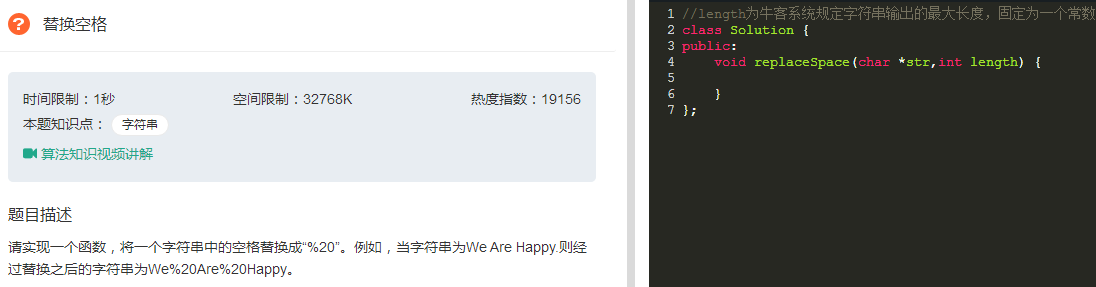
# 二维数组中的查找



|  |
| --- |
| **class Solution {**  **public:**  **bool Find(vector<vector<int> > array,int target) {**  **bool found = false;**  **if(array.size() > 0 && array[0].size() > 0)**  **{**  **int rows = array.size(),columns = array[0].size(),row = 0, column = columns - 1 ;**  **while(row < rows && column >= 0)**  **{**  **int tmp = array[row][column];**  **if(tmp == target)**  **{**  **found = true;**  **break;**  **}**  **else if(tmp > target)**  **--column;**  **else**  **++row;**  **}**  **}**  **return found;**  **}**  **//length为牛客系统规定字符串输出的最大长度，固定为一个常数**  **void replaceSpace(char \*str,int length) {**  **if(str == NULL && length == 0)**  **return ;**  **int realLength = 0,numberOfBlank = 0;**  **int idx = 0 ;**  **while(str[idx]!='\0')**  **{**  **++realLength;**  **if(str[idx]==' ')**  **++numberOfBlank;**  **++idx;**  **}**  **int newLength = realLength + numberOfBlank \* 2;**  **if(newLength > length) return ;**  **int indexOfOriginal = realLength;**  **int indexOfNew = newLength;**  **while(indexOfOriginal >= 0 && indexOfNew > indexOfOriginal)**  **{**  **if(str[indexOfOriginal] == ' ')**  **{**  **str[indexOfNew--] = '0';**  **str[indexOfNew--] = '2';**  **str[indexOfNew--] = '%';**  **}**  **else**  **{**  **str[indexOfNew--] = str[indexOfOriginal];**  **}**  **--indexOfOriginal;**  **}**  **}**  **};** |

# 替换空格



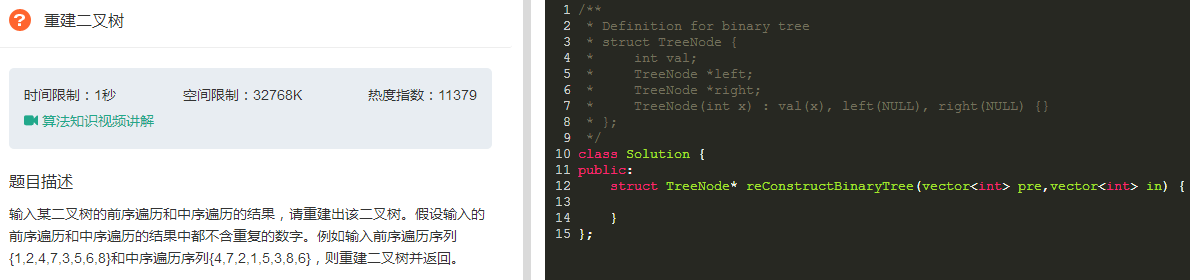
|  |
| --- |
| //length为牛客系统规定字符串输出的最大长度，固定为一个常数  class Solution {  public:  void replaceSpace(char \*str,int length) {  if(str == NULL && length == 0)  return ;  int realLength = 0,numberOfBlank = 0;  int idx = 0 ;  while(str[idx]!='\0')  {  ++realLength;  if(str[idx]==' ')  ++numberOfBlank;  ++idx;  }  int newLength = realLength + numberOfBlank \* 2;  if(newLength > length) return ;  int indexOfOriginal = realLength;  int indexOfNew = newLength;  while(indexOfOriginal >= 0 && indexOfNew > indexOfOriginal)  {  if(str[indexOfOriginal] == ' ')  {  str[indexOfNew--] = '0';  str[indexOfNew--] = '2';  str[indexOfNew--] = '%';  }  else  {  str[indexOfNew--] = str[indexOfOriginal];  }  --indexOfOriginal;  }  }  }; |

# 从头到尾打印链表



|  |
| --- |
| /\*\*  \* struct ListNode {  \* int val;  \* struct ListNode \*next;  \* ListNode(int x) :  \* val(x), next(NULL) {  \* }  \* };  \*/  class Solution {  public:  vector<int> printListFromTailToHead(struct ListNode\* head) {  vector<int> vec;  if(head != NULL)  {  vec.push\_back(head->val);  while(head->next != NULL){  vec.push\_back(head->next->val);  head=head->next;  }  reverse(vec.begin(),vec.end());  }  return vec;  }  }; |

# 重建二叉树



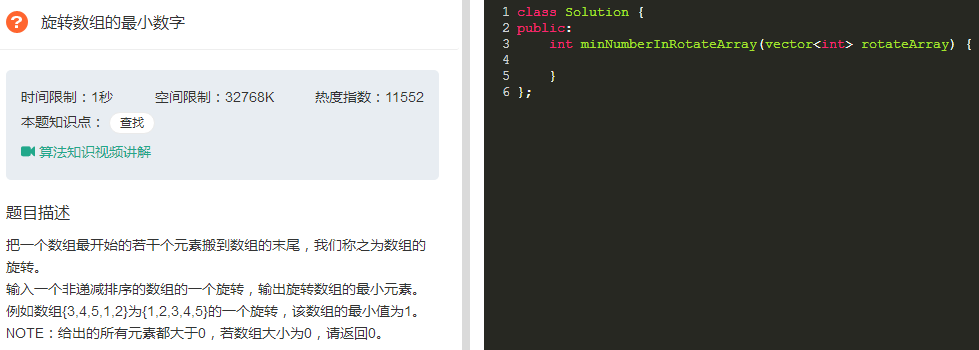
|  |
| --- |
| class Solution {  public:  struct TreeNode \* subTree(vector<int> pre, int preStart, int preEnd, vector<int> in, int inStart, int inEnd) {  struct TreeNode \* rootTree = (struct TreeNode \*) malloc (sizeof(struct TreeNode));  rootTree -> val = pre[preStart];  rootTree -> left = NULL;  rootTree -> right = NULL;  if (preStart == preEnd && inStart == inEnd && pre[preStart] == in[inStart])  return rootTree;  int rootIndex = inStart;  while (in[rootIndex] != pre[preStart])  rootIndex ++;  int newLength = rootIndex - inStart;  if (newLength > 0)  rootTree -> left = subTree(pre, preStart + 1, preStart + newLength, in, inStart, rootIndex - 1);  if (inEnd - rootIndex > 0)  rootTree -> right = subTree(pre, preStart + newLength + 1, preEnd, in, rootIndex + 1, inEnd);  return rootTree;  }  struct TreeNode\* reConstructBinaryTree(vector<int> pre,vector<int> in) {  int preLength = pre.size();  int inLength = pre.size();  if (preLength == 0 || inLength == 0)  return NULL;  return subTree(pre, 0, preLength - 1, in, 0, inLength - 1);  }  }; |

# 用两个栈实现一个队列



|  |
| --- |
| class Solution  {  public:  void push(int node) {  stack1.push(node);  }  int pop() {  if(stack2.empty()){  //stack1 往stack2转移  while(!stack1.empty()){  //  int tmp = stack1.top();  stack2.push(tmp);  stack1.pop();  }  }  int ret = stack2.top();  stack2.pop();  return ret;  }    private:  stack<int> stack1;  stack<int> stack2;  }; |

# 旋转数组最小数字（待优化）



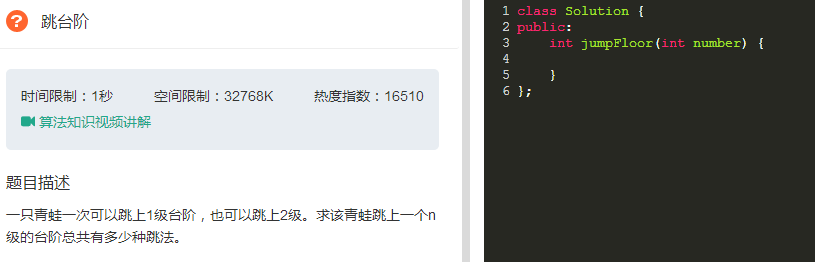
|  |
| --- |
| class Solution {  public:  int minNumberInRotateArray(vector<int> rotateArray) {  if(0 == rotateArray.size() || 1 == rotateArray.size()) return 0;  int pre,idx,sz=rotateArray.size();  pre = rotateArray[0];  for(idx = 1 ; idx != sz ; ++idx)  {  if(pre > rotateArray[idx])  return rotateArray[idx];  }  return 0;  }  }; |

# 斐波那契数列



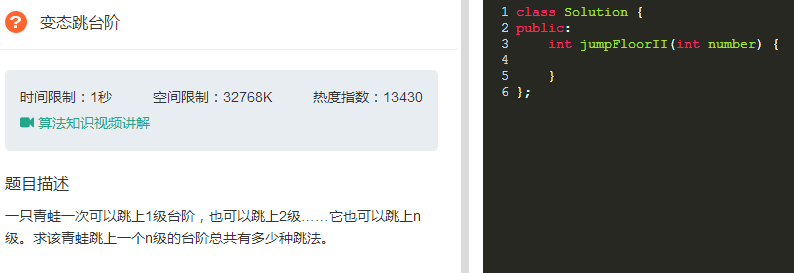
|  |
| --- |
| class Solution {  public:  int Fibonacci(int n) {  int \*arr = new int[40];  arr[0]=0;  arr[1]=arr[2]=1;  for(int i=3;i<40;i++){  arr[i]=arr[i-1]+arr[i-2];  }  return arr[n];  }  }; |

# 跳台阶（斐波那契数列）



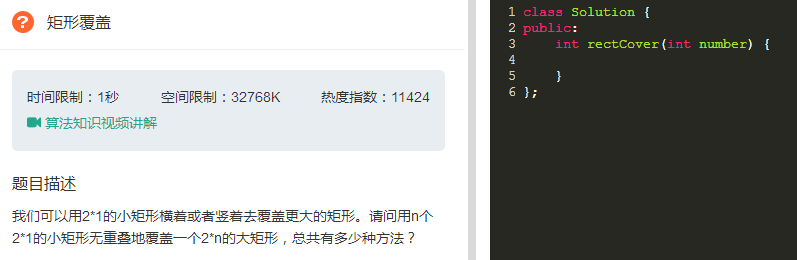
|  |
| --- |
| class Solution {  public:  int jumpFloor(int number) {  if(1 == number || 2 == number) return number;  else return jumpFloor(number - 1) + jumpFloor(number-2);  }  }; |

# 变相跳台阶（2^n-1）数学归纳



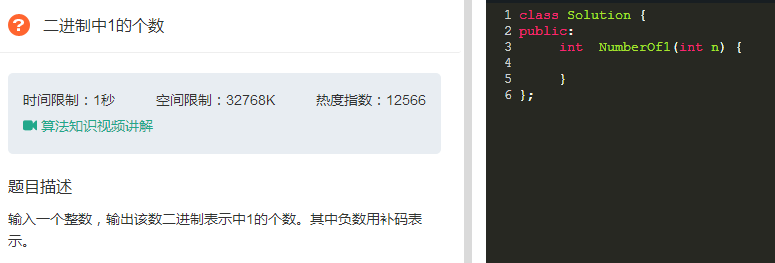
|  |
| --- |
| class Solution {  public:  int jumpFloorII(int number) {  int sum = 1;  if(number == 1) return sum;  for(int i = 1 ; i < number; ++i)  sum \*= 2;  return sum;  }  }; |

# 矩阵覆盖（斐波那契数列）



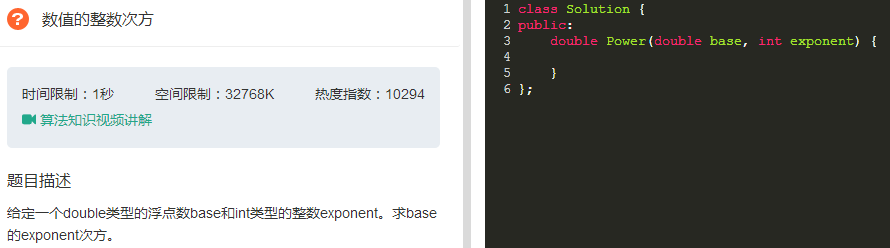
|  |
| --- |
| class Solution {  public:  int rectCover(int number) {  int \*arr = new int[100];  arr[0] = arr[1] = 1;  for(int i = 2; i<100;++i)  arr[i]=arr[i-1]+arr[i-2];  arr[0] = 0;  int flag = arr[number];  delete []arr;  return flag;  }  }; |

# 二进制中1的个数



|  |
| --- |
| //n=n&(n-1)方法更简单  class Solution {  public:  int NumberOf1(int n) {  int count = 0;  unsigned int flag = 1;  while(flag){  if(n&flag)  count++;  flag = flag<<1;  }  return count;  }  }; |

# 数值的整数次方



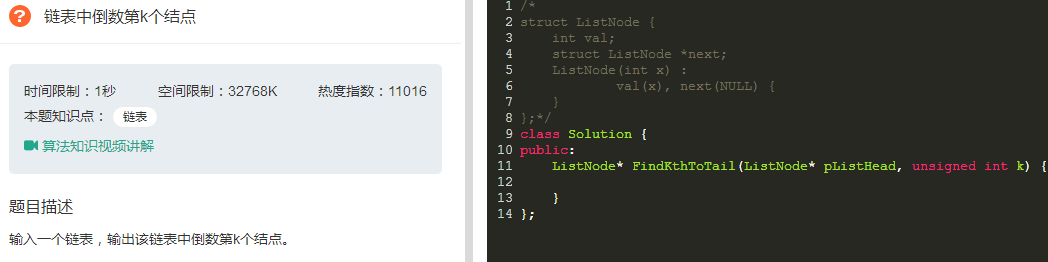
|  |
| --- |
| class Solution {  public:  double Power(double base, int exponent) {  double sum = 1 ;  if(exponent >= 0){  for(int i = 0 ; i < exponent ; i++){  sum = sum \* base ;  }  }else{  for(;exponent < 0;exponent++){  sum = sum / base ;  }  }  return sum ;  }  }; |

# 调整数组顺序使奇数位于偶数前面



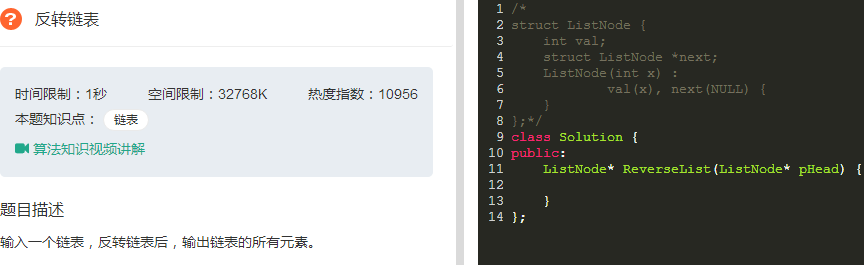
|  |
| --- |
|  |

# 链表中倒数第k个结点



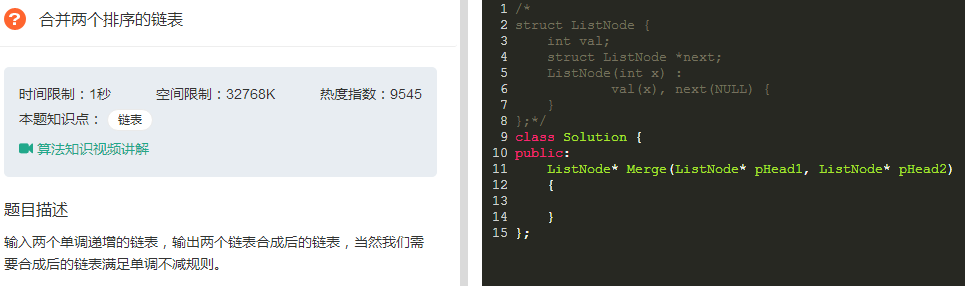
|  |
| --- |
| /\*  struct ListNode {  int val;  struct ListNode \*next;  ListNode(int x) :  val(x), next(NULL) {  }  };\*/  class Solution {  public:  ListNode\* FindKthToTail(ListNode\* pListHead, unsigned int k) {  if(pListHead==NULL||k==0)  return NULL;  ListNode \*tmp = pListHead;  unsigned int idx = 0 ;  while(pListHead != NULL){  if(idx < k ){  ++idx ;  }else{  // cout << tmp->val << " " << pListHead->val << endl;  tmp=tmp->next;  }  pListHead = pListHead->next;  }  if(idx < k ){  return NULL;  }  return tmp;  }  }; |

# 反转链表



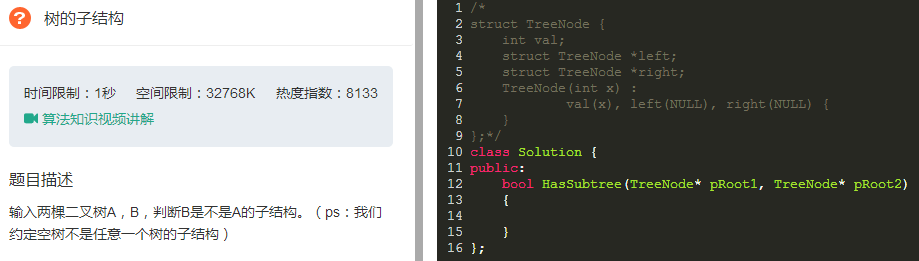
|  |
| --- |
| /\*  struct ListNode {  int val;  struct ListNode \*next;  ListNode(int x) :  val(x), next(NULL) {  }  };\*/  class Solution {  public:  ListNode\* ReverseList(ListNode\* pHead) {  ListNode \*newHead,\*tmp;  newHead = NULL;  while(pHead != NULL){  //从头到尾依次摘取结点  tmp = pHead ;  pHead = pHead->next;  if(newHead == NULL)  {  newHead = tmp ;  newHead->next = NULL ;  }  else  {  tmp->next = newHead ;  newHead = tmp ;  }  }  return newHead ;  }  }; |

# 合并两个排序链表



|  |
| --- |
| /\*  struct ListNode {  int val;  struct ListNode \*next;  ListNode(int x) :  val(x), next(NULL) {  }  };\*/  class Solution {  public:  ListNode\* Merge(ListNode\* pHead1, ListNode\* pHead2)  {  ListNode \*newHead = NULL ,\*tmp ,\*tail;  if(pHead1 == NULL && pHead2 == NULL) return NULL;  else if(pHead1 == NULL && pHead2 != NULL) return pHead2;  else if(pHead1 != NULL && pHead2 == NULL) return pHead1;    while(pHead1 != NULL && pHead2 != NULL){  if(pHead1->val > pHead2->val){  tmp = pHead2 ;  pHead2 = pHead2->next ;  }else{  tmp = pHead1 ;  pHead1 = pHead1->next ;  }    tmp->next = NULL;    if(newHead == NULL){  newHead = tmp ;  newHead->next = NULL;  tail = newHead ;  }else{  tail->next = tmp ;  tail = tail->next ;  }  }  //最多只剩一个不为空  if(pHead1 != NULL)  tail->next = pHead1 ;  if(pHead2 != NULL)  tail->next = pHead2 ;  return newHead;  }  }; |

# 树的子结构



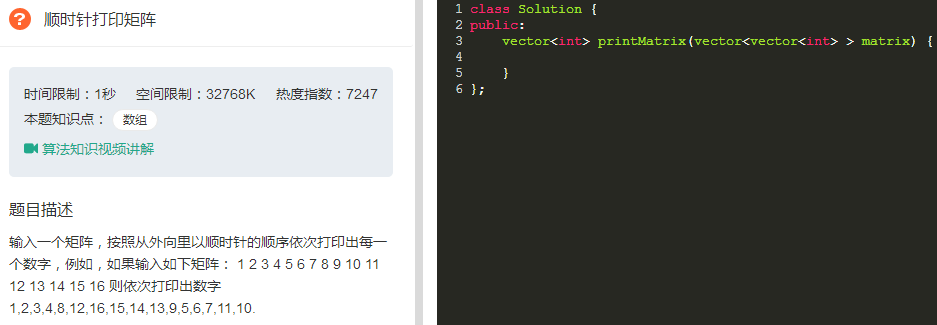
|  |
| --- |
| /\*  struct TreeNode {  int val;  struct TreeNode \*left;  struct TreeNode \*right;  TreeNode(int x) :  val(x), left(NULL), right(NULL) {  }  };\*/  class Solution {  public:  bool HasSubtree(TreeNode\* pRoot1, TreeNode\* pRoot2)  {  bool result = false;  if(pRoot1 != NULL && pRoot2 != NULL)  {  if(pRoot1->val == pRoot2->val)  result = DoesTree1HaveTree2(pRoot1,pRoot2);  if(!result)  result = DoesTree1HaveTree2(pRoot1->left,pRoot2);  if(!result)  result = DoesTree1HaveTree2(pRoot1->right,pRoot2);  }  return result;  }    bool DoesTree1HaveTree2(TreeNode\* pRoot1,TreeNode\* pRoot2)  {  if(pRoot2 == NULL) return true;  if(pRoot1 == NULL) return false;  if(pRoot1->val != pRoot2->val) return false;  return DoesTree1HaveTree2(pRoot1->left,pRoot2->left)&&DoesTree1HaveTree2(pRoot1->right,pRoot2->right);  }  }; |

# 镜像二叉树



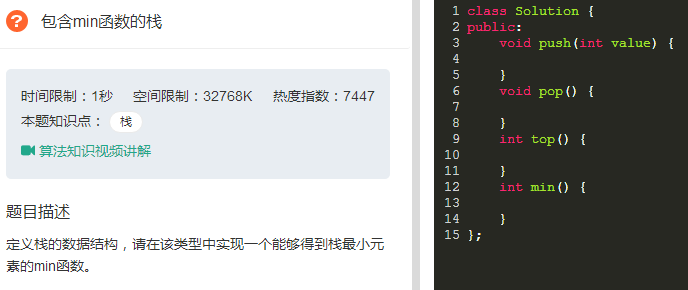
|  |
| --- |
| /\*  struct TreeNode {  int val;  struct TreeNode \*left;  struct TreeNode \*right;  TreeNode(int x) :  val(x), left(NULL), right(NULL) {  }  };\*/  class Solution {  public:  void Mirror(TreeNode \*pRoot) {  if(pRoot == NULL) return ;  else if(pRoot->left == NULL && pRoot->right == NULL){  return;  }    TreeNode \*pTemp = pRoot->left ;  pRoot->left = pRoot->right;  pRoot->right = pTemp;  if(pRoot->left)  Mirror(pRoot->left);  if(pRoot->right)  Mirror(pRoot->right);  }  }; |

# 顺时针打印矩阵



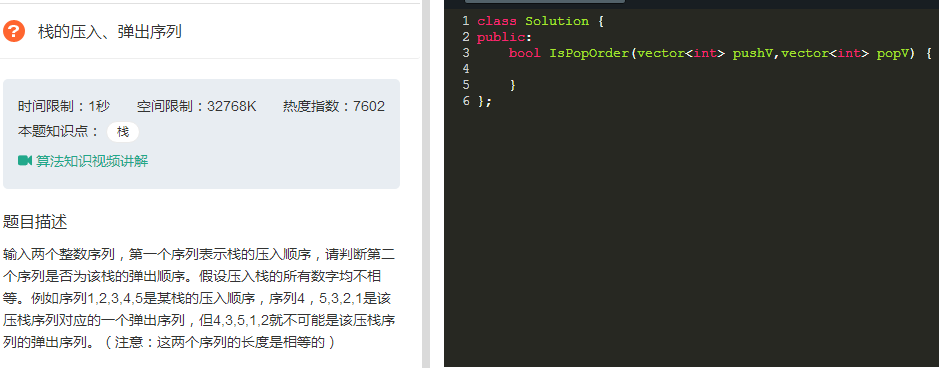
|  |
| --- |
| class Solution {  public:  vector<int> printMatrix(vector<vector<int> > matrix) {  vector<int> vec\_it;  if(matrix.size()<=0||matrix[0].size()<=0)  return vec\_it;  else  {  int columns = matrix[0].size(),rows = matrix.size();  int start = 0;  while(columns > start \* 2 && rows > start \* 2)  {  PrintMatrixInCricle(vec\_it,matrix,columns,rows,start);  ++start;  }  return vec\_it;  }  }  void PrintMatrixInCricle(vector<int> &vec,vector<vector<int> > matrix,int columns,int rows,int start)  {  //  int endX = columns - start - 1;  int endY = rows - start - 1;  for(int i = start ; i <= endX ; ++i)  vec.push\_back(matrix[start][i]);// -->  if(start < endY)  for(int i = start + 1 ;i <= endY;++i)  vec.push\_back(matrix[i][endX]);//  if(start < endY && start < endX)  for(int i = endX - 1 ;i >= start ; --i)  vec.push\_back(matrix[endY][i]);// -->  if(start < endY - 1 && start < endX)  for(int i = endY - 1 ;i >= start + 1 ; --i)  vec.push\_back(matrix[i][start]);// -->  }  }; |

# 包含min函数的栈



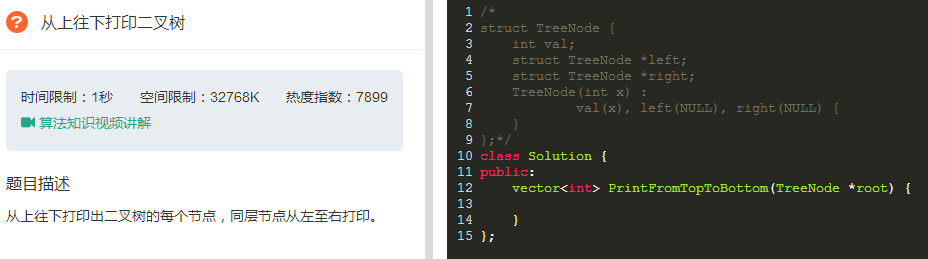
|  |
| --- |
| class Solution {  private:    public:  stack<int> m\_data,m\_min;  void push(int value) {  m\_data.push(value);    if(m\_min.size()==0||value < m\_min.top())  m\_min.push(value);  else  m\_min.push(m\_min.top());  }  void pop() {  //assert(m\_data.size()>0 && m\_min.size()>0);  m\_data.pop();  m\_min.pop();  }  int top() {  //assert(m\_data.size()>0);  return m\_data.top();  }  int min() {  //assert(m\_data.size()>0 && m\_min.size()>0);  return m\_min.top();  }  }; |

# 栈的压入、弹出序列



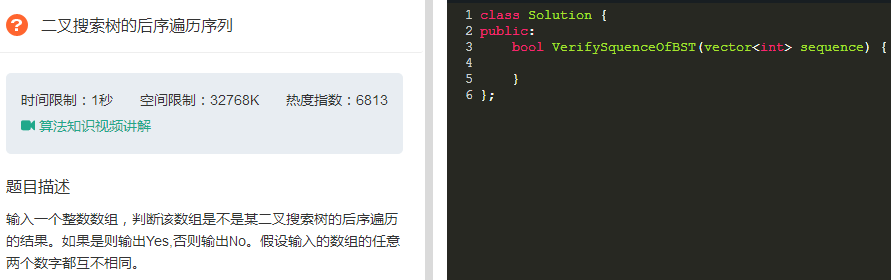
|  |  |
| --- | --- |
| |  | | --- | | class Solution {  public:      bool IsPopOrder(vector<int> pushV,vector<int> popV) {          int ipop=0,popLength=popV.size(),ipush=0,pushLength=pushV.size();          if(pushV.size()==0||popV.size()==0||popLength!=pushLength) return false;          bool isPossiable = false;          stack<int> S;          while(ipop < popLength){              while(S.empty()||popV[ipop]!=S.top()){                  if(ipush == pushLength) break;                  S.push(pushV[ipush++]);              }              if(S.top()!=popV[ipop]) break;              S.pop();              ++ipop;          }          if(ipop==popLength&&S.empty())              isPossiable=true;          return isPossiable;      }  }; | |

# 从上往下打印二叉树



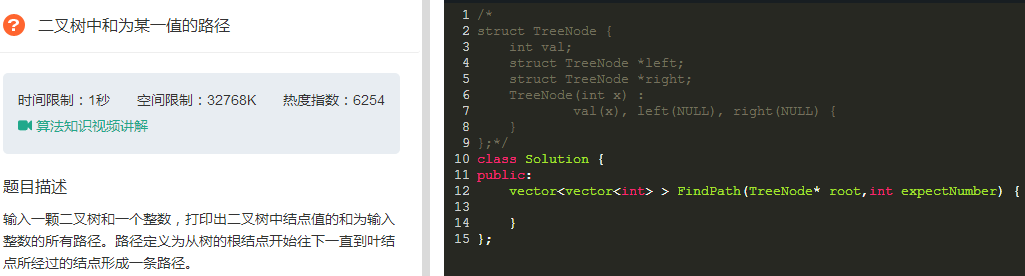
|  |
| --- |
| class Solution {  public:  vector<int> PrintFromTopToBottom(TreeNode \*root) {  vector<int> vec\_it;  if(!root)  return vec\_it;  std::deque<TreeNode \*> dequeTreeNode;  dequeTreeNode.push\_back(root);    while(dequeTreeNode.size()){  TreeNode \*pNode = dequeTreeNode.front();  dequeTreeNode.pop\_front();  vec\_it.push\_back(pNode->val);    if(pNode->left)  dequeTreeNode.push\_back(pNode->left);  if(pNode->right)  dequeTreeNode.push\_back(pNode->right);  }  return vec\_it;  }  }; |

# 二叉搜索树的后序遍历序列



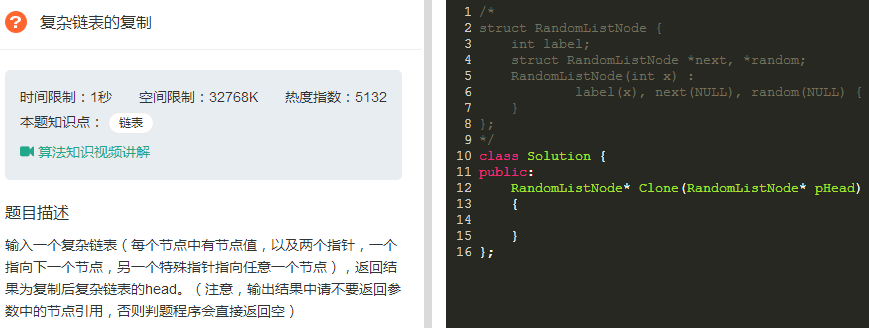
|  |
| --- |
| class Solution {  public:      bool VerifySquenceOfBST(vector<int> sequence) {          if(sequence.size()==0) return false;          return Verify(sequence,0,sequence.size());      }      bool Verify(vector<int> sequence,int start,int length){          int root = sequence[length - 1];          //左子树          int i = start;          for(;i < length - 1;++ i){              if(sequence[i] > root)                  break;          }          int j = i;          for(;j < length - 1; ++ j){              if(sequence[j] < root){                  return false;              }          }          bool left = true;          if(i > 0)              left = Verify(sequence,0,i);          bool right = true;          if(i < length - 1)              right = Verify(sequence,i,length - 1);            return left&&right;      }  }; |

# 二叉树中和为某一值的路径



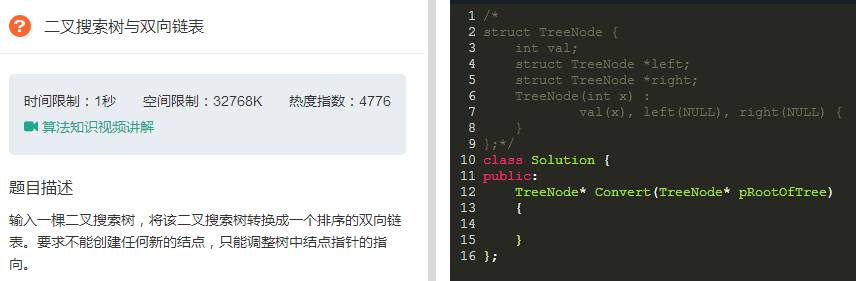
|  |
| --- |
| class Solution {  public:      vector<vector<int> > FindPath(TreeNode\* pRoot,int expectNumber) {          vector<vector<int> > vec\_ii;          if(pRoot == NULL) return vec\_ii;            std::vector<int> path;          int currentSum = 0 ;          FindPath(pRoot,expectNumber,path,currentSum,vec\_ii);          return vec\_ii;      }      void FindPath(TreeNode\* pRoot,int expectedSum,std::vector<int>& path,int &currentSum,vector<vector<int> >& vec\_ii){          currentSum += pRoot->val;          path.push\_back(pRoot->val);          //bool          bool isLeaf = pRoot->left == NULL && pRoot->right == NULL;          if(isLeaf && currentSum == expectedSum)              vec\_ii.push\_back(path);            if(pRoot->left != NULL) FindPath(pRoot->left,expectedSum,path,currentSum,vec\_ii);          if(pRoot->right != NULL) FindPath(pRoot->right,expectedSum,path,currentSum,vec\_ii);            currentSum -= pRoot->val;          path.pop\_back();      }  }; |

# 复杂链表的复制



|  |
| --- |
| class Solution {  public:      RandomListNode\* Clone(RandomListNode\* pHead)      {          CloneNodes( pHead );          ConnectSiblingNodes( pHead );          return ReconnectNodes( pHead );      }      void CloneNodes(RandomListNode\* pHead)      {          RandomListNode\* pNode = pHead;          while(pNode != NULL)          {              RandomListNode \*pCloned = (RandomListNode \*)malloc(sizeof(RandomListNode));              pCloned->label = pNode->label;              pCloned->next = pNode->next;              pCloned->random = NULL;                pNode->next = pCloned;         //将新复制的结点链接在原始结点的后面              pNode = pCloned->next;          }      }      void ConnectSiblingNodes(RandomListNode\* pHead)      {          RandomListNode\* pNode = pHead;          while(pNode != NULL)                 //遍历链表更新随机指针          {              RandomListNode \*pCloned = pNode->next;              if(pNode->random != NULL)              {                  pCloned->random = pNode->random->next;       //新复制结点的随机指针就是原始结点的随机指针指向的结点的下一个结点              }              pNode = pCloned->next;          }      }      RandomListNode\* ReconnectNodes(RandomListNode\* pHead)      {          RandomListNode\* pNode = pHead;          RandomListNode\* pClonedHead = NULL;          RandomListNode\* pClonedNode = NULL;          if(pNode != NULL)          {              pClonedHead = pClonedNode = pNode->next;              pNode->next = pClonedNode->next;              pNode = pNode->next;          }          while(pNode != NULL)          {              pClonedNode->next = pNode->next;   //把偶数位置的结点链接起来就是复制出来的新链表              pClonedNode = pClonedNode->next;              pNode->next = pClonedNode->next;   //把奇数位置的结点链接起来就是原始链表              pNode = pNode->next;          }          return pClonedHead;      }  }; |

# 二叉搜索树与双向链表



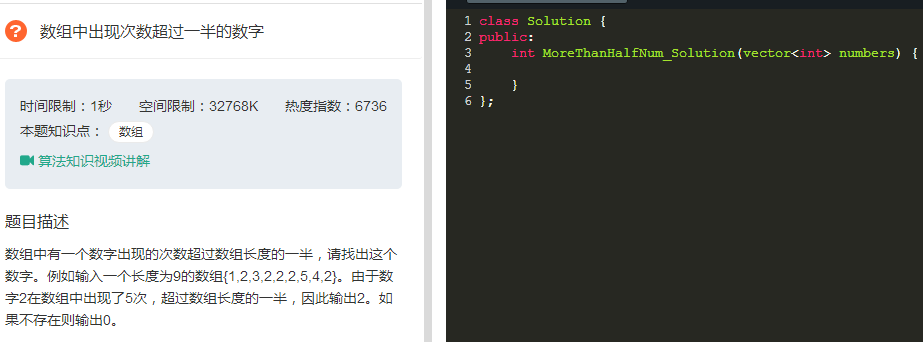
|  |
| --- |
|  |

# 字符串的排列



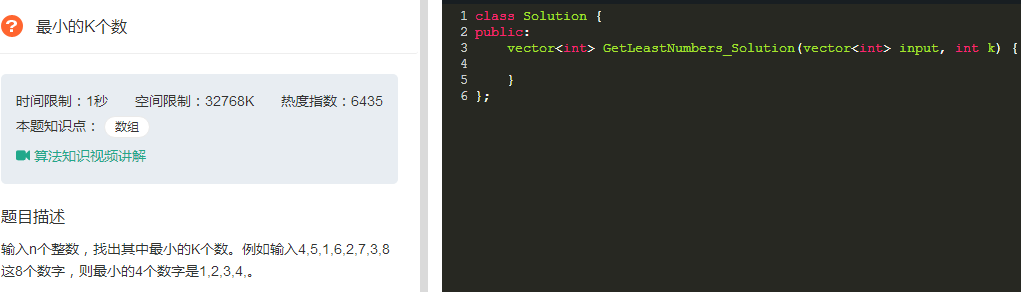
|  |
| --- |
|  |

# 数组中出现次数超过一半的数字



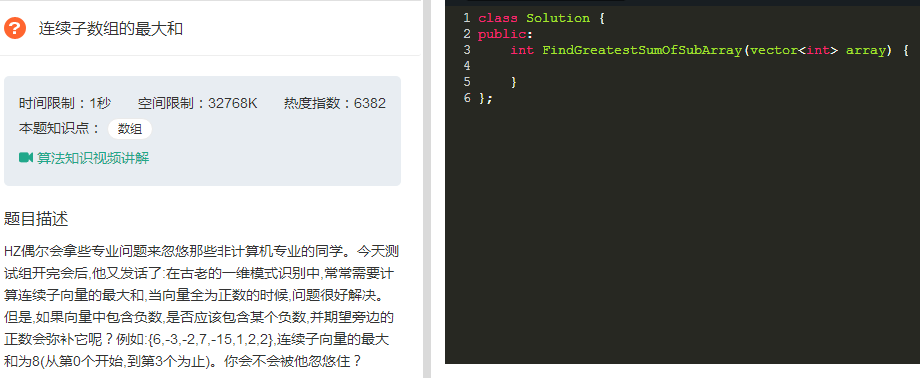
|  |
| --- |
| class Solution {  public:  int MoreThanHalfNum\_Solution(vector<int> numbers) {  int sz = numbers.size();  if(0 == sz) return 0;  int search\_val,nTimes,i;  for(i = nTimes = 0 ;i < sz ; ++i)  {  if(0 == nTimes)  {  search\_val = numbers[i];  nTimes = 1;  }  else  {  if(search\_val == numbers[i])  ++nTimes;  else  --nTimes;  }  }  nTimes = 0 ;  for(i = 0 ;i < sz ; ++i)  {  if(search\_val == numbers[i])  ++nTimes;  }  if(nTimes \* 2 < sz)  {  return 0;  }  else  {  return search\_val ;  }  }  }; |

# 最小的K个数



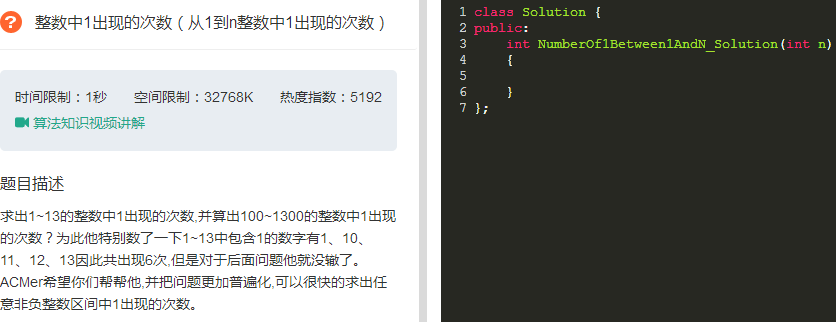
|  |
| --- |
| class Solution {  public:  int Partition(vector<int> &data,int len,int start,int end)  {  if(data.size() < 0 || start < 0 || end >= len)  return 0;  int small = start - 1,idx;  for(idx = start ; idx < end ; ++idx)  {  if(data[idx]<data[end])  {  ++small;  if(small!=idx) swap(data[idx],data[small]);  }  }  ++small;  swap(data[small],data[end]);  return small;  }  vector<int> GetLeastNumbers\_Solution(vector<int> input, int k) {  vector<int> vec\_it;  int n = input.size() ;  if(n <= 0 || k<= 0 || n < k)  {  vec\_it.clear();  return vec\_it;  }  int start = 0,end = n - 1;  int idx = Partition(input,n,start,end);  while(idx != k - 1)  {  if(idx > k - 1)  {  end = idx - 1;  idx = Partition(input,n,start,end);  }  else  {  start = idx + 1;  idx = Partition(input,n,start,end);  }  }  for(idx = 0 ; idx < k;++idx)  vec\_it.push\_back(input[idx]);  return vec\_it;  }  }; |

# 连续子数组的最大和



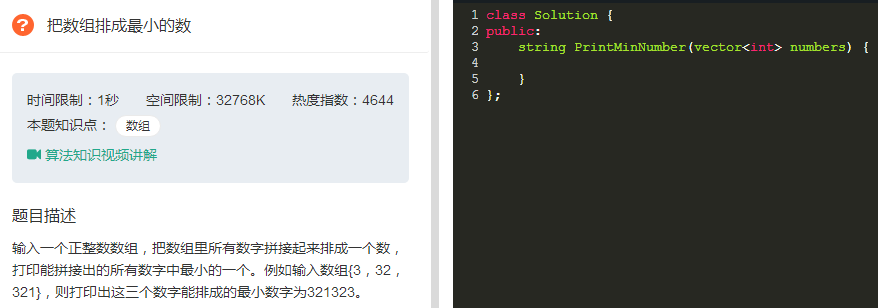
|  |
| --- |
| class Solution {  public:  int FindGreatestSumOfSubArray(vector<int> array) {  int sz = array.size();  if(sz <= 0) return 0;  int endmax = array[0];  int answer = array[0];  for(int i = 1 ;i < sz ; ++i){  endmax = max(endmax,0) + array[i];  answer = max(answer,endmax);  }  return answer;  }  }; |

# 整数中1出现的次数（从1到n整数中1出现的次数）



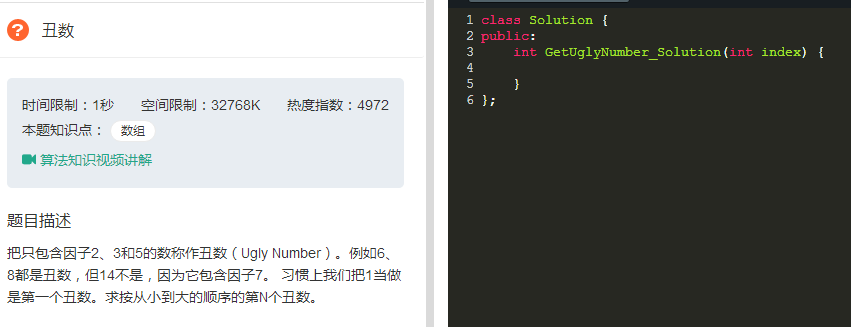
|  |
| --- |
|  |

# 把数组排成最小的数



|  |
| --- |
|  |

# 丑数



|  |
| --- |
| class Solution {  public:  int GetUglyNumber\_Solution(int index) {  if(index <= 0)  return 0;    int \*pUglyNumbers = new int[index];  pUglyNumbers[0] = 1;  int nextUglyIndex = 1;  int \*pMultiply2 = pUglyNumbers;  int \*pMultiply3 = pUglyNumbers;  int \*pMultiply5 = pUglyNumbers;  while(nextUglyIndex < index){  int minnum = min(\*pMultiply2 \* 2,min(\*pMultiply3 \* 3,\*pMultiply5 \* 5));  pUglyNumbers[nextUglyIndex] = minnum;    while(\*pMultiply2 \* 2 <= pUglyNumbers[nextUglyIndex])  ++pMultiply2;  while(\*pMultiply3 \* 3 <= pUglyNumbers[nextUglyIndex])  ++pMultiply3;  while(\*pMultiply5 \* 5 <= pUglyNumbers[nextUglyIndex])  ++pMultiply5;  ++nextUglyIndex;  }  int ugly = pUglyNumbers[nextUglyIndex - 1];  delete[] pUglyNumbers;    return ugly;  }  }; |

|  |
| --- |
|  |

|  |
| --- |
|  |