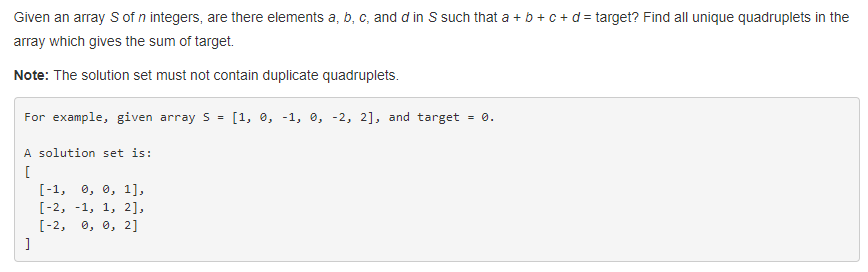
**LeetCode习题**

**18. 4Sum**



**C++**

解法一、

根据3Sum解法在外层继续加一层循环，通过去除最外面的数将4Sum转化为3Sum问题，最后遍历最外面的数。

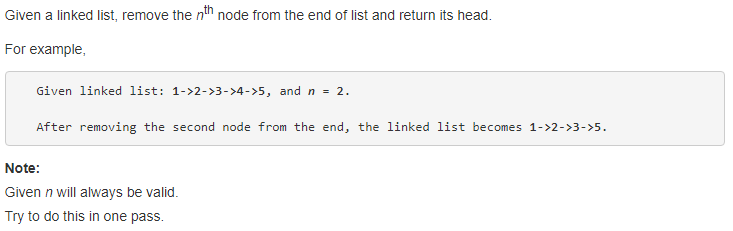
|  |
| --- |
| class Solution {  public:  vector<vector<int>> fourSum(vector<int>& nums, int target) {  set<vector<int> > res;  vector<vector<int> > ret;  if(nums.size()<4) return ret;  sort(nums.begin(), nums.end());  for(int i=0;i<nums.size()-3;i++)  {  for(int j=i+1;j<nums.size()-2;j++)  {  int left = j+1;  int right = nums.size()-1;  while(left<right)  {  int sum = nums[i] + nums[j] + nums[left] + nums[right];  if(sum == target)  {  vector<int> tmp;  tmp.push\_back(nums[i]);  tmp.push\_back(nums[j]);  tmp.push\_back(nums[left]);  tmp.push\_back(nums[right]);  res.insert(tmp);  ++left;  --right;  }  else if(sum < target) ++left;  else --right;  }  }  }  return vector<vector<int> > (res.begin(), res.end());  }  }; |

解法二、

枚举出所有两个数的和存放在hashmap中，其中map的key对应是两个数的和，map的value是一个链表，链表的每个节点存的是这两个数在数组中的下标。

|  |
| --- |
| class Solution {  public:  vector<vector<int>> fourSum(vector<int>& nums, int target) {  int n = nums.size();  vector<vector<int> > res;  if(n < 4) return res;  sort(nums.begin(), nums.end());  unordered\_map<int, vector<pair<int, int> > > pairs;    //给vector预分配存储区大小，但是没有给这段内存进行初始化，实际分配的空间可能大于或等于这个值。  pairs.reserve(n\*n);    //将n个数字两两组合求和存进map中，相同和的对在map中按照链表存储，即存在vector中，j从i+1开始  for(int i=0;i<n;i++)  {  for(int j=i+1;j<n;j++)  {  pairs[nums[i]+nums[j]].push\_back(make\_pair(i,j));  }  }    for(int i=0;i<n-3;i++)  {  //去重  if(i>0 && nums[i] == nums[i-1]) continue;  for(int j=i+1;j<n-2;j++)  {  //去重  if(j>i+1 && nums[j] == nums[j-1]) continue;  if(pairs.find(target-nums[i]-nums[j]) != pairs.end())  {  vector<pair<int, int> > sum2 = pairs[target - nums[i] - nums[j]];  bool first\_push = true;    //将符合要求且不重复的pair存进vector中  for(int k=0;k<sum2.size();k++)  {  if(sum2[k].first <= j) continue;  //第一次存储的重复数字，或者不是重复数字判断第三位是否相同  if(first\_push || (res.back())[2] != nums[sum2[k].first])  {  vector<int> tmp{nums[i], nums[j], nums[sum2[k].first], nums[sum2[k].second]};  res.push\_back(tmp);  first\_push = false;  }  }  }  }  }  return res;  }  }; |

**19. Remove Nth Node From End of List**



**C++**

解法一、

因为是删除倒数第n个位置的数字，可以先求出链表的长度，进而求出删除节点的位置，最后找到节点删除。

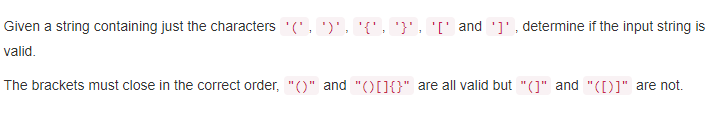
|  |
| --- |
| /\*\*  \* Definition for singly-linked list.  \* struct ListNode {  \* int val;  \* ListNode \*next;  \* ListNode(int x) : val(x), next(NULL) {}  \* };  \*/  class Solution {  public:  ListNode\* removeNthFromEnd(ListNode\* head, int n) {  if(n==0) return head;  int length = 0;  ListNode\* dummy = head;  //遍历链表，计算长度  while(dummy != NULL)  {  dummy = dummy->next;  length++;  }  if(length == n)  {  ListNode\* res = head->next;  delete head;  return res;  }  else{  //计算出删除节点的位置  int loc = length - n - 1;  dummy = head;  while(loc--)  {  dummy = dummy->next;  }  ListNode\* res = dummy->next;  dummy->next = dummy->next->next;  delete res;  return head;  }  }  }; |

解法二、

使用两个指针，第一个先走N步，第二个从头开始，两个一起走，当第一个指针到尾部时，第二个指针正好指向要删除的节点。

|  |
| --- |
| /\*\*  \* Definition for singly-linked list.  \* struct ListNode {  \* int val;  \* ListNode \*next;  \* ListNode(int x) : val(x), next(NULL) {}  \* };  \*/  class Solution {  public:  ListNode\* removeNthFromEnd(ListNode\* head, int n) {  if(head == NULL) return NULL;  //用于记录删除节点的前一个节点  ListNode\* pre = NULL;  ListNode\* p = head;  ListNode\* q = head;  //先走n-1步  for(int i=0;i<n-1;i++)  {  q = q->next;  }  while(q->next)  {  pre = p;  p = p->next;  q = q->next;  }  //如果n大于等于链表长度  if(pre == NULL)  {  head = p->next;  delete p;  }  else{  pre->next = p->next;  delete p;  }  return head;  }  }; |

**20. Valid Parentheses**



**C++**

解法一、

通过压栈的方法来解决。对于字符串中每个字符C，如果C是左括号，就亚入栈中，其余情况就和栈进行比对。如果栈空了，则说明这是无效的。如果没有空，则比较栈顶元素，栈顶是“（”，而当前元素是“）”，则出栈；栈顶是“[”，当前元素是“]”，则出栈；栈顶是“{”，当前是“}”，则出栈。其余情况都是无效。最终看栈里是否为空。

|  |
| --- |
| class Solution {  public:  bool isValid(string s) {  stack<char> stk;  int len = s.length();  for(int i=0;i<len;i++)  {  //将左括号压入栈中  if(s[i]=='(' || s[i]=='[' || s[i]=='{')  {  stk.push(s[i]);  }  else{  //若栈已空，则不匹配  if(stk.empty()) return false;  if(stk.top()=='(' && s[i]==')') stk.pop();  else if(stk.top()=='[' && s[i]==']') stk.pop();  else if(stk.top()=='{' && s[i]=='}') stk.pop();  else return false;  }  }  return stk.empty();  }  }; |

**21. Merge Two Sorted Lists**



**C++**

解法一、

先比较两个链表第一个数字，谁小放在新链表的开头。接着同时遍历两个链表，每次都取小的那个，最后哪个链表还剩把剩余的都插入到新链表中。

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

解法二、

递归法，调用自己，直到两个链表都为空。

|  |
| --- |
| /\*\*  \* Definition for singly-linked list.  \* struct ListNode {  \* int val;  \* ListNode \*next;  \* ListNode(int x) : val(x), next(NULL) {}  \* };  \*/  class Solution {  public:  ListNode\* mergeTwoLists(ListNode\* l1, ListNode\* l2) {  if(l1 == NULL) return l2;  if(l2 == NULL) return l1;  if(l1->val < l2->val)  {  l1->next = mergeTwoLists(l1->next, l2);  return l1;  }  else{  l2->next = mergeTwoLists(l2->next, l1);  return l2;  }  }  }; |

**23. Merge k Sorted Lists**



**C++**

解法一、

利用二分思想，把第一个和第n/2+1个list合并，第2个和第n/2+2个合并…，最终合并成一个list。

|  |
| --- |
| /\*\*  \* Definition for singly-linked list.  \* struct ListNode {  \* int val;  \* ListNode \*next;  \* ListNode(int x) : val(x), next(NULL) {}  \* };  \*/  class Solution {  public:  ListNode\* mergeKLists(vector<ListNode\*>& lists) {  int sz = lists.size();  if(sz == 0) return NULL;    //二分合并  while(sz>1)  {  int k = (sz+1)/2;  for(int i=0;i<sz/2;i++)  {  lists[i] = mergeTwoLists(lists[i],lists[i+k]);  }  sz = k;  }  return lists[0];  }    //合并两个排序的链表  ListNode\* mergeTwoLists(ListNode\* l1, ListNode\* l2)  {  if(!l1) return l2;  if(!l2) return l1;    ListNode\* start;  ListNode\* cur;  if(l1->val < l2->val)  {  start = cur = l1;  l1 = l1->next;  }  else{  start = cur = l2;  l2 = l2->next;  }  while(l1 != NULL && l2 != NULL)  {  if(l1->val < l2->val)  {  cur->next = l1;  cur = l1;  l1 = l1->next;  }  else{  cur->next = l2;  cur = l2;  l2 = l2->next;  }  }  if(l1 != NULL)  {  cur->next = l1;  }  else{  cur->next = l2;  }  return start;  }  }; |