
| m | m | m | m |  |  |  |

(2) 按照下面这种写法, m最终在前一半的结尾.

解释: e最终一定指向NULL, e每个循环先移动一部, 非空时再和m一起移动一部, 这样一定要e走了2步, m才走一部, 所以m最终在前一半的开头.

|  |
| --- |
| while(e) {  e = e->next;  if(e) {  e = e->next;  m = m->next;  }  } |

如{1,2,3,4,5,6,7}, m指向4.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| e |  | e |  | e |  | e | e |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | NULL |
| m | m | m | m |  |  |  |  |

如{1,2,3,4,5,6}, m指向3.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| e |  | e |  | e |  | e |
| 1 | 2 | 3 | 4 | 5 | 6 | NULL |
| m | m | m |  |  |  |  |

# 237 删除链表中的节点(简单)

题目描述:

请编写一个函数，使其可以删除某个链表中给定的（非末尾）节点。传入函数的唯一参数为 要被删除的节点 。

链表至少包含两个节点。

链表中所有节点的值都是唯一的。

给定的节点为非末尾节点并且一定是链表中的一个有效节点。

不要从你的函数中返回任何结果。

|  |
| --- |
| void deleteNode(struct ListNode\* node) {  node->val = node->next->val;  node->next = node->next->next;  } |

# 242 有效的字母异位词(简单)

题目描述:

给定两个字符串 s 和 t ，编写一个函数来判断 t 是否是 s 的字母异位词。

|  |
| --- |
| bool isAnagram(char \* s, char \* t){  int ht[26] = {0};  while(\*s) {  ht[\*s - 'a']++;  s++;  }  while(\*t) {  ht[\*t - 'a']--;  t++;  }  for(int i = 0; i < 26; i++) {  if(ht[i])  return false;  }  return true;  } |

# 268 丢失的数字(简单)

|  |
| --- |
| class Solution {  public:  int missingNumber(vector<int>& nums) {  int sum = 0;  int len = nums.size();  if(len == 0) return 0;    for(int i = 0; i < len; i++){  sum = sum ^ i ^ nums[i];  }  return sum ^ len;  }  }; |

# 278 第一个错误的版本(简单)

# 283 移动零(简单)

# 295

# 297

# 300 最长递增子序列(中等)

题目描述:

给你一个整数数组 nums ，找到其中最长严格递增子序列的长度。

子序列是由数组派生而来的序列，删除（或不删除）数组中的元素而不改变其余元素的顺序。例如，[3,6,2,7] 是数组 [0,3,1,6,2,2,7] 的子序列。

动态规划方法

dp[j]表示以元素A[j]结尾的严格递增子序列子序列的长度. 最终得到最大的dp[]元素即可.

(1) 对于A[j]前面的元素(0≤i<j), 都有A[i]≥A[j], 则A[j]自己形成新的严格递增子序列, dp[j]=1.

(2) 对于A[j]前面的元素(0≤i<j), 存在A[i]<A[j], 则计算每一个A[i]+1, 取最大值, 赋值给dp[j]. 加1操作是指将A[j]拼接在以A[i]的严格递增子序列后面, 使得序列的长度增长1.

边界条件dp[0]=1.

对于每个dp[j]的计算,也可先赋值为1, 即先默认所有A[i]≥A[j], 发现A[i]<A[j]之后, 再执行计算A[i]+1的操作, 取最大值赋值给dp[j].

代码

|  |
| --- |
| class Solution {  public:  int lengthOfLIS(vector<int>& nums) {  int size = nums.size();  int rslt = 1;  vector<int> dp(size);  dp[0] = 1;  for(int j = 1; j < size; j++) {  dp[j] = 1;  for(int i = 0; i < j; i++) {  if(nums[i] < nums[j])  dp[j] = max(dp[i] + 1, dp[j]);  }  rslt = max(rslt, dp[j]);  }  return rslt;  }  }; |

性能: 400ms 10.MB

# 322

# 325

# 326 3的幂(简单)

# 344 反转字符串(简单)

# 350 两个数组的交集II

# 368 最大整除子集(中等)04.23

题目描述:

给你一个由 无重复 正整数组成的集合 nums ，请你找出并返回其中最大的整除子集 answer ，子集中每一元素对 (answer[i], answer[j]) 都应当满足：

answer[i] % answer[j] == 0 ，或

answer[j] % answer[i] == 0

如果存在多个有效解子集，返回其中任何一个均可。

此题与leetcode300相似

首先对nums[]从小到大排序.

dp[j]表示以nums[j]结尾的最大整除子集的大小.

(1) 对于所有nums[i](0≤i<j), 不存在nums[j]%nums[i]==0, 则nums[j]自己形成新的子集, dp[j]=1.

(2) 对于nums[i](0≤i<j), 存在nums[j]%nums[i]==0, 则dp[j]=max(dp[i]+1).

边界: 所有dp[i]=1.

上面只是能得到最大整除子集的大小, 但是构成子集的元素没有得到.

dp[]可以表示成结构体数组

struct Node {

int len;

int val;

int pre = -1;

};

找到len值最大的dp元素dp[maxIndex].

初始时i=maxIndex.

1) 输出dp[i].val;

2) 如果dp[i].pre!=-1, i=pre, 执行1); 否则结束.

|  |
| --- |
| class Solution {  public:  struct Node {  int len = 1;  int val;  int pre = -1;  };  vector<int> largestDivisibleSubset(vector<int>& nums) {  vector<Node> dp;  sort(nums.begin(), nums.end());  for(int j = 0; j < nums.size(); j++) {  dp.push\_back({1, nums[j], -1});  for(int i = 0; i < j; i++) {  if(nums[j] % nums[i] == 0) {  if(dp[j].len < dp[i].len + 1) {  dp[j].len = dp[i].len + 1;  dp[j].pre = i;  }  }  }  }  int maxIndex = 0;  for(int i = 0; i < dp.size(); i++) {  if(dp[maxIndex].len < dp[i].len)  maxIndex = i;  }    vector<int> rslt;  for(int i = maxIndex; i != -1; i = dp[i].pre) {  rslt.push\_back(dp[i].val);  }  return rslt;  }  }; |

# 384 打乱数组(中等)

# 387 字符串中的第一个唯一字符(简单)

# 394

# 399 除法求值(中等)

# 412 Fizz Buzz(简单)

# 417 太平洋大西洋水流问题(中等)

思路: 与994腐烂的橘子相似. 采用"逆流而上"的思维, 看看海洋的水可以逆流到哪些格子, 找出哪些可以被两个大洋逆流到的格子. 可以先找出太平洋可以逆流到的格子, 再找出大西洋可以逆流到的格子. 以太平洋为例, 一开始, 采用广度优先, 与太平洋相接的格子可以当作"腐烂的橘子", 然后按上左下右四个方向流动, 只要地势高或地势相等且没有被流过的格子就可以被流过.

pacMark[][]用于标记可以流到太平洋的格子.

antMark[][]用于标记可以流到大西洋的格子.

若(pacMark[i][j] == 1 && antMark[i][j] == 1)则[i][j]即可以流到太平洋也可以流到大西洋.

|  |
| --- |
| class Solution {  int rowMax, colMax;  struct Coor {  int x, y;  };  bool isCoorValid(Coor c) {  return c.x >= 0 && c.x <= rowMax && c.y >= 0 && c.y <= colMax;  }  public:  vector<vector<int>> pacificAtlantic(vector<vector<int>>& heights) {  queue<Coor> qe;  rowMax = heights.size() - 1;  colMax = heights[0].size() - 1;  int cnt = 0;  vector<vector<int>> pacMark(rowMax + 1, vector<int>(colMax + 1, 0));  vector<vector<int>> antMark(rowMax + 1, vector<int>(colMax + 1, 0));  // 这两个循环将与太平洋相接的格子入队  for(int row = 0; row <= rowMax; row++) {  Coor c = {row, 0};  qe.push(c);  pacMark[row][0]++;  cnt++;  }  for(int col = 1; col <= colMax; col++) {  Coor c = {0, col};  qe.push(c);  pacMark[0][col]++;  cnt++;  }  while(!qe.empty()) {  int tmpCnt = 0;  for(int i = 0; i < cnt; i++) {  Coor c = qe.front();  Coor around[4] = {{c.x - 1, c.y}, {c.x, c.y + 1},  {c.x + 1, c.y}, {c.x, c.y - 1}}; // 上右下左  for(int k = 0; k < 4; k++) {  int x = around[k].x, y = around[k].y;  if(isCoorValid(around[k]) &&  pacMark[x][y] < 1 &&  heights[x][y] >= heights[c.x][c.y]) {  qe.push(around[k]);  pacMark[x][y]++;  tmpCnt++;  }  }  qe.pop();  }  cnt = tmpCnt;  }  cnt = 0;  // 这两个循环将与大西洋相接的格子入队  for(int row = 0; row <= rowMax; row++) {  Coor c = {row, colMax};  qe.push(c);  antMark[row][colMax]++;  cnt++;  }  for(int col = 0; col <= colMax - 1; col++) {  Coor c = {rowMax, col};  qe.push(c);  antMark[rowMax][col]++;  cnt++;  }  while(!qe.empty()) {  int tmpCnt = 0;  for(int i = 0; i < cnt; i++) {  Coor c = qe.front();  Coor around[4] = {{c.x - 1, c.y}, {c.x, c.y + 1},  {c.x + 1, c.y}, {c.x, c.y - 1}}; // 上右下左  for(int k = 0; k < 4; k++) {  int x = around[k].x, y = around[k].y;  if(isCoorValid(around[k]) &&  antMark[x][y] < 1 &&  heights[x][y] >= heights[c.x][c.y]) {  qe.push(around[k]);  antMark[x][y]++;  tmpCnt++;  }  }  qe.pop();  }  cnt = tmpCnt;  }  vector<vector<int>> rslt;  for(int i = 0; i <= rowMax; i++) {  for(int j = 0; j <= colMax; j++) {  if(antMark[i][j] && pacMark[i][j]) {  rslt.push\_back({i, j});  }  }  }  return rslt;  }  }; |

注意: 不可以对格子被流过的次数计数, 然后计数=2就认为既可以流到太平洋也可以流到大西洋. 这样做会导致重复计数. 应该用两个数组分别标记坐标[i,j]是否可以流到太平洋或大西洋, 然后挑出数组元素值均为1的下标.

# 435 无重叠的区间(中等)

思路: 无重叠区间问题, 与452一样.

|  |
| --- |
| class Solution {  public:    static bool cmp(vector<int> &a, vector<int> &b){  if(a[0] != b[0]) return a[0] > b[0];  else return a[1] <= b[1];  }  int eraseOverlapIntervals(vector<vector<int>>& intervals) {  int size = intervals.size();  if(size == 0) return 0;  sort(intervals.begin(), intervals.end(), cmp);  int lastx = intervals[0][0];  int cnt = 1;  for(int i = 1; i < size; i++){  if(intervals[i][1] <= lastx){  cnt++;  lastx = intervals[i][0];  }  }  return size - cnt;  }  }; |

# 452 用最少数量的箭引爆气球(中等)

问题的本质: 给定N个闭区间[x, y], 求最少需确定多少个点, 才能使每个闭区间中都至少存在一个点.

区间贪心: 给出N个开区间(x, y), 从中选择尽可能多的开区间, 使得这些开区间两两没有交集.

本题就是反过来, 将两两相交的区间归类, 求类别的数量.

思路: 对N个区间按x从大到小, 按y从小到大排序. 排好序后, 从第[0]个区间开始, 找到下一个区间, 要满足: 下一个区间的y<第[0]个区间的x, 找到区间的数量cnt++. 重复上升过程, 直到最后一个区间被遍历完了.

|  |
| --- |
| class Solution {  public:  static bool cmp(vector<int> a, vector<int> b) {  if(a[0] != b[0]) return a[0] > b[0];  else return b[1] < a[1];  }  int findMinArrowShots(vector<vector<int>>& points) {  if(points.size() == 0) return 0;  sort(points.begin(), points.end(), cmp);  int lastx = points[0][0];  int rslt = 1;  for(int i = 1; i < points.size(); i++) {  if(points[i][1] >= lastx) continue;  rslt++;  lastx = points[i][0];  }  return rslt;  }  }; |

# 518 零钱兑换II(中等)

题目描述:

给定不同面额的硬币和一个总金额。写出函数来计算可以凑成总金额的硬币组合数。假设每一种面额的硬币有无限个。

这个问题是完全背包问题的变种.

dp[i][v] 前i件物品恰好放入容量为v的背包中能获得的最大价值.

(1) 不放第i件物品, dp[i][v]=dp[i-1][v]

(2) 放第i件物品, dp[i][v]=dp[i][v-w[i]]+c[i]

dp[i][j] 使用前i种金币可以恰好可以组合成总金额j的方案数量.

0≤i≤coins.size(), 0≤j≤amount

注意: 前i种金币, 则最后一种金币是c[i-1]

故数组的形状为[coins.size()+1][amount+1]

如果要凑的金额为0, 那么方法就是1种: dp[][0]=1

如果不使用任何面额, 那么方法就是0种: dp[0][]=0

注意dp[0][0]=1

(1) 不使用第i种金币, dp[i][j] = dp[i-1][j]

(2) 使用第i种金币, dp[i][j] = dp[i][j-c[i-1]]

(3) 如果c[i-1]>j, 那就只能采取方法(1)

if(j<c[i-1]) dp[i][j]=dp[i-1][j];

else dp[i][j]=dp[i-1][j]+dp[i-1][j-c[i-1]];

|  |
| --- |
| class Solution {  public:  int change(int amount, vector<int>& coins) {  vector<vector<int>> dp(coins.size() + 1, vector<int>(amount + 1));    for(int i = 0; i <= coins.size(); i++)  dp[i][0] = 1;    for(int j = 1; j <= amount; j++)  dp[0][j] = 0;  for(int i = 1; i <= coins.size(); i++) {  for(int j = 1; j <= amount; j++) {  if(j < coins[i - 1]) dp[i][j] = dp[i - 1][j];  else dp[i][j] = dp[i - 1][j] + dp[i][j - coins[i - 1]];  }  }  return dp[coins.size()][amount];  }  }; |

性能: 64ms, 17.9MB

对于dp[][]每次计算都只需要用到正上方和左侧的某个元素, 所以可以采用滚动数组的方法(注意j要从1顺序遍历到amount).

|  |
| --- |
| for(int i = 1; i <= coins.size(); i++) {  for(int j = 1; j <= amount; j++) {  if(j < coins[i - 1]) dp[j] = dp[j];  else dp[j] = dp[j] + dp[j - coins[i - 1]];  }  } |

注意到dp[j] = dp[j];这种运算是没有意义的, 相当于什么也没有执行, 而它的执行条件是j<coins[i-1],

那么遍历范围改为coins[i-1]→amount就可以了.

边界条件变为dp[0]=1, 初始时dp[]其它值为0, 数组形状为[amount + 1].

所以

|  |
| --- |
| for(int i = 1; i <= coins.size(); i++) {  for(int j = coins[i - 1]; j <= amount; j++) {  dp[j] = dp[j] + dp[j - coins[i - 1]];  }  } |

改进后的代码如下:

|  |
| --- |
| class Solution {  public:  int change(int amount, vector<int>& coins) {  vector<int> dp(amount + 1, 0);  dp[0] = 1;  for(int i = 1; i <= coins.size(); i++) {  for(int j = coins[i - 1]; j <= amount; j++) {  dp[j] = dp[j] + dp[j - coins[i - 1]];  }  }  return dp[amount];  }  }; |

性能: 16ms, 6.9MB

# 572

# 739

# 781 森林中的兔子(中等)

题目描述:

森林中，每个兔子都有颜色。其中一些兔子（可能是全部）告诉你还有多少其他的兔子和自己有相同的颜色。我们将这些回答放在 answers 数组里。

返回森林中兔子的最少数量。

示例:

输入: answers = [1, 1, 2]

输出: 5

思路:

对于数组中的数字, 由于兔子没有说谎, 那么兔子说的数字x, 就应该能够(x+1)和(x+1)匹配.

例如数字1应该两两匹配, 数字2应该三三匹配. 对于数组[1, 1, 2], 1可以两两匹配, 2只有一个, 因为2要三三匹配, 所以应该加两个2, 才能完成颜色的匹配, [1, 1, 2, 2, 2], 所以兔子有5个.

数字x出现的次数应该是(x+1)的整数倍.

对于任意整数x和y, 求出最小z≥x, 且z%y==0的方法:

z=x+((bool)(x%y))\*(y-(x%y)), 记为fun(x, y)

编程方法: 使用hash表

对answers中的数字进行计数, hash[answers[i]]表示answers[i]出现的次数. 然后对

fun(hash[answers[i]], answers[i] + 1)求和, 注意不要重复计算, hash计算完后清零, 只有不为零的hash[answers[i]]才计算.

|  |
| --- |
| // x按y向上对齐  int align(x, y) {  int t = x % y;  return x + ((bool)t) \* (y - t);  }  int numRabbits(int\* answers, int answersSize){  int ht[1001] = {0};  for(int i = 0; i < answersSize; i++)  ht[answers[i]]++;    int rslt = 0;  for(int i = 0; i < answersSize; i++) {  if(ht[answers[i]]) {  rslt += align(ht[answers[i]], answers[i] + 1);  ht[answers[i]] = 0;  }  }  return rslt;  } |

性能: 0ms(100%), 5.8MB(44%)

# 796

# 844 比较含退格的字符串(简单)

思路: 指针从后往前遍历字符串, 用一个变量skip记录'#'的数量, 指针一直往前走, 直到遇到的不是'#', 然后指针再往前走skip步, 直到指针越过了字符串的开头. 用这种方法同步处理两个字符串.

|  |
| --- |
| class Solution {  public:  bool backspaceCompare(string S, string T) {  int i = S.size() - 1;  int j = T.size() - 1;  int skipS = 0, skipT = 0;  while(i >= 0 || j >= 0) {  while(i >= 0 && S[i] == '#') {  // i移动到下一个有效的非'#'字符  while(i >= 0 && S[i] == '#') i--, skipS++;  // 准备删除skipS个字符, 但要考虑到删除过程中可能又会遇到'#'  // 如果这样则要令skipS++  while(skipS && i >= 0) {  if(S[i] == '#') skipS++;  else skipS--;  i--;  }  }  while(j >= 0 && T[j] == '#') {  // j移动到下一个有效的非'#'字符  while(j >= 0 && T[j] == '#') j--, skipT++;  // 准备删除skipT个字符, 但要考虑到删除过程中可能又会遇到'#'  // 如果这样则要令skipT++  while(skipT && j >= 0) {  if(T[j] == '#') skipT++;  else skipT--;  j--;  }  }  if(i < 0 && j < 0) return true;  if(i < 0 || j < 0) return false;  if(S[i] != T[j]) return false;  i--, j--;  }  return i == j;  }  }; |

注意: 最外层循环要使用(i >= 0 || j >= 0)判断, 而不是(i >= 0 && j >= 0), 因为只要有一个下标不合法就应该退出循环了.

# 983

# 989

# 994 腐烂的橘子(中等)

初步思路: 开始时将值为2的橘子入队. 记录这一轮入队的数量cnt.

(1) 出队cnt个橘子, 则出队每个橘子值为2, 对于出队的每个橘子, 将橘子周围值为1的橘子入队, 并令其值为2. 将这一轮入队的橘子数量(先用tmpcnt保存)记录在cnt.

(2) 队列为空, 转(3), 否则分钟数++, 转(1).

(3) 若不存在值为一的橘子, 返回分钟数, 否则返回-1;

|  |
| --- |
| class Solution {  int n;  int rowMax, colMax;  public:  struct Coor{  int x, y;  };  bool coorValid(Coor a){  return a.x >= 0 && a.x <= rowMax && a.y >= 0 && a.y <= colMax;  }  int orangesRotting(vector<vector<int>>& grid) {  queue<Coor> q;  int rowNum = grid.size();  int colNum = grid[0].size();  **rowMax = rowNum - 1;**  **colMax = colNum - 1;**  int cnt = 0;  int minute = 0;  for(int i = 0; i < rowNum; i++) {  for(int j = 0; j < colNum; j++) {  if(grid[i][j] == 2){  q.push({i, j});  cnt++;  }  }  }  while(!q.empty()) {  int tmpCnt = 0;    for(int i = 0; i < cnt; i++) {  Coor co = q.front();  q.pop();  Coor around[4] = {{co.x - 1, co.y}, {co.x, co.y + 1},  {co.x + 1, co.y}, {co.x, co.y - 1}}; // 上右下左  for(int k = 0; k < 4; k++){  int x = around[k].x, y = around[k].y;  if(coorValid(around[k]) && grid[x][y] == 1) {  grid[x][y] = 2;  q.push(around[k]);  tmpCnt++;  }  }    }  **if(!q.empty()) minute++;**  cnt = tmpCnt;  }  for(int i = 0; i < rowNum; i++) {  for(int j = 0; j < colNum; j++) {  if(grid[i][j] == 1){  return -1;  }  }  }  return minute;  }  }; |

注意点:

给定的矩形不一定为正方形.

每一轮结束后队列非空才可以累加分钟数.

# 1122

# 1297 杨辉三角(简单)

# 1300

# 1361 验证二叉树(简单)

# 1143 最长公共子序列(困难)

题目描述:

给定两个字符串 text1 和 text2，返回这两个字符串的最长 公共子序列 的长度。如果不存在 公共子序列 ，返回 0 。

一个字符串的 子序列 是指这样一个新的字符串：它是由原字符串在不改变字符的相对顺序的情况下删除某些字符（也可以不删除任何字符）后组成的新字符串。

例如，"ace" 是 "abcde" 的子序列，但 "aec" 不是 "abcde" 的子序列。

两个字符串的 公共子序列 是这两个字符串所共同拥有的子序列。

动态规划

dp[i][j]表示str1的前i个字符组成的字串和str2的前j个字符组成的子串的最长公共子序列的长度.

0≤i≤len1, 0≤j≤len2, 所以dp的形状为[len1+1][len2+1]

边界:

空串和任意字符串的最长公共子序列的长度为0, dp[0][j]=0, dp[i][0]=0;

对于dp[i][j]

(1) str1[i-1]==str2[j-1], dp[i][j]=dp[i-1][j-1]+1.

(2) str1[i-1]!=str2[j-1], dp[i][j]=max{dp[i-1][j], dp[i][j-1]}.

手动模拟

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | j=0 "" | j=1 a | j=2 c | j=3 e |
| i=0 "" | 0 | 0 | 0 | 0 |
| i=1 a | 0 | 1 | max{1,0}=1 | max{1,0}=1 |
| i=2 b | 0 | max{0,1}=1 | max{1,1}=1 | max{1,1}=1 |
| i=3 c | 0 | max{0,1}=1 | 1+1=2 | max{1,1}=1 |
| i=4 d | 0 | max{0,1}=1 | max{1,2}=2 | max{2,1}=2 |
| i=5 e | 0 | max{0,1}=1 | max{1,2}=2 | 2+1=3 |

最终答案为3.

初始答案

|  |
| --- |
| class Solution {  public:  int longestCommonSubsequence(string text1, string text2) {  int len1 = text1.size(), len2 = text2.size();  vector<vector<int>> dp(len1 + 1, vector<int>(len2 + 1, 0));  for(int i = 1; i <= len1; i++) {  for(int j = 1; j <= len2; j++) {  if(text1[i - 1] == text2[j - 1])  dp[i][j] = dp[i - 1][j - 1] + 1;  else  dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);  }  }  return dp[len1][len2];  }  }; |

性能: 28ms, 12.7MB

滚动数组

每次计算dp[i][j]都只用到左上, 上方和左方的元素

dp[]初始值为全0, 长度为len2+1

(1) str1[i-1]==str2[j-1], dp[j]=dp[j-1]+1.

(2) str1[i-1]!=str2[j-1], dp[i][j]=max{dp[j], dp[j-1]}.

01背包问题是用到了左上和上方的元素, 所以子循环要倒着遍历.

|  |
| --- |
| for(int i = 1; i <= n; i++) {  for(int v = V; v >= w[i]; v--) {  dp[v] = fmax(dp[v - w[i]] + c[i], dp[v]);  rslt = fmax(rslt, dp[v]);  }  } |

完全背包问题是用到了左方和上方的元素, 所以子循环要顺着遍历

|  |
| --- |
| for(int i = 1; i <= n; i++)  for(int v = w[i]; v <= V; v++)  dp[i][v] = max{dp[v], dp[v - w[i]] + c[i]}; |

.

而本题同时用到了左上, 上方和左方的元素.

如果顺着遍历, 测试用例

"aaa"

"aaaaaa"

会输出7, 正确答案3.

如果倒着遍历, 测试用例

"hofubmnylkra"

"pqhgxgdofcvmr"

会输出4, 正确答案5.

可以看到对比到o时, 子答案为1, 忽略了前面h的结果.

|  |
| --- |
| \* p q h g x g d o f c v m r  i=00 \* 0 0 0 0 0 0 0 0 0 0 0 0 0 0  i=01 h 0 0 0 1 0 0 0 0 0 0 0 0 0 0  i=02 o 0 0 0 1 1 0 0 0 1 0 0 0 0 0  i=03 f 0 0 0 1 1 1 0 0 1 2 0 0 0 0  i=04 u 0 0 0 1 1 1 1 0 1 2 2 0 0 0  i=05 b 0 0 0 1 1 1 1 1 1 2 2 2 0 0  i=06 m 0 0 0 1 1 1 1 1 1 2 2 2 3 0  i=07 n 0 0 0 1 1 1 1 1 1 2 2 2 3 3  i=08 y 0 0 0 1 1 1 1 1 1 2 2 2 3 3  i=09 l 0 0 0 1 1 1 1 1 1 2 2 2 3 3  i=10 k 0 0 0 1 1 1 1 1 1 2 2 2 3 3  i=11 r 0 0 0 1 1 1 1 1 1 2 2 2 3 4  i=12 a 0 0 0 1 1 1 1 1 1 2 2 2 3 4 |

所以这道题因该不可以用滚动数组?

其实还是可以用, 用一个pre来保存上一轮的计算结果即可, 子循环顺着遍历.

|  |
| --- |
| class Solution {  public:  int longestCommonSubsequence(string text1, string text2) {  int len1 = text1.size(), len2 = text2.size();  vector<int> dp(len2 + 1, 0);  vector<int> pre = dp;  int rslt = 0;  for(int i = 1; i <= len1; i++) {  for(int j = 1; j <= len2; j++) {  if(text1[i - 1] == text2[j - 1])  dp[j] = pre[j - 1] + 1;  else  dp[j] = max(pre[j], dp[j - 1]);  }  pre = dp;  }  return dp[len2];  }  }; |

性能: 16ms, 6.4MB

滚动数组小结

(1) 用到左上和上方元素, 子循环倒着遍历

(2) 用到左方和上方元素, 子循环顺着遍历

(3) 用到左上和上方和左方的元素, 用一个pre数组保存上一轮计算的结果, 子循环顺着遍历.

# 01背包问题

有n件物品, 每件物品的重量为w[i], 价值为从c[i]. 现有一个容量为V的背包, 问如何选取物品放入背包, 使得背包内的物品的总价值最大. 其中每种物品都只有1件.

样例:

5 8 // n=5 V=8

3 5 1 2 2 // w[i]

4 5 2 1 3 // c[i]

输出: 10

dp[i][v] 前i件物品恰好装入容量为v的背包中所获得的最大价值.

1≤i≤n, 0≤v≤V.

(1) 不放第i件物品, 转化为求dp[i-1][v]

(2) 放第i件物品, 转化为dp[i-1][v-w[i]]+c[i]

dp[i][v]=max{dp[i-1][v], dp[i-1][v-w[i]]+c[i]}

|  |
| --- |
| for(int i = 1; i <= n; i++)  for(int v = w[i]; v <= V; v++)  dp[i][v] = max{dp[i - 1][v], dp[i - 1][v - w[i]] + c[i]}; |

v可以直接从w[i]开始, 因为若v<w[i]则装不下, 价值就是dp[i-1][v]

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| dp[i-1] |  | dp[i-1][v-w[i]] |  |  |  | dp[i-1][v] |  |  |  |
| dp[i] |  |  |  |  |  | dp[i][v] |  |  |  |

技巧: 滚动数组

计算dp[i][v]每次都只需要用到上一个行的值, 所以不需要存储dp[][]第一维.

|  |
| --- |
| for(int i = 1; i <= n; i++)  for(int v = V; v >= w[i]; v--)  dp[v] = max{dp[v], dp[v - w[i]] + c[i]}; |

注意:

v是从V到w[i]逆序遍历的, 如果是顺序遍历, dp[v]的计算用到前面的dp[]可能是被当前这次回合计算改变的值, 但是, dp[v]只能根据上一回合的dp[]来计算, 所以v只能从后往前遍历.

如果是原始的dp[][]二维数组的方法, 则可以按逆序或者顺序来求dp[v], 这是因为dp[i][v].

完整代码

|  |
| --- |
| #include <cstdio>  #include <cmath>  const int maxn = 100;  const int maxv = 1000;  int w[maxn], c[maxn], dp[maxv];  int main(int argc, char \*\*argv) {  int n, V;  scanf("%d %d", &n, &V);  for(int i = 0; i < n; i++) {  scanf("%d", &w[i]);  }  for(int i = 0; i < n; i++) {  scanf("%d", &c[i]);  }  for(int v = 0; v <= V; v++) {  dp[v] = 0;  }  int rslt = 0;  for(int i = 1; i <= n; i++) {  //for(int v = w[i]; v <= V; v++) {  for(int v = V; v >= w[i]; v--) {  dp[v] = fmax(dp[v - w[i]] + c[i], dp[v]);  rslt = fmax(rslt, dp[v]);  }  }  printf("%d\n", rslt);  return 0;  } |

# 完全背包问题

有n件物品, 每件物品的重量为w[i], 价值为从c[i]. 现有一个容量为V的背包, 问如何选取物品放入背包, 使得背包内的物品的总价值最大. 其中每种物品都只有无穷件.

leetcode 518题

dp[i][v] 前i件物品恰好放入容量为v的背包中能获得的最大价值.

(1) 不放第i件物品, dp[i][v]=dp[i-1][v]

(2) 放第i件物品, dp[i][v]=dp[i][v-w[i]]+c[i]

这是因为物品有无穷多件, 放了第i件物品还可以放第i件物品, 直到无法满足v-w[i]≥0为止.

dp[i][v]=max{dp[i-1][v], dp[i][v-w[i]]+c[i]}

如果使用二维数组, dp[i][v]的计算要用到上一行的正上方的元素和这一行的左边的某个元素, 那么使用压缩存储, 可以写成:

|  |
| --- |
| for(int i = 1; i <= n; i++)  for(int v = w[i]; v <= V; v++)  dp[i][v] = max{dp[v], dp[v - w[i]] + c[i]}; |

# 最长不下降子序列

在一个数字序列中, 找到一个最长的子序列(可以不连续), 使得这个子序列是不下降（非递减）的.

例如A={1,2,3,-1,-2,7,9}, 最长不下降子序列为{1,2,3,7,9}.

相似题目 leetcode 300

# 最长回文子串

见leetcode 5

# DAG最长路

DAG就是有向无环图, 最长路也就是关键路径.

两个问题:

① 求整个DAG中的最长路径(即不固定起点和终点)

② 固定终点, 求DAG的最长路径.

给定一个有向无环图, 怎样求整个图的所有路劲中权值之和最大的那条?

dp[i]表示从i号顶点出发能获得的最长路径长度, 这样max(dp[i])就是整个DAG的最长路径长度.

dp[i]=max{dp[j]+len[i→j]|(i,j)∈E}

边界: 出度为0的顶点dp[i]=0.

不妨令初始时dp[]=0.

|  |
| --- |
| int DP(int i) {  if(dp[i] > 0) return dp[i];  for(int j = 0; j < n; j++) {  if(G[i][j] != INF)  dp[i] = max{dp[i], DP[j] + G[i][j]};  }  } |

如何存储最长路径上的每个顶点?

开一个int数组choice来记录最长路径上顶点的后继顶点.

choice[i]=j 表示i顶点的后继结点为j. choice[]初始值为-1.

|  |  |
| --- | --- |
| int DP(int i) {  if(dp[i] > 0) return dp[i];  for(int j = 0; j < n; j++) {  if(G[i][j] != INF) {  **int t = DP(j) + G[i][j];**  **if(t > dp[i]) {**  **dp[i] = t;**  **choice[i] = j;**  **}**  }  }  }  void printPath(int i) {  printf("%d ", i);  while(choice[i] != -1) {  i = choice[i];  printf("->%d", i);  }  } |  |

固定终点T, 求DAG的最长路径长度.

dp[i]表示从i号顶点出发到达终点T能获得的最长路径长度.

dp[i]=max{dp[j]+len[i→j]|(i,j)∈E}

边界: dp[T]=0.

可以初始化dp[]为-INF, 设置一个vis数组来表示顶点是否已经被计算.

(1) 为什么初始值不设置成INF? 因为计算用有dp[i]=max(dp[i], DP[j]+G[i][j]), 如果初始值为INF, 则会一直无法更新dp[i]

(2) 为什么第一个问题不需要设置vis数组来标记是否被计算过? 因为第一个问题中dp[]初始值为0, 如果dp[]被计算过, dp一定会变为非零, 或者有些顶点没有出度, 它不需要被计算. 而本题中-INF表示无法到达, dp[T]=0, 如果i能达到T, dp[i]为一个正整数(不是INF), 被计算过以后, dp[]仍有可能是-INF.

|  |
| --- |
| int DP(int i) {  if(vis[i]) return dp[i];  vis[i] = true;  for(int j = 0; j < n; j++) {  if(G[i][j] != INF) {  dp[i] = max(dp[i], DP[j] + G[i][j]);  }  }  return dp[i];  } |