编程辑略

sosohu

Email: ustc.sosohu@gmail.com
Blog: sosohu.github.io

前言

背景

说到程序员面试中的算法题面试，我们不得不去做很多准备，最简单粗暴的做法就是去刷题，本书也就是作者在知名刷题网站leetcode刷题过程中针对各个算法题的解法总结以及一些基本知识的补充，由于作者水平有限，著作此书仅为交流，不具备任何权威性，有什么不正确的地方敬请指教.

目标读者和假设

本书假定读者熟悉基本的数据结构,比如熟悉数组，链表，树，图，堆，栈，队列和图等概念并且对各种数据结构的特性有着深刻的认识. 本书假定读者熟悉算法导论上的一些常用和经典算法，比如各种排序算法，树和图的经典算法，字符串匹配，动态规划，贪心，回溯，分治等常见算法策略．由于本书的代码都是由C++编写，所以可能还需要读者能够大致了解C++的一些语法和库以免读代码阅读带来困扰.

本书的内容

本书的主要内容分为两部分，第一部分也就是前八章是从数据结构的视角来看到问题，后面的九章则是从算法的角度来看待问题

本书的代码

本书的代码都是采用C++编写，遵守C++ 11标准，所有代码都是通过-O3 -std=c++0x编译选项使用g++编译运行过.

对于所有的出现在leetcode上的问题的解答代码都是在leetcode上accept过,完整解答源码可以访问作者的github查看; 而其他代码则不能保证一定是没有问题的.

本书的编写

本书采用latex编写，编写的模板是作者自己定义的，所以比较简陋，希望能够多加包涵.

关于作者

本书作者是sosohu, 相关联系方式如下，欢迎与我联系

- **Email**: ustc.sosohu@gmail.com
- **Blog**: http://sosohu.github.io/
- **Github**: https://github.com/sosohu

致谢

感谢leetcode网站提供了如此便捷的刷题机会.感谢soulmachine的leetcode题解，这是一份非常好的题解，作者水平也很高.感谢同实验室的一起刷题的zhuoyuan同学，CarlSama同学，cjneo同学和pmon同学.

目录

第 1 章　链表

1.1　链表的基本操作

链表的基本操作大致包括插入，删除，反转等，这些基本操作是链表问题最核心的部分，很多链表问题都需要这三种操作的某种或者某几种.

1.1.1　InsertNode

问题: Given a list-node pointer, insert a new node after the node that the pointer to.

Care : 时间复杂度O(1), 空间复杂度O(1)

```
1  void insertNode(ListNode *pPos, ListNode *pInsert){
2      if(!pPos || !pInsert)   return;
3      pInsert->next = pPos->next;
4      pPos->next = pInsert;
5  }
```

问题: Given a list-node pointer, insert a new node before the node that the pointer to.

Trick : 时间复杂度O(1), 空间复杂度O(1) 这里给出节点指针，要求插在该节点之前，我们可以先插到该节点之后再交换两节点的值，即完成插入.

```
1  void insertNode(ListNode *pPos, ListNode *pInsert){
2      if(!pPos || !pInsert)   return;
3      pInsert->next = pPos->next;
4      pPos->next = pInsert;
5      int tmp = pPos->val;
6      pPos->val = pInsert->val;
7      pInsert->val = tmp;
8  }
```

1.1.2　DeleteNode

问题: Given a list-node pointer, delete the node after the node that the pointer to.

Care : 时间复杂度O(1), 空间复杂度O(1)

```
1  void deleteNode(ListNode *pDelete){
2      if(!pDelete || !pDelete->next)  return;
3      ListNode* cur = pDelete->next;
4      pDelete->next = pDelete->next->next;
5      delete cur;
6  }
```

问题: Given a list-node pointer, delete the node that the pointer to.

Trick：时间复杂度O(1), 空间复杂度O(1) 这里给出节点指针，要求删除该节点之前，我们可以先将后序节点的值拷过来，再删除后序节点，当然，如果当前节点是尾节点则必须从链表头部开始遍历然后删除了.

```
void deleteNode(ListNode *pHead, ListNode *pDelete){
    if(!pHead || !pDelete)  return;
    if(pDelete->next){
        //不是最后一个节点，只需要复制下一个节点的值
        //过来然后删除下一个节点，O(1)
        ListNode *tmp = pDelete->next;
        pDelete->val = pDelete->next->val;
        pDelete->next = pDelete->next->next;
        delete tmp;  // 很容易忘记
    }else{
        //是最后一个节点，只能从前遍历到倒数第二个节点
        if(pHead == pDelete){
            delete pHead;
            pHead = NULL;
        }
        ListNode *pos = pHead;
        while(pos->next != pDelete){
            pos = pos->next;
        }
        pos->next = NULL;
        delete pDelete;   // 这一句很容易忘记
    }
}
```

虽然，当节点是尾节点首需要O(n)次操作，但是平均下来时间复杂度是O(1)

1.1.3 RGetKthNode

问题: Given a list, return the last Kth Node.

迭代：时间复杂度O(n), 空间复杂度O(1) 两个指针问题.

```
ListNode * RGetKthNode(ListNode *pHead, unsigned int k){
    if(k < 1)   return NULL;
    ListNode *pre = pHead;
    ListNode *last = pHead;
    while(k-- > 0 && last){
        last = last->next;
    }
    if(!last) return NULL;
    while(last){
        last = last->next;
        pre = pre->next;
    }
    return pre;
}
```

1.1.4 ReverseList

问题: Reverse a singly linked list. *(leetcode 206)*

头插法：时间复杂度O(n)，空间复杂度O(1)

新建一个头节点，然后依次将后面节点插入到头节点之后，便形成链表的倒序.

```
ListNode *ReverseList(ListNode *pHead){
    if(!pHead || !pHead->next)  return pHead;
    ListNode * newhead = new ListNode(0);
    ListNode *pos = pHead, *tmp;
    newhead->next = pHead;
    while(pos->next){
        tmp = pos->next;
        pos->next = tmp->next;
        tmp->next = newhead->next;
        newhead->next = tmp;
    }
    return newhead->next;
}
```

就地反转：时间复杂度O(n)，空间复杂度O(1)

所谓就地反转就是每次把i指向i+1直接转为i指向i+1即可.

```
ListNode *ReverseList(ListNode *pHead){
    if(!pHead || !pHead->next)  return pHead;
    ListNode *pre = pHead, *cur = pHead->next, *next = NULL;
    pHead->next = NULL;
    while(cur){
        next = cur->next;
        cur->next = pre;
        pre = cur;
        cur = next;
    }
    return pre;
}
```

递归：时间复杂度O(n)，空间复杂度O(1)

链表其实就是一个二叉树，所以可以套用二叉树的递归做法.

```
ListNode *ReverseList(ListNode *pHead){
    if(!pHead || !pHead->next)  return pHead;
    // 返回为段倒转后的首节点leftleft
    ListNode *left = ReverseList(pHead->next);
    // pHead->next 一开始是段的首节点，倒转后就是末节点left
    pHead->next->next = pHead;
    pHead->next = NULL;
    return left;
}
```

1.2 链表的基本问题

链表的基本问题大多都是很经典的链表问题，例如中间节点，是否有环等等.

1.2.1 GetMidNode

问题: Given a list, return the middle node.

快慢指针：时间复杂度O(n)，空间复杂度O(1)

慢指针以每步一个节点的速度前进，快指针以每步两个节点的速度前进

```
ListNode* getMidNode(ListNode *pHead){
    if(!pHead || !pHead->next)  return pHead;
    ListNode *slow = pHead, *fast = pHead->next;
    while(fast && fast->next){
        slow = slow->next;
        fast = fast->next;
        fast = fast->next;
    }
    return slow;
}
```

1.2.2 IsMeetList

问题: Given two lists, return wheather the two lists meet.

Null：时间复杂度(n+m), 空间复杂度O(1)

判断两个链表是否相交，只需要判断两个链表最后一个节点是否为同一个即可

```
bool isMeetList(ListNode *pHead1, ListNode *pHead2){
    if(!pHead1 || !pHead2)  return false;
    ListNode *pos1 = pHead1, *pos2 = pHead2;
    while(pos1->next){
        pos1 = pos1->next;
    }
    while(pos2->next){
        pos2 = pos2->next;
    }
    return pos1 == pos2;
}
```

1.2.3 Intersection of Two Linked Lists

问题:

Write a program to find the node at which the intersection of two singly linked lists begins.
leetcode 160

Note:

Write a program to find the node at which the intersection of two singly linked lists begins.
If the two linked lists have no intersection at all, return null.
The linked lists must retain their original structure after the function returns.
You may assume there are no cycles anywhere in the entire linked structure.
Your code should preferably run in O(n) time and use only O(1) memory.

双指针：时间复杂度(n+m), 空间复杂度O(1)

找到两个链表第一个相交点，首先遍历两个链表，得到两个链表的长度，比如说l1, l2.然后先移动较长的那个链表（比如l1¿l2），移动l1-l2步，这样双方剩余的节点数就相等了.接着以前往后走，第一次相遇的点就是答案

```
ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
    if(!headA || !headB)    return NULL;
    int la = 1, lb = 1;
    ListNode *posA = headA, *posB = headB;
    while(posA->next || posB->next){
        if(posA->next){
            la++;
            posA = posA->next;
        }
        if(posB->next){
            lb++;
            posB = posB->next;
        }
    }
    if(posA != posB)    return NULL;
    if(lb > la){
        swap(headA, headB);
        swap(la, lb);
    }
    for(auto i = 0; i < la -lb; i++)
        headA = headA->next;
    while(headA != headB){
        headA = headA->next;
        headB = headB->next;
    }
    return headA;
}
```

1.2.4 IsCircleList

问题: Given a linked list, determine if it has a cycle in it.(*leetcode 141*)

快慢指针：时间复杂度O(n) , 空间复杂度O(1)

快慢指针, 慢指针每步一个节点，快指针每步两个节点，如果有环肯定会相遇

```
bool IsCircleList(ListNode *pHead){
    if(!pHead || !pHead->next) return false;
    ListNode *slow = pHead, *fast = pHead;
    while(fast && fast->next){
        slow = slow->next;
        fast = fast->next->next;
        if(slow == fast)    return true;
    }
    return false;
}
```

1.2.5 GetFirstCircleNode

问题: Given a linked list, return the node where the cycle begins. If there is no cycle, return null. *(leetcode 142)*

快慢指针：时间复杂度O(n)，空间复杂度O(1)

```
ListNode *GetFirstCircleNode(ListNode* pHead){
    if(!pHead || !pHead->next)  return pHead;
    ListNode *slow = pHead, *fast = pHead;
    // 指针先进入环内fast
    while(fast && fast->next){
        slow = slow->next;
        fast = fast->next->next;
        if(slow == fast)
            break;
    }
    if(!fast || !fast->next)
        return NULL; // 不存在环
    // 指针再次从头开始slow，指针减速fast
    slow = pHead;
    while(slow != fast){
        slow = slow->next;
        fast = fast->next;
    }
    return slow; // 相遇点就是环的入口
}
```

1.2.6 MergeSortedList

问题: Merge two sorted lists.*(leetcode 21)*

迭代：时间复杂度(n), 空间复杂度O(1)

```
ListNode* MergeSortedList(ListNode *pHead1, ListNode *pHead2){
    if(!pHead1) return pHead2;
    if(!pHead2) return pHead1;
    ListNode *pos1 = pHead1, *pos2 = pHead2;
    ListNode *ret = pHead1->val < pHead2->val? (pos1 = pos1->next,
        pHead1)
                  : (pos2 = pos2->next, pHead2);
    ListNode *pos = ret;
    while(pos1 && pos2){
        if(pos1->val < pos2->val){
            pos->next = pos1;
            pos1 = pos1->next;
        }else{
            pos->next = pos2;
            pos2 = pos2->next;
        }
        pos = pos->next;
    }
    if(pos1){
        pos->next = pos1;
    }
    if(pos2){
        pos->next = pos2;
    }
    return ret;
}
```

递归：时间复杂度(n), 空间复杂度O(1)

```
ListNode* MergeSortedList(ListNode *pHead1, ListNode *pHead2){
    if(!pHead1) return pHead2;
    if(!pHead2) return pHead1;
    ListNode *ret, *pos1 = pHead1, *pos2 = pHead2;
    ret = pHead1->val < pHead2->val?  (pos1 = pos1->next, pHead1) : (
        pos2 = pos2->next, pHead2);
    ret->next = MergeSortedList(pos1, pos2);
    return ret;
}
```

1.2.7 SortList

问题: Sort a linked list in O(n log n) time using constant space complexity. *(leetcode 148)*

迭代：时间复杂度O(nlgn)，空间复杂度O(1)

仿照SGI STL里面的List容器的sort函数，实现迭代版的归并排序

```
 ListNode *sortList(ListNode *pHead){
    if(!pHead || !pHead->next)  return pHead;
    vector<ListNode*> counter;
    for(int i = 0; i < 64; i++)
        counter.push_back(NULL);
    ListNode *carry;
    ListNode *pos = pHead;
    int fill = 0;
    while(pos){
        carry = new ListNode(pos->val);
        pos = pos->next;
        int i = 0;
        for(i = 0; i < fill && counter[i]; i++){
            carry = MergeSortedList(carry, counter[i]);
            counter[i] = NULL;
        }
        counter[i] = carry;
        if(i == fill) fill++;
    }
    for(int i = 1; i < fill; i++){
        counter[i] = MergeSortedList(counter[i-1], counter[i]);
    }
    return counter[fill-1];
}
```

这个迭代版的归并方法很有用，几乎所有的归并函数都可以这样改装

递归：时间复杂度O(nlgn)，空间复杂度O(lgn)

```
ListNode *ListSort(ListNode *pHead){
    if(!pHead || !pHead->next)  return pHead;
    ListNode *mid = getMidNode(pHead);
    ListNode *right = pHead, *left = mid->next;
    mid->next = NULL;
    right = ListSort(right);
    left = ListSort(left);
    pHead = MergeSortedList(right, left);
    return pHead;
}
```

1.2.8 Insertion Sort List

问题: Sort a linked list using insertion sort. *(leetcode 147)*

排序：时间复杂度O(n^2) , 空间复杂度O(1)

和数组的插入排序几乎一样,使用一个新的头节点更简单.

```
ListNode *insertionSortList(ListNode *head) {
    if(!head || !head->next)    return head;
    ListNode newhead(0);
    newhead.next = head;
    ListNode *pos = head, *begin = &newhead, *tmp;
    while(pos->next){
        begin = &newhead;
        while(begin != pos && begin->next->val <= pos->next->val){
            begin = begin->next;
        }
        if(begin == pos){
            pos = pos->next;
        }else{
            tmp = pos->next;
            pos->next = tmp->next;
            tmp->next = begin->next;
            begin->next = tmp;
        }
    }
    return newhead.next;
}
```

1.3 基于链表的问题

基于链表的问题，大多需要前面所说的基本操作和基本问题的组合，另外，对于链表这个容器的认识也要很深刻才行，链表容器的优势在于能够快速的插入和删除指定节点，而且节省空间，但是它的问题就是不能随机访问，可以说和vector是互补的.

1.3.1 Add Two Numbers

问题: You are given two linked lists representing two non-negative numbers. The digits are stored in reverse order and each of their nodes contain a single digit. Add the two numbers and return it as a linked list. *(leetcode 2)*

举例: Input: $(2 \to 4 \to 3) + (5 \to 6 \to 4)$, Output: $7 \to 0 \to 8$

Care : 时间复杂度O(n+m), 空间复杂度O(1)
这道题主要是要注意进位和一些边界条件判断的简化，这里有很多边界条件，怎么把很多if语句归纳到一起使代码更简洁.

```
ListNode *addTwoNumbers(ListNode *l1, ListNode *l2) {
    ListNode head(0), *ret = &head;
    int cin = 0, val;
    while(l1 || l2 || cin){
        val = (l1? l1->val : 0) + (l2? l2->val : 0) + cin;
        ret = ret->next = new ListNode(val % 10);
        cin = val / 10;
        if(l1)  l1 = l1->next;
        if(l2)  l2 = l2->next;
    }
    return head.next;
}
```

1.3.2 Remove Nth Node From End of List

问题: Given a linked list, remove the nth node from the end of list and return its head. *(leetcode 19)*

举例 : Given linked list: $1 \to 2 \to 3 \to 4 \to 5$, and n = 2. After removing the second node from the end, the linked list becomes $1 \to 2 \to 3 \to 5$.

Note : Given n will always be valid.

两个指针 : 时间复杂度O(n), 空间复杂度O(1)
两个指针相距n,然后一起后移.

```
ListNode *removeNthFromEnd(ListNode *head, int n) {
    if(!head || n < 1)  return head;
    ListNode newhead(0);
    newhead.next = head;
    ListNode *last = &newhead, *first = &newhead;
    while(n-- > 0 && last->next){
        last = last->next;
    }
    while(last->next){
        first = first->next;
        last = last->next;
    }
    first->next = first->next->next;
```

```
14      return newhead.next;
15  }
```

1.3.3 Merge k Sorted Lists

问题: Merge k sorted linked lists and return it as one sorted list. Analyze and describe its complexity. *(leetcode 23)*

: 时间复杂度，空间复杂度O

这道题其实是Merge two list的延伸，我们在Merge两个链表时候，每次都是比较一下取较小的入新链表，当有k个链表的时候，我们也是取这k个节点最小的那个，然后用它的next替换它，再进行下次选取.我们想一下，最naive的做法是，每次这k个都比较一遍,那么这个每取一个的时间复杂度就是O(k)，整个的时间复杂度就是O(k*k*n)．其实这里面的过程有一个信息被我们遗漏了，或者说是没有被利用好,那就是当我们取出这k个节点最小那个时候，其实除了最小的那个其他的节点之间大小关系也有比较过，比如是节点s和t,然后新替入的那个进来之后，再来一次取最小值，这时候其实s和t的大小关系我们是已知的，但是我们没有利用，而是有可能还会取比较一次! 为了避免这个重复比较，我们想到使用最小堆的方法: 把这个k个节点放入最小堆，然后取堆顶点，再加入新节点，整个操作只需要O(lgk)的时间复杂度，所以整个的时间复杂度降为O(k*nlgk)

```
1   ListNode *mergeKLists(vector<ListNode*> &lists) {
2       multimap<int, int> data;
3       ListNode ret(0) , *pos = &ret;
4       for(int i = 0; i < lists.size(); i++){
5           if(lists[i])
6               data.insert(make_pair(lists[i]->val, i));
7       }
8       while(!data.empty()){
9           int index = data.begin()->second;
10          data.erase(data.begin());
11          pos = pos->next = lists[index];
12          lists[index] = lists[index]->next;
13          if(lists[index])
14              data.insert(make_pair(lists[index]->val, index));
15      }
16      return ret.next;
17  }
```

上面没有使用堆，但是使用的是tree map,它其实可以用来模拟堆操作，整个的时间复杂度和用堆是一样的.

1.3.4 Swap Nodes in Pairs

问题: Given a linked list, swap every two adjacent nodes and return its head. *(leetcode 24)*

举例: Given $1 \to 2 \to 3 \to 4$, you should return the list as $2 \to 1 \to 4 \to 3$.

Note: Your algorithm should use only constant space. You may not modify the values in the list, only nodes itself can be changed.

两个指针: 时间复杂度O(n), 空间复杂度O(1)

这题主要是考查细心程度，还有对边界条件的考虑.

```
1   ListNode *swapPairs(ListNode *head) {
2       if(!head || !head->next)    return head;
3       ListNode newhead(0), *first = &newhead, *second = &newhead;
```

```
    newhead.next = head;
    while(first->next && first->next->next){
        second = first->next;
        first->next = second->next;
        second->next = first->next->next;
        first->next->next = second;
        first = second;
    }
    return newhead.next;
}
```

1.3.5 Reverse Nodes in k-Group

问题: Given a linked list, reverse the nodes of a linked list k at a time and return its modified list. *(leetcode 25)*

举例：Given this linked list: $1 \to 2 \to 3 \to 4 \to 5$, For k = 2, you should return: $2 \to 1 \to 4 \to 3 \to 5$, For k = 3, you should return: $3 \to 2 \to 1 \to 4 \to 5$

Note：If the number of nodes is not a multiple of k then left-out nodes in the end should remain as it is. You may not alter the values in the nodes, only nodes itself may be changed. Only constant memory is allowed.

Care：时间复杂度O(n), 空间复杂度O(1)

这题最保守的做法就是先算出链表的长度，然后一段一段的反转.

```
ListNode *reverseKGroup(ListNode *head, int k) {
    if(!head || k <= 1) return head;
    ListNode newhead(0), *first = &newhead, *second = &newhead, *pos =
        head;
    newhead.next = head;
    int count = 0;
    while(pos){
        pos = pos->next;
        count++;
    }
    while(count >= k){
        second = first->next;
        for(int i = 0; i < k - 1; i++){
            pos = second->next;
            second->next = pos->next;
            pos->next = first->next;
            first->next = pos;
        }
        count = count - k;
        first = second;
    }
    return newhead.next;
}
```

1.3.6 Rotate List

问题: Given a list, rotate the list to the right by k places, where k is non-negative. *(leetcode 61)*

示例：Given $1 \to 2 \to 3 \to 4 \to 5 \to NULL$ and k = 2, return $4 \to 5 \to 1 \to 2 \to 3 \to NULL$.

Care：时间复杂度O(n), 空间复杂度O(1)

首尾连接一下轻松搞定链表旋转

```
ListNode *rotateRight(ListNode *head, int k) {
    if(!head || k < 1)  return head;
    ListNode *pos = head, *pre = head;
    int count = 0;
    for(; count < k && pos; count++){
        pos = pos->next;
    }
    if(!pos)    return rotateRight(head, k%count);
    while(pos->next){
        pos = pos->next;
        pre = pre->next;
    }
    pos->next = head;
    head = pre->next;
    pre->next = NULL;
    return head;
}
```

1.3.7 Reorder List

问题: Given a singly linked list L: $L_0 \to L_1 \to \ldots \to L_{n-1} \to L_n$, reorder it to: $L_0 \to L_n \to L_1 \to L_{n-1} \to L_2 \to L_{n-2} \to$ ...*(leetcode 143)*

反转：时间复杂度O(n) , 空间复杂度O(1)

先获得链表的中间节点，再对后半部分反转，然后间隔插入到前半部分.

```
void reorderList(ListNode *head) {
    if(!head || !head->next)    return;
    ListNode *slow = head, *fast = head->next, *left = head, *right, *
        tmp;
    while(fast && fast->next){
        slow = slow->next;
        fast = fast->next->next;
    }
    right = slow->next;
    while(right->next){
        tmp = right->next;
        right->next = tmp->next;
        tmp->next = slow->next;
        slow->next = tmp;
    }
    right = slow->next;
    slow->next = NULL;
    while(left->next){
        tmp = right->next;
        right->next = left->next;
        left->next = right;
        right = tmp;
        left = left->next->next;
    }
    left->next = right;
}
```

1.3.8 Remove Linked List Elements

问题:
Remove all elements from a linked list of integers that have value val.

Example
Given: $1 \to 2 \to 6 \to 3 \to 4 \to 5 \to 6, val = 6$
Return: $1 \to 2 \to 3 \to 4 \to 5$
(leetcode 203)

递归：时间复杂度O(n), 空间复杂度O(1)
递归写法很简洁,虽然里面有三目运算符还有逗号表达式

```
ListNode* removeElements(ListNode* head, int val) {
    if(!head)   return NULL;
    return head->val == val? removeElements(head->next, val) :
            (head->next = removeElements(head->next, val), head);
}
```

1.3.9 Remove Duplicates from Sorted List

问题: Given a sorted linked list, delete all duplicates such that each element appear only once. *(leetcode 83)*

示例：Given $1 \to 1 \to 2$, return $1 \to 2$. Given $1 \to 1 \to 2 \to 3 \to 3$, return $1 \to 2 \to 3$.

两个指针：时间复杂度O(n), 空间复杂度O(1)
遇到一样的就删除，不一样就转移.

```
ListNode *deleteDuplicates(ListNode *head) {
    if(!head || !head->next)    return head;
    ListNode *pre = head , *next = head->next;
    while(next){
        if(pre->val == next->val){
            pre->next = next->next;
        }else{
            pre = pre->next;
        }
        next = pre->next;
    }
    return head;
}
```

1.3.10 Remove Duplicates from Sorted List II

问题: Given a sorted linked list, delete all nodes that have duplicate numbers, leaving only distinct numbers from the original list. *(leetcode 82)*

示例：Given $1 \to 2 \to 3 \to 3 \to 4 \to 4 \to 5$, return $1 \to 2 \to 5$. Given $1 \to 1 \to 1 \to 2 \to 3$, return $2 \to 3$.

两个指针：时间复杂度O(n) , 空间复杂度O(1)
需要注意的就是如果最后几个是相同的也要删除.

```
ListNode *deleteDuplicates(ListNode *head) {
    if(!head || !head->next)    return head;
    ListNode newhead(0);
    ListNode *pre = &newhead, *next = head->next;
    newhead.next = head;
    while(next){
        if(pre->next->val == next->val){
            next = next->next;
        }else{
            if(pre->next->next == next){
                pre = pre->next;
            }else{
                pre->next = next;
            }
            next = next->next;
        }
    }
    if(pre->next->next != next)
        pre->next = next;
    return newhead.next;
}
```

1.3.11 Partition List

问题:Given a linked list and a value x, partition it such that all nodes less than x come before nodes greater than or equal to x. *(leetcode 86)*

举例：Given $1 \to 4 \to 3 \to 2 \to 5 \to 2$ and x = 3, return $1 \to 2 \to 2 \to 4 \to 3 \to 5$.

Care：时间复杂度O(n), 空间复杂度O(1)

分别用两个链表指针维护小于x和大于x的，最后再粘结到一起就行了.

```
ListNode *partition(ListNode *head, int x) {
    if(!head || !head->next)    return head;
    ListNode    less(0), greater(0), *pos = head;
    ListNode    *lpos = &less, *gpos = &greater;
    while(pos){
        if(pos->val < x){
            lpos = lpos->next = pos;
        }else{
            gpos = gpos->next = pos;
        }
        pos = pos->next;
    }
    gpos->next = NULL;
    lpos->next = greater.next;
    return less.next;
}
```

1.3.12 Reverse Linked List II

问题: Reverse a linked list from position m to n. Do it in-place and in one-pass. *(leetcode 92)*

举例：Given $1 \to 2 \to 3 \to 4 \to 5 \to NULL$, m = 2 and n = 4, return $1 \to 4 \to 3 \to 2 \to 5 \to NULL$.

Note：Given m, n satisfy the following condition: $1 \leqslant m \leqslant n \leqslant$ length of list.

Care：时间复杂度O(n), 空间复杂度O(1)

这题主要是边界条件比较复杂，如何写出优雅的囊括边界条件检测的代码是个问题.

```
ListNode *reverseBetween(ListNode *head, int m, int n) {
    if(!head || !head->next || m >= n)  return head;
    ListNode newhead(0);
    newhead.next = head;
    ListNode *pos = &newhead, *next, *tmp;
    for(int i = 0; i < m - 1; i++){
        pos = pos->next;
    }
    next = pos->next;
    for(int i = 0; i < n - m; i++){
        tmp = next->next;
        next->next = tmp->next;
        tmp->next = pos->next;
        pos->next = tmp;
    }
    return newhead.next;
}
```

使用一个新的头节点是一个很好的习惯.

1.3.13 Copy List with Random Pointer

问题: A linked list is given such that each node contains an additional random pointer which could point to any node in the list or null. Return a deep copy of the list. *(leetcode 138)*

哈希表：时间复杂度O(n), 空间复杂度O(n)

链表的缺点在于不能随机访问，但是如果和哈希表来联合使用那么就可以做到随机访问了.这道题在遍历链表时候，一边遍历一边建立新的链表节点，然后使用哈希表记录新老节点的映射关系，等到再一次遍历老链表的时候根据映射表就可以快速的定位到新链表对应位置，不必再从头开始找.

```
RandomListNode *copyRandomList(RandomListNode *head){
    if(head == NULL) return NULL;
    stack<RandomListNode*> to_be;
    unordered_map<RandomListNode*, RandomListNode*> have_new;
    RandomListNode* pos, *next, *rand;
    to_be.push(head);
    pos = head;
    RandomListNode* new_head, *new_pos, *new_next, *new_rand;
    new_head = new RandomListNode(head->label);
    new_pos = new_head;
    have_new[pos] = new_pos;
    while(!to_be.empty()){
        pos = to_be.top();
        to_be.pop();
        next = pos->next;
        rand = pos->random;
        new_pos = have_new[pos];
        if(next != NULL){
            if(have_new.count(next) == 1){
                new_pos->next = have_new[next];
            }
            else{
```

```
23                  new_next  = new RandomListNode(next->label);
24                  new_pos->next = new_next;
25                  have_new[next] = new_next;
26                  to_be.push(next);
27              }
28          }else{
29              new_pos->next = NULL;
30          }
31          if(rand != NULL){
32              if(have_new.count(rand) == 1){
33                  new_pos->random = have_new[rand];
34              }
35              else{
36                  new_rand = new RandomListNode(rand->label);
37                  new_pos->random = new_rand;
38                  have_new[rand] = new_rand;
39                  to_be.push(rand);
40              }
41          }else{
42              new_pos->random = NULL;
43          }
44      }
45      return new_head;
46  }
```

插入法：时间复杂度O(n), 空间复杂度O(1)

插入法实质上也是在建立新老节点的对应关系，不像哈希表这种需要额外空间的对应，而是非常巧妙的在老链表的结构上建立映射关系，举重若轻.

```
1   RandomListNode *copyRandomList(RandomListNode *head){
2       if(!head)   return NULL;
3       RandomListNode *pos = head, *tmp, *ret;
4       while(pos){
5           tmp = new RandomListNode(pos->label);
6           tmp->next = pos->next;
7           pos->next = tmp;
8           pos = pos->next->next;
9       }
10      pos = head;
11      while(pos){
12          pos->next->random = pos->random? pos->random->next : NULL;
13          pos = pos->next->next;
14      }
15      pos = head;
16      ret = pos->next;
17      while(pos){
18          tmp = pos->next;
19          pos->next = tmp->next;
20          tmp->next = pos->next? pos->next->next : NULL;
21          pos = pos->next;
22      }
23      return ret;
24  }
```

其实这道题你还可以这样看，这里链表的节点有个next指针和random指针，其实这就是一个图了(链表是特殊的树，树是特殊的图)，所以套用图的DFS和BFS，你在处理过程中就有两种选择： 比如你建立老链表第一个节点的对应新节点后，你可以类似DFS选择直接深度

优先的访问next节点next的next节点等等；同样你也可以类似BFS下次访问next节点和rand节点.所以你可以使用递归的方式实现.

第 2 章　树

2.1　二叉树的遍历

二叉树的遍历是解决很多二叉树问题的基础，它的递归写法和非递归写法更是需要都要掌握的，这里的遍历就是将树的节点都依次访问一遍，因为访问顺序的问题，就可以分为前序，中序，后序以及层次等很多种遍历的方法。

2.1.1　PreOrederTraversal

问题: Given a binary tree, return the preorder traversal of its nodes' values.(*leetcode 144*)

递归：时间复杂度O(n)，空间复杂度O(lgn)

前序遍历的递归写法，非常简单，只需要先访问跟节点，再递归的执行左子树和右子树

```
1  void BiTree::PreOrderTraversal(TreeNode* root){
2      if(!root)    return;
3      cout<<root->val<<endl;
4      if(root->left)
5          PreOrderTraversal(root->left);
6      if(root->right)
7          PreOrderTraversal(root->right);
8  }
```

栈式迭代：时间复杂度O(n)，空间复杂度O(lgn)

使用栈来模拟递归过程也是很显而易见的，具体做法就是先访问根节点，然后先让右孩子入栈接着是左孩子，然后左孩子出栈后重复这个过程

```
1  void BiTree::PreOrderTraversal(TreeNode* root){
2      if(!root)    return;
3      stack<TreeNode*> s;
4      TreeNode *cur;
5      s.push(root);
6      while(!s.empty()){
7          cur = s.top();
8          s.pop();
9          cout<<cur->val<<endl;
10         if(cur->right)
11             s.push(cur->right);
12         if(cur->left)
13             s.push(cur->left);
14     }
15 }
```

Mirror迭代：时间复杂度O(n)，空间复杂度O(1)

Mirror迭代法是经过Lee介绍过来的，非常的迷人,它的做法就是在遍历的过程中，访问了当前节点之后，先找当前节点的前驱并让此前驱的右孩子指向它，再访问它的左孩子并重复这个过程。在此之后会访问到它前驱然后再次回到当前节点，此时再次试图建立前驱，发现已经建立了，这就说明当前节点左边已经全部遍历完，则继续访问当前节点的右边节点，不断的重复此过程。

```
void BiTree::PreOrderTraversal(TreeNode* root){
    if(!root)    return;
    TreeNode *curr = root, *next;
    while(curr){
        next = curr->left;
        if(!next){
            cout<<curr->val<<endl;
            curr = curr->right;
            continue;
        }
        while(next->right && next->right != curr){
            next = next->right;
        }
        if(next->right == curr){
            next->right = NULL;
            curr = curr->right;
        }else{
            cout<<curr->val<<endl;
            next->right = curr;
            curr = curr->left;
        }
    }
}
```

这个*Mirror*算法一旦掌握后，威力无穷，你可以用它方便的建立二叉树前序索引并且遇到那些要求用迭代来实现的二叉树问题也可以很快的写出来

2.1.2 InOrederTraversal

问题: Given a binary tree, return the inorder traversal of its nodes' values. *(leetcode 94)*

递归：时间复杂度O(n)，空间复杂度O(lgn)

```
void BiTree::InOrderTraversal(TreeNode* root){
    if(!root)    return;
    if(root->left)
        InOrderTraversal(root->left);
    cout<<root->val<<endl;
    if(root->right)
        InOrderTraversal(root->right);
}
```

栈式迭代：时间复杂度O(n)，空间复杂度O(lgn)

这里需要说一下的是，数据结构那本书上写了两种栈式迭代方法，这是其中之一，使用两重循环的那个

```
void BiTree::InOrderTraversal(TreeNode* root){
    vector<int> data;
    if(!root)    return data;
    stack<TreeNode*> s;
```

```
    TreeNode *pos = root;
    while(!s.empty() || pos){
        while(pos){
            s.push(pos);
            pos = pos->left;
        }
        pos = s.top();
        s.pop();
        std::cout<<pos->val<<std::endl;
        pos = pos->right;  //这个非常重要
    }
}
```

栈式迭代：时间复杂度O(n)，空间复杂度O(lgn)

这里需要说一下的是，数据结构那本书上写了两种栈式迭代方法，这是其中之二，使用一重循环，实际上是一样的

```
void BiTree::InOrderTraversal(TreeNode* root){
    vector<int> data;
    if(!root)    return data;
    stack<TreeNode*> s;
    TreeNode *pos = root;
    while(!s.empty() || pos){
        if(pos){
            s.push(pos);
            pos = pos->left;
        }else{
            pos = s.top();
            s.pop();
            std::cout<<pos->val<<std::endl;
            pos = pos->right;  //这个非常重要
        }
    }
}
```

Mirror迭代：时间复杂度O(n)，空间复杂度O(1)

这里Mirror方法和前序的Mirror方法基本一样，唯一的区别就是打印当前值的时机

```
void BiTree::InOrderTraversal(TreeNode* root){
    if(!root)    return;
    TreeNode *curr = root, *next;
    while(curr){
        next = curr->left;
        if(!next){
            cout<<curr->val<<endl;
            curr = curr->right;
            continue;
        }
        while(next->right && next->right != curr){
            next = next->right;
        }
        if(next->right == curr){
            next->right = NULL;
            cout<<curr->val<<endl;
            curr = curr->right;
        }else{
            next->right = curr;
```

```
20              curr = curr->left;
21          }
22      }
23  }
```

2.1.3 PostOrederTraversal

问题: Given a binary tree, return the postorder traversal of its nodes' values. *(leetcode 145)*

递归：时间复杂度O(n)，空间复杂度O(lgn)

```
1  void BiTree::PostOrderTraversal(TreeNode* root){
2      if(!root)    return;
3      if(root->left)
4          PostOrderTraversal(root->left);
5      if(root->right)
6          PostOrderTraversal(root->right);
7      cout<<root->val<<endl;
8  }
```

栈式迭代: 时间复杂度O(n), 空间复杂度O(lgn)

这里判断一个节点的孩子是否被访问过的方法是：记录上一次打印的节点，如果上一次打印的节点是它的孩子节点，那么必然它的所有孩子及其子树都访问完了，换句话说该访问它本身了.

```
1  void BiTree::PostOrderTraversal(TreeNode* root){
2      if(!root)    return;
3      stack<TreeNode*> s;
4      TreeNode *pos = root, *last = root;
5      s.push(root);
6      while(!s.empty()){
7          pos = s.top();
8          if(pos->left == last || pos->right == last || (!pos->left && !
               pos->right)){
9          //孩子已经打印完毕或者根本就没有孩子
10             cout<<pos->val<<endl;
11             last = pos;
12             s.pop();
13         }else{
14             if(pos->right){
15                 s.push(pos->right);
16             }
17             if(pos->left){
18                 s.push(pos->right);
19             }
20         }
21     }
22 }
```

这里没有出现万众期待的*Mirror*算法，主要是后序使用*Mirror*很复杂，我暂时还没有想到怎么实现 ^_^

2.1.4 LevelOrderTraversal

问题: Given a binary tree, return the level order traversal of its nodes' values. (ie, from left to right, level by level). *(leetcode 145)*

队列：时间复杂度O(n)，空间复杂度O(w), w为二叉树最大宽度

使用队列，上一层进入队列，然后添加下一层，直到不再有节点进入队列

```
vector<vector<int> > levelOrder(TreeNode *root) {
    vector<vector<int> > data;
    if(!root)    return data;
    queue<TreeNode*> cur;
    cur.push(root);
    int size = 0;
    TreeNode* now;
    while(!cur.empty()){
        vector<int> tmp;
        size = cur.size();
        for(int i = 0; i < size; i++){
            now = cur.front();
            cur.pop();
            tmp.push_back(now->val);
            if(now->left)
                cur.push(now->left);
            if(now->right)
                cur.push(now->right);
        }
        data.push_back(tmp);
    }
    return data;
}
```

其实层次遍历除了这个先上而下，先左而右的顺序以外还有很多顺序，但是都可以通过这个顺序来转换，所以就不再仔细讨论

2.1.5 Recover Binary Search Tree

问题: Two elements of a binary search tree (BST) are swapped by mistake. Recover the tree without changing its structure. *(leetcode 99)*

BST树的中序遍历是排序好的，那么可以通过中序遍历来看一下哪两个地方发生了交换,发生交换的地方必然是前面那个数比后面的大,只要在遍历过程记录这个位置就可以了.

递归：时间复杂度O(n), 空间复杂度O(lgn)

这段代码很有技巧，需要细细品读. 对于1,5,3,4,2这个序列: fisrt是5, second是2;对于1,3,2,4,5这个序列: first是3, second是2.

```
void dfs(TreeNode* root, int& last, TreeNode* &first, TreeNode* &second)
    {
    if(root->left){
        dfs(root->left, last, first, second);
    }
    if(root->val < last){
        second = root;
    }
    if(!second){
        first = root;
    }
    last = root->val;
    if(root->right){
        dfs(root->right, last, first, second);
    }
}

```

```
17 void recoverTree(TreeNode* root) {
18     if(!root)    return;
19     TreeNode *first = NULL, *second = NULL;
20     int min = (-1)<<31;
21     dfs(root, min, first, second);
22     int tmp = first->val;
23     first->val = second->val;
24     second->val = tmp;
25 }
```

迭代：时间复杂度O(n), 空间复杂度O(lgn)

```
1  void recoverTree(TreeNode* root) {
2      if(!root)    return;
3      TreeNode *first = NULL, *second = NULL, *cur;
4      int last = INT_MIN;
5      stack<TreeNode*> s;
6      s.push(root);
7      while(!s.empty()){
8          cur = s.top();
9          while(cur){
10             s.push(cur->left);
11             cur = cur->left;
12         }
13         s.pop();
14         if(!s.empty()){
15             cur = s.top();
16             if(cur->val < last){
17                 second = cur;
18             }
19             if(!second){
20                 first = cur;
21             }
22             last = cur->val;
23             s.pop();
24             s.push(cur->right);
25         }
26     }
27     int tmp = first->val;
28     first->val = second->val;
29     second->val = tmp;
30 }
```

2.2 二叉树的建立

2.2.1 Construct Binary Tree from Preorder and Inorder Traversal

问题: Given preorder and inorder traversal of a tree, construct the binary tree. *(leetcode 105)*

Note: You may assume that duplicates do not exist in the tree.

递归: 时间复杂度O($nlgn$) , 空间复杂度O(lgn)

前序序列的第一个元素肯定是树的根节点，而且使用这个值在中序序列中查找，找到的那个位置之前的必然是左子树，之后的必然是右子树，所以根据这个特点就可以很容易的使用递归的做法解题。

```
1  TreeNode* detail(vector<int> &inorder, vector<int> &preorder,
2                   int in_start, int in_end, int pre_start, int pre_end){
3  //inorder[in_start, in_end], preorder[pre_start, pre_end]
4      if(in_start > in_end || pre_start > pre_end) return NULL;
5      int val = preorder[pre_start];
6
7      TreeNode* father = new TreeNode(val);
8      TreeNode *left = NULL, *right = NULL;
9
10     int in_pos = in_start, pre_pos = pre_start;
11     for(; in_pos <= in_end; in_pos++, pre_pos++){
12         if(val == inorder[in_pos])
13             break;
14     }
15
16     left = detail(inorder, preorder, in_start, in_pos-1, pre_start+1,
           pre_pos);
17     right = detail(inorder, preorder, in_pos+1, in_end, pre_pos+1,
           pre_end);
18
19     father->left = left;
20     father->right = right;
21
22     return father;
23 }
24
25 TreeNode *buildTree(vector<int> &preorder, vector<int> &inorder) {
26     int size = inorder.size();
27     if(size == 0) return NULL;
28         return detail(inorder, preorder, 0, size - 1, 0, size - 1);
29 }
```

2.2.2 Construct Binary Tree from Postorder and Inorder Traversal

问题: Given postorder and inorder traversal of a tree, construct the binary tree. *(leetcode 106)*

Note: You may assume that duplicates do not exist in the tree.

递归: 时间复杂度O($nlgn$) , 空间复杂度O(lgn)

后序序列的最后一个元素肯定是树的根节点，而且使用这个值在中序序列中查找，找到的那个位置之前的必然是左子树，之后的必然是右子树，所以根据这个特点就可以很容易的使用递归的做法解题。

```
TreeNode* recursion(vector<int> &inorder, vector<int> &postorder,
                    int s_in, int e_in, int s_po, int e_po){
    //inorder[s_in, e_in],postorder[s_po, e_po]
    if(s_in > e_in)    return NULL;
    if(s_in == e_in) return (new TreeNode(inorder[s_in]));
    TreeNode *root = new TreeNode(postorder[e_po]);
    int split_in, split_po;
    for(split_in = s_in; split_in <= e_in; split_in++){
        if(postorder[e_po] == inorder[split_in])
            break;
    }
    split_po = s_po + split_in - s_in;
    root->left = recursion(inorder, postorder, s_in, split_in - 1, s_po
        , split_po - 1 );
    root->right = recursion(inorder, postorder, split_in + 1, e_in,
        split_po, e_po - 1);
    return root;
}

TreeNode *buildTree(vector<int> &inorder, vector<int> &postorder) {
    if(inorder.size() != postorder.size() || inorder.size() == 0)
        return NULL;
    return recursion(inorder, postorder, 0, inorder.size() - 1, 0,
        postorder.size() - 1);
}
```

2.2.3 Convert Sorted Array to Binary Search Tree

问题: Given an array where elements are sorted in ascending order, convert it to a height balanced BST. *(leetcode 108)*

递归：时间复杂度O(n), 空间复杂度O(n)

建立平衡的二叉树，那么我们每次取数组的中间位置那个元素为根节点，然后它之前的部分创建左子树，之后的部分创建右子树，那么很容易就可以使用递归实现.

```
TreeNode* dfs(vector<int> &num, int start, int end){
    if(start > end)    return NULL;
    if(start == end)    return (new TreeNode(num[start]));
    int mid = (start + end) / 2;
    TreeNode* root = new TreeNode(num[mid]);
    root->left = dfs(num, start, mid - 1);
    root->right = dfs(num, mid + 1, end);
    return root;
}

TreeNode *sortedArrayToBST(vector<int> &num) {
    int size = num.size();
    return dfs(num, 0, size - 1);
}
```

因为是数组，所以可以很方便的找到中位数，但是如果是链表，则需要使用一些小技巧

2.2.4 Convert Sorted List to Binary Search Tree

问题: Given a singly linked list where elements are sorted in ascending order, convert it to a height balanced BST. *(leetcode 109)*

递归：时间复杂度O(n), 空间复杂度O(n)

前面使用数组构造BST树，我们可以看到每次需要求出它的中间的那个数，然后以它创建根节点，但是对于有序链表来说，找到中位数起码要花O(n)时间，那么这样算下来整个程序需要O($nlgn$)的时间！这似乎和数组的O(n)差别比较大。我们可以想一下，摒弃这种自上而下的思维，来一次自下而上的方法：我们先构建左子树，构建完了之后访问的最后一点节点是不是就是根节点的前驱？这样我们记下这个前驱，然后一记next是不是就求出我们梦寐以求的中位数那个节点！然后再next一下，构建右子树，看看发生了什么？我们一边建树一边就得到中间节点，所以就省掉了找中间节点的那个时间！

```
TreeNode* DFS(ListNode* &head, int n){
    if(n == 0)    return NULL;
    TreeNode *root;
    if(n == 1){
        root = new TreeNode(head->val);
        head = head->next;
        return root;
    }
    TreeNode *left = DFS(head, n/2);// head travel the list
    root = new TreeNode(head->val);
    head = head->next;
    TreeNode *right = DFS(head, n - 1 - n/2 );
    root->left = left;
    root->right = right;
    return root;
}

TreeNode *sortedListToBST(ListNode *head) {
    if(!head)    return NULL;
    int count = 0;
    ListNode *pos = head;
    while(pos){//compute the length of the list
        pos = pos->next;
        count++;
    }
    return DFS(head, count);
}
```

这里使用自下而上的做法很具有普遍性，我们从上面看得不到的东西，从下面可以积累到

2.2.5 Unique Binary Search Trees

问题: Given n, how many structurally unique BST's (binary search trees) that store values 1...n? *(leetcode 96)*

这道题虽然是一个二叉树的问题，但是其实是数学归纳法的问题，我们可以得到递推公式：

$$\begin{aligned} S(0) &= 1 \\ S(n) &= \sum_{i=0}^{n-1} S(i) * S(n-1-i) \end{aligned} \tag{2.1}$$

这是因为对于有n个元素的BST(1,2,...,n)，我们考虑由谁来作为根节点，如果以i+1为根节点，那么左子树为(1,2,...,i),右子树为(i+2,i+3,...,n),所以可以得到上面的递推关系.

迭代：时间复杂度O(n^2)，空间复杂度O(n)

```
int numTrees(int n) {
    if(!n)    return 0;
```

```
3       vector<int> num(n+1, 0);
4       num[1] = 1;
5       num[2] = 2;
6       for(int i = 3; i <= n; i++){
7           num[i] = 2*num[i-1];
8           for(int j = 1; j < i; j++){
9               num[i] = num[i] + num[j]*num[i-1-j];
10          }
11      }
12      return num[n];
13  }
```

2.2.6 Unique Binary Search Trees II

问题: Given n, generate all structurally unique BST's (binary search trees) that store values 1...n. *(leetcode 95)*

: 时间复杂度 TODO , 空间复杂度 TODO

每次对于构造序列(i,i+1,...,j),切分j-i+1次，然后分别递归构造.

```
1   vector<TreeNode*> generate(int start, int end){
2       if(start == end){
3           return vector<TreeNode*>(1, new TreeNode(start));
4       }
5       vector<TreeNode*> data;
6       if(start > end){
7           data.push_back(NULL);
8           return data;
9       }
10      for(int i = start; i <= end; i++){
11          TreeNode* root;
12          vector<TreeNode*> left = generate(start, i - 1);
13          vector<TreeNode*> right = generate(i + 1, end);
14          for(int j = 0; j < left.size(); j++){
15              for(int k = 0; k < right.size(); k++){
16                  root = new TreeNode(i);
17                  root->left = left[j];
18                  root->right = right[k];
19                  data.push_back(root);
20              }
21          }
22      }
23      return data;
24  }
25
26  vector<TreeNode*> generateTrees(int n) {
27      return generate(1, n);
28  }
```

2.3 二叉树的属性

2.3.1 Validate Binary Search Tree

问题: Given a binary tree, determine if it is a valid binary search tree (BST). *(leetcode 98)*
判断一个二叉树是否是合法的BST，我们可以想到BST树的中序序列是非减序列，于是我们可以使用中序遍历这颗二叉树，在遍历的过程中查看是否有反常的数据.
当然，根据上面说的三种中序遍历的方法，这里同样有三种解法.

递归：时间复杂度O(n)，空间复杂度O(lgn)

```
bool dfs(TreeNode *root, int& up){
    if(!root)    return true;
    if(root->left){
        bool left =  dfs(root->left, up);
        if(!left) return false;
    }
    if(root->val <= up && (MIN || up != (-1)<<31)){
        return false;
    }
    if(root->val == (-1)<<31)
        MIN = true;
    up = root->val;
    if(root->right){
        bool right = dfs(root->right, up);
        if(!right)    return false;
    }
    return true;
}

bool isValidBST(TreeNode *root) {
    if(!root)    return true;
    int up = (-1)<<31;
    MIN = false;
    return dfs(root, up);
}
```

这里可以看到一些边界条件的判断，显得有点复杂，其实就是简单的中序遍历

栈式迭代：时间复杂度O(n)，空间复杂度O(lgn)

```
bool isValidBST(TreeNode *root) {
    if(!root)    return false;
    stack<TreeNode*> s;
    TreeNode *p = root;
    s.push(root);
    while(p->left){
        s.push(p->left);
        p = p->left;
    }
    int last = p->val;
    s.pop();
    if(p->right){
        s.push(p->right);
        p = p->right;
        while(p->left){
            s.push(p->left);
            p = p->left;
        }
```

```
19      }
20      while(!s.empty()){
21          p = s.top();
22          s.pop();
23          if(last >= p->val) return false;
24          last = p->val;
25          if(p->right){
26              s.push(p->right);
27              p = p->right;
28              while(p->left){
29                  s.push(p->left);
30                  p = p->left;
31              }
32          }
33      }
34      return true;
35  }
```

Mirror迭代： 时间复杂度O(n)，空间复杂度O(1)

这里使用Mirror建立线索然后进行中序遍历，在中序遍历的同时进行判断

```
1   bool isValidBST(TreeNode *root) {
2       if(!root)    return true;
3       TreeNode *curr = root, *next;
4       int last = INT_MIN;
5       bool isFirst = true;
6       bool ret = true;
7       while(curr){
8           if(!curr->left){
9               if(!isFirst && curr->val <= last){
10                  ret = false;
11              }
12              if(isFirst)
13                  isFirst = false;
14              last = curr->val;
15              curr = curr->right;
16              continue;
17          }
18          next = curr->left;
19          while(next->right){
20              if(next->right == curr)    break;
21              next = next->right;
22          }
23          if(next->right == curr){
24              next->right = NULL;
25              if(!isFirst && curr->val <= last){
26                  ret = false;
27              }
28              if(isFirst)
29                  isFirst = false;
30              last = curr->val;
31              curr = curr->right;
32          }else{
33              next->right = curr;
34              curr = curr->left;
35          }
36      }
37      return ret;
```

```
38 }
```

有了*Mirror*算法，是不是你已经爱上它了，再也不用栈这么麻烦了,不过有一点需要注意的是一旦你使用*Mirror*算法，那么必须保证把整个树全遍历一遍，不能中途退出，因为那样树的结构被改变了

2.3.2 Symmetric Tree

问题: Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center). *(leetcode 101)*
这是求证树是不是自身Mirror(成镜像).

队列：时间复杂度O(n)，空间复杂度O(w), w为树的最大宽度

```
1  bool isSymmetric(TreeNode* root){
2      if(!root) return true;
3      deque<TreeNode*> left(1, root->left), right(1, root->right);
4      TreeNode *l, *r;
5      while(!left.empty() && !right.empty()){
6          l = left.front();
7          r = right.front();
8          left.pop_front();
9          right.pop_front();
10         if(!l && !r)    continue;
11         if(!l || !r || l->val != r->val)    return false;
12         left.push_back(l->left);
13         left.push_back(l->right);
14         right.push_back(r->right);
15         right.push_back(r->left);
16     }
17     return true;
18 }
```

递归：时间复杂度O(n)，空间复杂度O(lgn)
这里是把一棵树的对称问题看成两棵树的对称问题

```
1  bool recursion(TreeNode* root, TreeNode* symm){
2      if(!root && !symm)
3          return true;
4      if(!root || !symm)    return false;
5      if(root->val != symm->val)    return false;
6      if(root == symm)    return recursion(root->left, symm->right);
7      return recursion(root->left, symm->right) && recursion(root->right,
           symm->left);
8  }
9
10 bool isSymmetric(TreeNode* root){
11     if(!root) return true;
12     return recursion(root, root);
13 }
```

这里还可以延伸出一个问题： 求一个二叉树的镜像树

2.3.3 Same Tree

问题: Given a binary tree, determine if it is height-balanced. *(leetcode 110)*

递归：时间复杂度O(n)，空间复杂度O(lgn)

先判断左子树是否高度平衡并返回左子树高度，再判断右子树是否高度平衡，再返回右子树高度，根据左右子树高度再判断当前树是否平衡.

```
bool dfs(TreeNode *root, int &hight){
    if(!root){
        hight = 0;
        return true;
    }
    int left, right;
    bool is_left = dfs(root->left, left);
    bool is_right = dfs(root->right, right);
    hight = left > right? left + 1 : right + 1;
    return is_left && is_right && (abs(left - right) < 2);
}

bool isBalanced(TreeNode *root) {
    int hight;
    return dfs(root, hight);
}
```

2.3.4 Maximum Depth of Binary Tree

问题: Given a binary tree, find its maximum depth.*(leetcode 104)*

从根节点来看，它的深度就是左右子树深度较大的那个+1,所以很自然的想到递归

递归：时间复杂度O(n)，空间复杂度O(lgn)

递归代码十分简洁

```
int maxDepth(TreeNode *root){
    if(!root)   return 0;
    int left = maxDepth(root->left);
    int right = maxDepth(root->right);
    return left < right? right + 1 : left + 1;
}
```

除了递归，其实这道题能不能用迭代的做法呢？答案是肯定的，最初你可能会想到用两个栈，一个栈存放节点，一个栈存放深度，其实可以把这个两者打包成一个pair，使用一个栈就可以啦

迭代：时间复杂度O(n)，空间复杂度O(lgn)

```
int maxDepth(TreeNode *root) {
    if(!root)    return 0;
    stack<pair<TreeNode*, int> > s;
    s.push(make_pair(root, 1));
    pair<TreeNode*, int> curr;
    int result = INT_MIN;
    while(!s.empty()){
        curr = s.top();
        s.pop();
        if(!curr.first->left && !curr.first->right){
            if(result < curr.second)
                result = curr.second;
            continue;
        }
        if(curr.first->left){
```

```
16                 s.push(make_pair(curr.first->left, curr.second + 1));
17             }
18             if(curr.first->right){
19                 s.push(make_pair(curr.first->right, curr.second + 1));
20             }
21         }
22         return result;
23     }
```

这种自上而下的过程(比如节点深度)，用迭代实现就很简单，但是如果是自下而上的过程呢？好像很复杂，你有什么好的想法？

2.3.5 Minimum Depth of Binary Tree

问题: Given a binary tree, find its minimum depth. The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node. *(leetcode 111)*

递归：时间复杂度O(n), 空间复杂度O(lgn)

自下而上的递归，非常的简单

```
1  int minDepth(TreeNode *root) {
2      if(!root)    return 0;
3      if(!root->left && !root->right)    return 1;
4      if(!root->left)
5          return minDepth(root->right) + 1;
6      if(!root->right)
7          return minDepth(root->left) + 1;
8      return min(minDepth(root->left), minDepth(root->right)) + 1;
9  }
```

DFS：时间复杂度O(n)，空间复杂度O(lgn)

这也是递归，但是是一种自上而下的递归，可以进行剪枝而不必把整个树都访问一遍

```
1  void dfs(TreeNode *root, int &result, int depth){
2      if(result < depth + 1) return;
3      if(!root->left && !root->right){
4          result = depth + 1;
5          return;
6      }
7      if(root->left)
8          dfs(root->left, result, depth + 1);
9      if(root->right)
10         dfs(root->right, result, depth + 1);
11 }
12
13     int minDepth(TreeNode *root) {
14     if(!root)    return 0;
15     int result = INT_MAX;
16     dfs(root, result, 0);
17     return result;
18 }
```

迭代：时间复杂度O(n)，空间复杂度O(lgn)

同样我们也可以剪枝

```cpp
int minDepth(TreeNode *root) {
    if(!root)    return 0;
    stack<pair<TreeNode*, int> > s;
    s.push(make_pair(root, 1));
    int     result = -((1<<31) + 1);
    TreeNode *node;
    int depth;
    while(!s.empty()){
        node = s.top().first;
        depth = s.top().second;
        s.pop();
        if(result < depth)    continue;
        if(!node->left && !node->right)
            result = depth;

        if(node->left )
            s.push(make_pair(node->left, depth + 1));
        if(node->right)
            s.push(make_pair(node->right, depth + 1));
    }
    return result;
}
```

2.3.6 Path Sum

问题: Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum. *(leetcode 112)*

递归：时间复杂度O(n)，空间复杂度O(lgn)

先减去当前节点的值，剩余的值再分别递归求解左右子树

```cpp
bool hasPathSum(TreeNode *root, int sum) {
    if(root == NULL) return false;
    int val = root->val;
    sum = sum - val;
    if(sum == 0 && !root->left && !root->right)
        return true;

    bool left = false;
    if(root->left){
        left = hasPathSum(root->left, sum);
    }
    if(left) return true;
    if(root->right)
        return hasPathSum(root->right, sum);
    return false;
}
```

仿照最大/小深度问题，可以在迭代栈加上附加路径和这个信息，那么实现起来就非常简单了

迭代：时间复杂度O(n)，空间复杂度O(lgn)

```cpp
bool hasPathSum(TreeNode *root, int sum) {
    if(!root)    return false;
    stack<pair<TreeNode*, int> > s;
    s.push(make_pair(root, root->val));
```

```
5       TreeNode *node;
6       int e;
7       while(!s.empty()){
8           node = s.top().first;
9           e = s.top().second;
10          s.pop();
11          if(!node->left && !node->right && e == sum)
12              return true;
13          if(node->left)
14              s.push(make_pair(node->left, e + node->left->val));
15          if(node->right)
16              s.push(make_pair(node->right, e + node->right->val));
17      }
18      return false;
19  }
```

可以看得出，迭代栈加入附件信息是一个常用的技巧，需要深刻理解和掌握. 另外，这道题还可以变形为求二叉树最远两个节点距离的问题，这就相当于把每个节点的权值都设为1，方法和这道题一样，而且还要比它简单.

2.3.7 Path Sum II

问题: Given a binary tree and a sum, find all root-to-leaf paths where each path's sum equals the given sum. *(leetcode 113)*
这题是Path Sum问题的延续，解法其实是一模一样的.

递归：时间复杂度O(n)，空间复杂度O(n)
这里有个技巧需要注意的就是，可以使用一个引用参数cur来记录跟节点到当前节点这条路径中的所有值，我们在进入某个节点后cur要push这个节点，在离开这个节点后就要pop这个节点

```
1   vector<vector<int> > pathSum(TreeNode *root, int sum){
2       vector<int> cur;
3       vector<vector<int> > result;
4       recursion(root, cur, result, sum);
5       return result;
6   }
7
8   void recursion(TreeNode *root, vector<int> &cur, vector<vector<int> > &
        result, int sum){
9       if(!root)     return;
10      cur.push_back(root->val);
11      if(!root->left && !root->right && root->val == sum)
12          result.push_back(cur);
13      if(root->left)
14          recursion(root->left, cur, result, sum - root->val);
15      if(root->right)
16          recursion(root->right, cur, result, sum - root->val);
17      cur.pop_back();
18  }
```

我们知道Path Sum有迭代的做法，但是那种通过栈附加信息的做法对于我们这题还要求求出路径的问题不是很适合，所以就没有写出这种做法了，你有什么好想法么?

2.3.8 Binary Tree Maximum Path Sum

问题: Given a binary tree, find the maximum path sum.The path may start and end at any node in the tree. *(leetcode 124)*

递归：时间复杂度O(n)，空间复杂度O(lgn)

这里先说约定一些叫法：

1 最大路径和: max{树中任意两节点之间的路径和}

2 最大到根路径和: max{树中任意节点到根节点的路径和}

这里我们知道，这个路径肯定是有个拐点，那么这个拐点肯定是某个子树的根节点，所以我们只要递归的求每个子树的过根最大路径和，然后在这些路径和中选出最大的那个就可以了.

过根最大路径和怎么求呢？可以先分别求出左右子树的最大到根路径和，根据这个两个路径和与根节点则可以求出当前树的最大路径和.

```
int dfs(TreeNode *root, int &result){
    if(!root)    return 0;
    if(!root->left && !root->right){
        if(result < root->val)
            result = root->val;
        return root->val > 0? root->val : 0;
    }
    int left = 0, right = 0;
    if(root->left){
        left = dfs(root->left, result);
    }
    if(root->right){
        right = dfs(root->right, result);
    }
    int cur = root->val + left + right;
    if(result < cur)
        result = cur;
    int add = left > right ? left : right;
    if(add < 0)    return root->val > 0? root->val : 0;
    return root->val + add > 0 ? root->val + add : 0;
}

int maxPathSum(TreeNode *root) {
    int result = INT_MIN;
    dfs(root, result);
    return result;
}
```

2.3.9 Sum Root to Leaf Numbers

问题: Given a binary tree containing digits from 0-9 only, each root-to-leaf path could represent a number.An example is the root-to-leaf path 1-¿2-¿3 which represents the number 123.Find the total sum of all root-to-leaf numbers. *(leetcode 129)*
其实这还是属于那种自上而下带有附加信息过来的问题，和最大/小深度问题是一样的.

递归：时间复杂度O(n), 空间复杂度O(lgn)

从上面传来上面路径产生的数字，类似于传来上面路径的深度(最大深度问题)和上面路径的和(Path Sum问题)

```
void dfs(TreeNode *root, int track, int &sum){
    if(!root)    return;
    track = track*10 + root->val;
    if(!root->left && !root->right){
        sum = sum + track;
```

```
6       }
7       if(root->left)
8           dfs(root->left, track, sum);
9       if(root->right)
10          dfs(root->right, track, sum);
11  }
12
13  int sumNumbers(TreeNode *root){
14      int sum = 0;
15      int track = 0;
16      dfs(root, track, sum);
17      return sum;
18  }
```

同样，类似与Path Sum这类问题，可以常用附加路径信息的栈实现迭代，具体做法就是当前节点左右孩子入栈时候当前节点的附加信息值*10加左右孩子节点的值然后作为左右孩子的附加信息值，这里就不再写出.

2.3.10 Least Common Ancest

问题: Given a binary tree, return the least common ancest of two nodes.

递归：时间复杂度O(n), 空间复杂度O(lgn)

```
1   TreeNode * NearestCommAncestor(TreeNode *root, TreeNode *node1, TreeNode
        *node2){
2       if(!root || !node1 || !node2)   return NULL;
3       if(node1 == root || node2 == root)
4           return root;
5       if(!root->left)
6           return NearestCommAncestor(root->right, node1, node2);
7       if(!root->right)
8           return NearestCommAncestor(root->left, node1, node2);
9       TreeNode *left, *right;
10      left = NearestCommAncestor(root->left, node1, node2);
11      right = NearestCommAncestor(root->right, node1, node2);
12      if(left && right)   return root;
13      return left? left : right;
14  }
```

2.4 其他

2.4.1 Flatten Binary Tree to Linked List

问题: Given a binary tree, flatten it to a linked list in-place by the pre-order. *(leetcode 114)*
这是一个基于先序遍历的问题，所以可以使用递归和迭代的方法.

递归：时间复杂度O(n)，空间复杂度O(lgn)

```
void flatten(TreeNode *root) {
    TreeNode *tail;
    recursion(root, tail);
}

TreeNode* recursion(TreeNode *root, TreeNode* &tail){
    if(!root)    return NULL;
    TreeNode *next = NULL;
    tail = root;
    if(root->left)
        next = recursion(root->left, tail);
    if(root->right)
        tail->right = recursion(root->right, tail);
    root->left = NULL;
    if(next)
        root->right = next;
    return root;
}
```

迭代：时间复杂度O(n)，空间复杂度O(lgn)
这里就是完完全全的迭代版前序遍历,这里使用了栈，同样你也可以使用Mirror算法.

```
void flatten(TreeNode *root) {
    if(!root)    return;
    stack<TreeNode*> s;
    s.push(root);
    TreeNode *last = NULL, *cur;
    while(!s.empty()){
        cur = s.top();
        s.pop();
        if(last)
            last->right = cur;
        if(cur->right)
            s.push(cur->right);
        if(cur->left)
            s.push(cur->left);
        cur->left = NULL;
        last = cur;
    }
    last->right = NULL;
}
```

2.4.2 Populating Next Right Pointers in Each Node

问题: Given a binary tree:

```
struct TreeLinkNode {
    TreeLinkNode *left;
```

```
3       TreeLinkNode *right;
4       TreeLinkNode *next;
5   }
```

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL.
Note:
You may only use constant extra space.
You may assume that it is a perfect binary tree (ie, all leaves are at the same level, and every parent has two children). *(leetcode 116)*
其实就是一个很简单的BFS过程.

迭代：时间复杂度O(n), 空间复杂度O(1)
因为是满二叉树，所以每次在上一层建立这一层的next，然后再到这一层来,这样就不需要队列，使用常数的空间复杂度.

```
1   void connect(TreeLinkNode *root) {
2       if(!root)    return;
3       TreeLinkNode *cur = root, *next;
4       cur->next = NULL;
5       while(cur->left){
6           next = cur->left;
7           while(cur){
8               cur->left->next = cur->right;
9               cur->right->next = cur->next? cur->next->left : NULL;
10              cur = cur->next;
11          }
12          cur = next;
13      }
14  }
```

2.4.3 Populating Next Right Pointers in Each Node II

问题: Follow up for problem "Populating Next Right Pointers in Each Node".What if the given tree could be any binary tree? Would your previous solution still work? **Note**: You may only use constant extra space. *(leetcode 117)*

迭代：时间复杂度O(n) , 空间复杂度O(1)
这里其实和上一题一样，只不过多了一些判断条件.

```
1   void connect(TreeLinkNode *root) {
2       if(!root)    return;
3       TreeLinkNode *cur = root, *next, *last;
4       cur->next = NULL;
5       do{
6           next = NULL;
7           while(cur){
8               if(cur->left){
9                   if(next){
10                      last->next = cur->left;
11                      last = last->next;
12                  }
13                  else{
14                      last = cur->left;
15                      next = last;
16                  }
17              }
```

```
18              if(cur->right){
19                  if(next){
20                      last->next = cur->right;
21                      last = last->next;
22                  }
23                  else{
24                      last = cur->left;
25                      next = last;
26                  }
27              }
28              cur = cur->next;
29          }
30          last->next = NULL;
31          cur = next;
32      }while(cur);
33  }
```

2.4.4 Binary Search Tree Iterator

问题:

Implement an iterator over a binary search tree (BST). Your iterator will be initialized with the root node of a BST.

Calling next() will return the next smallest number in the BST.
(leetcode 173)

Note:

next() and hasNext() should run in average O(1) time and uses O(h) memory, where h is the height of the tree.

Stack : 时间复杂度O(1), 空间复杂度O(lgn)

这里使用一个栈来保存上级节点，每个next和hasNext的操作平均时间复杂度是O(1)

```
1   class BSTIterator {
2   private:
3       TreeNode* root;
4       stack<TreeNode*> path;
5   public:
6       BSTIterator(TreeNode *root) : root(root){
7           TreeNode* iter = root;
8           while(iter){
9               path.push(iter);
10              iter = iter->left;
11          }
12      }
13
14      /** @return whether we have a next smallest number */
15      bool hasNext() {
16          return !path.empty();
17      }
18
19      /** @return the next smallest number */
20      int next() {
21          TreeNode *cur = path.top();
22          TreeNode *iter = cur->right;
23          path.pop();
24          while(iter){
```

```
25                 path.push(iter);
26                 iter = iter->left;
27             }
28             return cur->val;
29         }
30 };
```

第 3 章　字符串

3.1　库函数

字符串的库函数有很多,虽然大多数算法上都不是很难,但是需要考虑很多细节问题,所以需要研究一下.关于字符串库函数需要注意的问题大致可以总结为以下几点:

- 输入参数是什么类型,该不该是const修饰
- 输入的指针是否为NULL
- 字符串处理完毕后,新产生的字符串末尾是否要加'\0'
- 要不要返回值,该返回什么类型的(对于要返回值的是为了实现链式操作)

还有一些我觉得可能需要考虑的:

- 有没有地址重叠
- src和dest地址一样
- 需不需要优化做法

3.1.1　strlen

问题: 求出字符串的长度.

Care : 时间复杂度O(n), 空间复杂度O(1)

考虑输入指针不为空,输入参数使用const char

```
1  int strlen(const char *str){
2      int len = 0;
3      assert(str);  //判断不为NULL
4      while(*str++ != '\0'){
5          len++;
6      }
7      return len;
8  }
```

3.1.2　strcat

问题: 将一个字符串连接到另一个字符串后面

Care : 时间复杂度O(n), 空间复杂度O(1)

输入参数const性和非NULL,尾赋'\0', 有返回值

```
char *strcat(char *strDest, const char *strScr){
    assert(strDest && strScr);
    if(!strScr) return strDest;
    char* p = strDest;
    while(*p){
        p++;
    }
    while(*strScr){
        *p++ = *strScr++;
    }
    *p = '\0';
    return strDest;
}
```

3.1.3 strcmp

问题: 比较两个字符串是否完全相同.

Care : 时间复杂度O(n), 空间复杂度O(1)

输入参数const性和非NULL,注意如何写的简洁.

```
int strcmp(const char *str1,const char *str2){
    if(str1 == str2)    return 0;
    assert(str1 && str2);
    //这样写很简洁
    while(*str1 && *str2 && (*str1 == *str2)){
        str1++;
        str2++;
    }
    return (*str1) - (*str2);
}
```

3.1.4 strcpy

问题: 将源字符串完全拷贝给目的字符串.

Care : 时间复杂度O(n), 空间复杂度O(1)

考虑输入参数的const性,输入参数是否合法,返回值,最后位置为'\0'

```
char *strcpy(char *dest,const char *src){
    if(dest == src) return dest;
    assert(dest && src);  //输入参数不为NULL
    char* address = dest;
    int lens = strlen(src);
    int lend = strlen(dest);
    if(src + lens > dest){ // 从后往前拷贝
        dest[lens] = '\0';   //当lens < 就很重要lend
        for(int i = lens - 1; i >= 0; i--){
            dest[i] = src[i];
        }
    }else{ //从前往后拷贝
        for(int i = 0; i < lens; i++){
            dest[i] = src[i];
        }
        dest[lens] = '\0';  //当lens < 就很重要lend
    }
```

```
18      return address; //注意此函数有返回值
19  }
```

至于src和dest地址重叠问题,libc里面并没有考虑,我觉得考虑一下还是很好的.

3.1.5 strncpy

问题: 将源字符串拷贝前n个字符给目的字符串.

Care : 时间复杂度O(n), 空间复杂度O(1)
注意事项同strcpy

```
1  char *strncpy(char *dest,const char *src, int n){
2      assert(dest && src);
3      char *address = dest;
4      int lens = strlen(src);
5      int lend = strlen(dest);
6      if(src + lens > dest && src + n > dest){//从后往前拷贝
7          dest[min(lens, n)] = '\0';
8          for(int i = min(lens, n) - 1; i >= 0; i--){
9              dest[i] = src[i];
10         }
11     }else{ //从前往后拷贝
12         for(int i = 0; i < lens && i < n; i++){
13             dest[i] = src[i];
14         }
15         dest[min(lens, n)] = '\0';
16     }
17     return address;
18 }
```

注意不能仅仅因为src == dest就放弃拷贝.

3.1.6 memcpy

问题: 将源地址开始的连续n个字节大小空间拷贝给给目的地址.

Care : 时间复杂度O(n), 空间复杂度O(1)
注意输入参数是void*和const void*, 输出参数是void*,要保证输出参数非NULL

```
1  void *memcpy(void *dest, const void *src, size_t count){
2      if(dest == src) return dest;
3      assert(src && dest);
4      if(!src || !dest)    return NULL;
5      unsigned char *d = (unsigned char*)dest, *s = (unsigned char*)src;
6      while(count--){
7          *d++ = *s++;
8      }
9      //不需要设置因为是内存拷贝'\0',
10     return dest;
11 }
```

libc使用了page copy, unsigned long copy做到快速拷贝

3.2 字符串经典问题

字符串的经典问题有很多,虽然很多解法都属于算法范畴,诸如动态规划等,这里还是归结为字符串的问题.

3.2.1 strStr

问题: Implement strStr(). Returns the index of the first occurrence of needle in haystack, or -1 if needle is not part of haystack. *(leetcode 28)*

KMP : 时间复杂度O(n), 空间复杂度O(m)

strstr属于字符串的库函数,放在这里讲完全是因为它是字符串问题中最有名的之一,其解法多样,这里推荐KMP解法,关于此解法的介绍可以参考我们ThreeCobblers主页上zhuoyuan的博文

```
1  void gen_next(const char *p) {
2      next[0] = -1;
3      int i = 0;
4      int j = -1;
5      int lp = strlen(p);
6      while(i < lp)
7          if(j == -1 || p[i] == p[j]) i++, j++, next[i] = j;
8          else j = next[j];
9  }
10 int kmp(const char *s, const char *p) {
11     gen_next(p);
12     int ls = strlen(s);
13     int lp = strlen(p);
14     int i = -1;
15     int j = -1;
16     while(i < ls && j < lp)
17         if(j == -1 || s[i] == p[j]) i++, j++;
18         else j = next[j];
19     if(j == lp) return i - lp;
20     return -1;
21 }
```

这段代码水很深,写的非常老道,需要好好揣测才可以明白,关于求*next*数组问题可以看上面说的那篇博客的介绍

3.2.2 atoi

问题: Implement atoi to convert a string to an integer. *(leetcode 8)*

Note: The function first discards as many whitespace characters as necessary until the first non-whitespace character is found. Then, starting from this character, takes an optional initial plus or minus sign followed by as many numerical digits as possible, and interprets them as a numerical value.

The string can contain additional characters after those that form the integral number, which are ignored and have no effect on the behavior of this function.

If the first sequence of non-whitespace characters in str is not a valid integral number, or if no such sequence exists because either str is empty or it contains only whitespace characters, no conversion is performed.

If no valid conversion could be performed, a zero value is returned. If the correct value is out of the range of representable values, INT_MAX (2147483647) or INT_MIN (-2147483648) is returned.

Care：时间复杂度O(n), 空间复杂度O(1)

这个问题主要要注意两个方面,一个是空格和非法字符,一个是溢出,空格和非法字符都很好处理,溢出的处理方式就值得去研究,在C++中INT_MAX的绝对值比INT_MIN小1,所以负数个数比正数个数多一个,你可以都转化位负数来判断溢出. 当然,对付溢出还有一种偷懒的方法就是使用double类型取存储数据.

```
int atoi(const char *str){
    int sum = 0;  // 存负值.
    bool isMinus = false;
    const char* p = str;
    if(!p)  return 0;
    while(*p == ' ' && *p != '\0') p++; //空格
    if(*p == '\0')  return 0;
    if(*p == '-'){
        isMinus = true;
        p++;
    }else{
        if(*p == '+') p++;
    }
    while(*p != '\0'){
        int cur = *p++ - '0';
        if(!(cur >= 0 && cur <= 9)) break;  //非法字符
        if(isMinus && sum == INT_MIN/10 && cur > INT_MAX%10 + 1){
            return INT_MIN;
        }
        if(isMinus && sum < INT_MIN/10){
            return INT_MIN;
        }
        if(!isMinus && sum == -INT_MAX/10 && cur > INT_MAX%10){
            return INT_MAX;
        }
        if(!isMinus && sum < -INT_MAX/10){
            return INT_MAX;
        }
        sum = sum*10 - cur;
    }
    if(!isMinus)    return -sum;
    return sum;
}
```

3.2.3 Reverse Words in a String

问题: Given an input string, reverse the string word by word.*(leetcode 151)*

举例:

Given s = "the sky is blue",
return "blue is sky the".

反转：时间复杂度O(n), 空间复杂度O(n)

这也是一道经典的字符串题目,题目的要求可以通过先把整个字符串反转一下,然后从前到后spilit,每spilit一个词就反转这个词然后添加到后面,即全局反转和逐个反转.

```
void reverseWords(string &s) {
    reverse(s.begin(), s.end());
    int i = 0, last;
    string str, word;
    while(i < s.length()){
```

```
6           while(i < s.length() && s[i] == ' ') i++;
7           if(i == s.length()) break;
8           last = i;
9           while(i < s.length() && s[i] != ' ') i++;
10          word = s.substr(last, i - last);
11          reverse(word.begin(), word.end());
12          str = str + word + " ";
13      }
14      if(!str.empty())
15          s.assign(str.begin(), str.end() - 1);
16      else
17          s = str;
18  }
```

3.2.4 Valid Palindrome

问题: Given a string, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases.*(leetcode 125)*

举例: "A man, a plan, a canal: Panama" is a palindrome. "race a car" is not a palindrome.

Care : 时间复杂度O(n) , 空间复杂度O(1)

```
1   bool isAlpha(char c){
2       if(c <= 'Z' && c >= 'A') return true;
3       if(c <= 'z' && c >= 'a') return true;
4       if(c <= '9' && c >= '0') return true;
5       return false;
6   }
7
8   bool isPalindrome(string s) {
9           int len = s.length();
10          int start = 0, end = len - 1;
11          while(start < end){
12              while(start < end && !isAlpha(s[start]))    start++;
13              if(start == end)    return true;
14              while(end > start && !isAlpha(s[end]))  end--;
15              if(s[start] != s[end] && abs(s[start] - s[end]) != abs('a' -
                    'A')) return false;
16              start++;
17              end--;
18          }
19          return true;
20      }
```

3.2.5 LongestPalindrome

问题: Longest Palindromic Substring. *(leetcode 5)*

Manacher : 时间复杂度O(n), 空间复杂度O(n)

这里介绍的是Manacher算法,关于此算法的思想和证明可以参考我们ThreeCobblers主页上sosohu的博文

```
1   string longestPalindrome(string const& s) {
2           int n = s.length();
3           if(n == 0)  return "";
```

```
        string str = "#";
        int count = 0;
        for(int i = 0; i < n; i++){
            str += s[i];
            str += '#';
        }
        int id = 0, mx = 0;
        vector<int> p(2*n+1, 0);
        p[0] = 1;
        for(int i = 1; i < 2*n + 1; i++){
            int j = 2*id - i;
            p[i] = mx > i? min(p[j], mx - i) : 1;
            while(i + p[i] < 2*n + 1 && i - p[i] >= 0){
                if(str[i + p[i]] != str[i - p[i]]) break;
                p[i]++;
            }
            if(i + p[i] > mx){
                mx = i+ p[i];
                id = i;
            }
        }
        int max = INT_MIN;
        int pos = 0;
        for(int i = 0; i < 2*n + 1; i++){
            if(max < p[i]){
                max = p[i];
                pos = i;
            }
        }
        int index, len;
        len = (max - 1);
        index = pos/2 - len/2;
        return s.substr(index, len);
    }
```

3.2.6 Longest Substring Without Repeating Characters

问题: Given a string, find the length of the longest substring without repeating characters. For example, the longest substring without repeating letters for "abcabcbb" is "abc", which the length is 3. For "bbbbb" the longest substring is "b", with the length of 1. *(leetcode 3)*

DP : 时间复杂度O(n) , 空间复杂度O(1)

这道题其实和最大连续和问题是一个类型的问题,都是属于一维的动态规划,说它是动态规划是因为父问题可以通过子问题来构造,我们是d[i]表示字符串s[1...i]中的最长无重复串的长度,c[i]表示s[1...i]中以s[i]为结尾的最长无重复子串的长度, 例如s[1...5] = "abcdb", 那么d[5] = 4, c[5] = 3.

所以我们有递推关系:

$$d[i+1] = max(d[i], s_{i+1} \in s[i - c[i] + 1...i])?d[i] : c[i] + 1)$$

根据这个递推关系式,我们可以扫描字符串,边扫描边统计d[i]和更新c[i],最终求得答案.

```
int lengthOfLongestSubstring(string s) {
    vector<bool> appear(256, false);
    int ret = 0, con = 0;
    for(int i = 0; i < s.length(); i++){
        if(!appear[s[i]]){
```

```
6               con++;
7               appear[s[i]] = true;
8               ret = max(ret, con);
9           }else{
10              for(int j = i - con; j < i && s[j] != s[i]; j++){
11                  appear[s[j]] = false;
12                  con--;
13              }
14          }
15      }
16      return ret;
17  }
```

这类问题很常见,像最大连续和,最长递增子串等等,它们的共同点就是连续而且可以很方便的使用新加入的元素来更新最长靠右连续最优解*(*本题的*c[i]),*父问题可以很容易的通过子问题求出.

3.2.7 Anagrams

问题: Given an array of strings, return all groups of strings that are anagrams.

Note: All inputs will be in lower-case. *(leetcode 49)*

sort : 时间复杂度O(n*m) , 空间复杂度O(n*m)

变位词是字符串的一种常见问题,很多时候都是先把一群互为变位词的词选出一个为它们的索引词(key),然后就可以把它们放在一起存储.这道题如果不是先这样,而是一个一个查找那么是非常低效的.

```
1   vector<string> anagrams(vector<string> &strs) {
2       int size = strs.size();
3       unordered_map<string, vector<string> > table;
4       for(int i = 0; i < size; i++){
5           string index = strs[i];
6           sort(index.begin(), index.end());
7           table[index].push_back(strs[i]);
8       }
9       vector<string> ret;
10      for(unordered_map<string, vector<string> >::iterator iter = table.
            begin();
11          iter != table.end(); iter++){
12          if(iter->second.size() > 1){
13              ret.insert(ret.end(), iter->second.begin(), iter->second.end
                    ());
14          }
15      }
16  return ret;
17  }
```

3.2.8 Valid Number

问题: Validate if a given string is numeric.

举例 ”0” is true ” 0.1 ” is true ”abc” is false ”1 a” is false ”2e10” is true

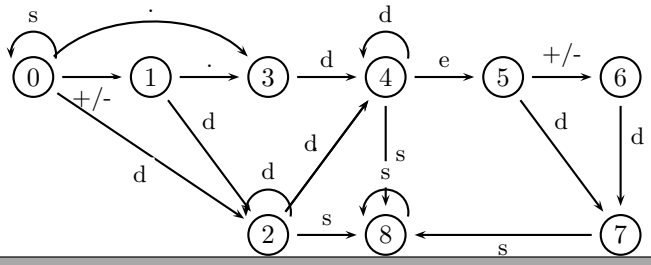
NFA : 时间复杂度O(n), 空间复杂度O(1)

这种问题最优美的解法就是先写出正则表达式,然后根据正则表达式画出NFA,然后根据NFA的

状态转移写出代码.

正则表达式: $s^*(+|-)?((d^+.?)|(.d))d^*(e(+|-)?d^+)?$ **s为空格,d为数字**

画出状态转移图如下:




```
enum lexical{ valid, space, sign, number, dot, e };

int isType(const char c){
    switch(c){
        case ' ': return 1;
        case '+': ;
        case '-': return 2;
        case '.': return 4;
        case 'e': return 5;
        default: if(c <= '9' && c >= '0')   return 3;
                 else return 0;
    }
}

bool isNumber(const char *s) {
    if(!s)  return false;
    //状态转移矩阵, 表示非法-1
    int map[9][6] = {
        -1, 0, 1, 2, 3, -1,  // 状态的转移0
        -1, -1, -1, 2, 3, -1,
        -1, 8, -1, 2, 4, 5,
        -1, -1, -1, 4, -1, -1,
        -1, 8, -1, 4, -1, 5,
        -1, -1, 6, 7, -1, -1,
        -1, -1, -1, 7, -1, -1,
        -1, 8, -1, 7, -1, -1,
        -1, 8, -1, -1, -1, -1
    };
    int state = 0;
    while(*s){
        state = map[state][isType(*s)];
        if(state == -1) return false;
        s++;
    }
    return state == 2 || state == 4 || state == 7 || state == 8;
}
```

知道NFA转移图后想到像上面这些编写代码也是蛮难的,这样编写的好处是可以灵活的改变NFA,不会对代码构成很大影响,值得学习.

3.2.9 Regular Expression Matching

问题:

Implement regular expression matching with support for '.' and '*'.

'.' Matches any single character.
'*' Matches zero or more of the preceding element.
The matching should cover the entire input string (not partial).
(leetcode 10)

举例:
isMatch("aa","a") → false
isMatch("aa","aa") → true
isMatch("aaa","aa") → false
isMatch("aa", "a*") → true
isMatch("aa", ".*") → true
isMatch("ab", ".*") → true
isMatch("aab", "c*a*b") → true

DP : 时间复杂度O(n*m), 空间复杂度O(n*m)
本题使用动态规划求解,设isMatch[i][j]表示s[1...i]与p[1...j]是否匹配
递推关系是如下:

$$isMatch[i][j] = \begin{cases} 1 & i=0, j=0 \\ 0 & i \neq 0, j=0 \\ isMatch[i-1][j-1] \;\&\&\; (s_{i-1} == p_{j-1} \;||\; p_{j-1} == '.') & p_j \neq '*' \\ isMatch[i][j] = isMatch[i][j-2] \;||\; isMatch[i-1][j] & p_j = '*', p_{j-1} = '.' \\ isMatch[i][j-1] & p_j = '*', p_{j-1} = '*' \\ isMatch[i][j-2] \;||\; (s_{i-1} == p_{j-1} \;\&\&\; isMatch[i-1][j]) & other \end{cases}$$

```
1   bool isMatch(const char *s, const char *p) {
2       int ls = strlen(s);
3       int lp = strlen(p);
4       vector<vector<bool> > isMatch(ls+1, vector<bool>(lp+1, false));
5       isMatch[0][0] = true;
6       int di, dj;
7       for(int i = 0; i < ls + 1; i++){ // 起始坐标
8           di = i - 1;
9           for(int j = 1; j < lp + 1; j++){ // 起始坐标
10              dj = j - 1;
11              if(p[dj] != '*'){
12                  isMatch[i][j] = di != -1 && isMatch[i-1][j-1] && (s[
                        di] == p[dj] || p[dj] == '.');
13              }else{
14                  if(dj == 0) { isMatch[i][j] = false; continue; }
15                  if(p[dj-1] == '.'){
16                          isMatch[i][j] = isMatch[i][j-2] || (di != -1
                                && isMatch[i-1][j]);
17                  }else{
18                      if(p[dj-1] == '*')  isMatch[i][j] = isMatch[i][j
                            -1];
19                      else isMatch[i][j] = isMatch[i][j-2] || (di !=
                            -1 && s[di] == p[dj-1] && isMatch[i-1][j]);
20                  }
21              }
22          }
23      }
24      return isMatch[ls][lp];
25  }
```

3.2.10 Wildcard Matching

问题:

Implement wildcard pattern matching with support for '?' and '*'. '?'
Matches any single character. '*' Matches any sequence of characters (including the empty sequence).
The matching should cover the entire input string (not partial). *(leetcode 44)*

举例:

isMatch("aa","a") → false
isMatch("aa","aa") → true
isMatch("aaa","aa") → false
isMatch("aa", "*") → true
isMatch("aa", "a*") → true
isMatch("ab", "?*") → true
isMatch("aab", "c*a*b") → false

迭代: 时间复杂度O(n*m) , 空间复杂度O(1)

这道题其实是可以套用Regular Expression Matching的解法,使用动态规划来计算,但是那样在leetcode上会超时,具体原因我还不清楚. 本题的解法其实也很明了,唯一需要注意的就是每次我们遇到*时候都会记下此次*的位置以及主串位置,然后一次一次的试探*是否要生成字母,试探不成功就回到刚才我们记下的状态然后试探下一个状态,但是,当我们遇到下一个*时候,就可以更新这个记录点了,因为都已经到这个*了,上个*走到这个*走的路肯定是对的,即使不太对,也可以通过这个*不断生成字母来弥补,这就是需要注意的地方,不是很好理解,需要仔细琢磨.

```
bool isMatch(const char *s, const char *p){
    int ls = strlen(s);
    int lp = strlen(p);
    const char *ps = s, *pp = p, *lasts = NULL, *lastp = NULL;
    if(!s || !p)    return false;
    while(*s){
        switch(*p){
            case '?':   s++; p++; break;
            case '*':   while(*(p+1) && *(p+1) == '*')  p++;
                        lasts = s; lastp = p; p++; break;
            default:    if(*s == *p){ s++; p++;}
                        else{
                            if(lasts){
                                s = ++lasts;
                                p = lastp + 1;
                            }else{
                                return false;
                            }
                        }
        }
    }
    if(*p == '\0' && *s == '\0')    return true;
    if((*p) == '\0' && *(p-1) != '*')   return false;
    while(*p && *p == '*'){
        p++;
    }
    if(*p != '\0')  return false;
    return true;
}
```

3.2.11 Edit Distance

问题:

Given two words word1 and word2, find the minimum number of steps required to convert word1 to word2. (each operation is counted as 1 step.)
You have the following 3 operations permitted on a word:
a) Insert a character
b) Delete a character
c) Replace a character
(leetcode 71)

DP : 时间复杂度O(n*m), 空间复杂度O(n*m)

两个字符串的编辑距离是一个很经典的问题,在信息检索领域有着重要的作用,它的解法需要使用动态规划来求解

我们设$d_{i,j}$表示字符串s[1...i]和p[1...j]的编辑距离,w_i, w_d, w_r分别表示插入,删除和替换的操作代价(在本题都为1),那么我们可以得到递推关系式:

$$d_{i,j} = \begin{cases} j & i = 0 \\ i & j = 0 \\ min\{d_{i-1,j} + w_i, d_{i,j-1} + w_d, d_{i-1,j-1} + (s_i == p_j ? 0 : w_r)\} & other \end{cases}$$

```
int minDistance(string word1, string word2) {
    int l1 = word1.length(), l2 = word2.length();
    vector<vector<int> > d(l1+1, vector<int>(l2+1, 0));
    for(int i = 0; i < l1 + 1; i++) d[i][0] = i;
    for(int j = 0; j < l2 + 1; j++) d[0][j] = j;

    for(int i = 1; i < l1 + 1; i++){
        for(int j = 1; j < l2 + 1; j++){
            if(word1[i-1] == word2[j-1])    d[i][j] = d[i-1][j-1];
            else    d[i][j] = d[i-1][j-1] + 1;
            d[i][j] = min(d[i][j], d[i-1][j] + 1);
            d[i][j] = min(d[i][j], d[i][j-1] + 1);
        }
    }

    return d[l1][l2];
}
```

3.2.12 Minimum Window Substring

问题:

Given a string S and a string T, find the minimum window in S which will contain all the characters in T in complexity O(n).*(leetcode 76)*

举例:

S = "ADOBECODEBANC"
T = "ABC"
Minimum window is "BANC".

Note:

If there is no such window in S that covers all characters in T, return the emtpy string "".
If there are multiple such windows, you are guaranteed that there will always be only one unique minimum window in S.

Hash : 时间复杂度O(n), 空间复杂度O(1)

这道题很容易想到用哈希表来求解,但是对于这个存在重复多的key,我一开始想到的是哈希的value存储一个list来放这些重复元素,后来发现对于这道题,其实你只要记录次数就行了,每次准备向右移动窗口时候,只要要被移掉的字符它的hash值大于原数目就说明是多的,可以移除的,所以这题就简单开朗很多了.

```
string minWindow(string S, string T) {
    vector<int> table(256, 0);
    vector<int> apper(256, 0);
    for(int i = 0; i < T.length(); i++) table[T[i]]++;
    int len = INT_MAX, num = 0, last = 0, pos = 0;
    for(int i = 0; i < S.length(); i++){
        apper[S[i]]++;
        if(apper[S[i]] <= table[S[i]]) num++;
        while(apper[S[last]] > table[S[last]]){
            apper[S[last]]--;
            last++;
        }
        if(num == T.length() && len > i - last + 1){
            len = i - last + 1;
            pos = last;
        }
    }
    return num == T.length()? S.substr(pos, len) : "";
}
```

在字符串很多问题中都需要用到哈希表,特别是一个主串一个副串,在主串找副串的元素这类问题.

3.3 字符串其他问题

3.3.1 Integer to Roman

问题: Given an integer, convert it to a roman numeral. Input is guaranteed to be within the range from 1 to 3999. *(leetcode 12)*

Trick : 时间复杂度O(1), 空间复杂度O(1)

这道题其实比较简单,但是怎么写的优雅是一个很难的事,下面的代码就很trick,需要记住.

```
string  intToRoman(int num) {
        string ret;
        int stand[] = {1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5,
            4, 1};
        string symbol[] = {"M", "CM", "D", "CD", "C", "XC", "L", "XL", "
            X", "IX", "V", "IV", "I"};
        for(int i = 0; num > 0; i++){
            int count = num / stand[i];
            num = num % stand[i];
            for(; count > 0; count--) ret += symbol[i];
        }
        return ret;
    }
```

写的实在是太简洁,太美,太富有穿透力了,应该当做教科书一样背下来.

3.3.2 Roman to Integer

问题: Given a roman numeral, convert it to an integer.Input is guaranteed to be within the range from 1 to 3999. *(leetcode 13)*

Care : 时间复杂度O(1), 空间复杂度O(1)

同样,怎么写的优雅是这题的最难的地方.

```
int mapNum(char c){
    switch(c){
        case 'I': return 1;
        case 'V': return 5;
        case 'X': return 10;
        case 'L': return 50;
        case 'C': return 100;
        case 'D': return 500;
        case 'M': return 1000;
        default: return 0;
    }
}

int romanToInt(string s) {
    int len = s.length();
    int sum = 0;
    for(int i = 0; i < len; i++){
        if(i < len - 1 && mapNum(s[i]) < mapNum(s[i+1]))
                sum -= mapNum(s[i]);
        else    sum += mapNum(s[i]);
    }
    return sum;
}
```

3.3.3 Longest Common Prefix

问题: Write a function to find the longest common prefix string amongst an array of strings. *(leetcode 14)*

Care : 时间复杂度O(n*m) , 空间复杂度O(1)

这道题比较简单,需要注意的还是怎么写的简洁优雅.

```
string longestCommonPrefix(vector<string> &strs) {
    string ret;
    if(strs.size() == 0)    return ret;
    int len = INT_MAX;
    for(int i = 0; i < strs.size(); i++)
        if(len > strs[i].length())  len = strs[i].length();
    for(int j = 0; j < len; j++){
        for(int i = 1; i < strs.size(); i++)
            if(strs[i][j] != strs[i-1][j])  return ret;
        ret.push_back(strs[0][j]);
    }
    return ret;
}
```

3.3.4 Count and Say

问题:

The count-and-say sequence is the sequence of integers beginning as follows:
1, 11, 21, 1211, 111221, ...
1 is read off as "one 1" or 11.
11 is read off as "two 1s" or 21.
21 is read off as "one 2, then one 1" or 1211.
Given an integer n, generate the nth sequence.*(leetcode 38)*

NoteThe sequence of integers will be represented as a string.

递推 : 时间复杂度O(n*m) , 空间复杂度O(m)

根据递推关系一步步的推.

```
string parse(string& str){
    string ret;
    int last = 0, len = str.length();
    for(int i = 0; i < len; i++){
        last = i;
        while(i < len - 1 && str[i] == str[i+1])    i++;
        ret.push_back(i - last + 1 + '0');
        ret.push_back(str[last]);
    }
    return ret;
}

string countAndSay(int n) {
    string last = "1", cur;
    if(n < 1)   return cur;
    while(n-- > 1){
        last = cur = parse(last);
    }
    return last;
}
```

3.3.5 Add Binary

问题: Given two binary strings, return their sum (also a binary string).*(leetcode 67)*

举例:

a = "11"
b = "1"
Return "100".

迭代: 时间复杂度O(n), 空间复杂度O(1)

和链表的那个完全类似,解法也是基本一致,优雅的代码.

```
string addBinary(string a, string b) {
    int la = a.length(), lb = b.length();
    int carry = 0, sum = 0;
    string ret;
    while(la || lb || carry){
        sum = carry + (la > 0? a[la-1] - '0' : 0) + (lb > 0? b[lb-1] - '
            0' : 0);
        ret.push_back(sum%2 + '0');
        carry = sum/2;
        if(la)  la--;
        if(lb)  lb--;
    }
    reverse(ret.begin(), ret.end());
    return ret;
}
```

3.3.6 ZigZag Conversion

问题:

The string "PAYPALISHIRING" is written in a zigzag pattern on a given number of rows like this: (you may want to display this pattern in a fixed font for better legibility)

P A H N
A P L S I I G
Y I R
And then read line by line: "PAHNAPLSIIGYIR"
Write the code that will take a string and make this conversion given a number of rows:

string convert(string text, int nRows);
convert("PAYPALISHIRING", 3) should return "PAHNAPLSIIGYIR".
(leetcode 6)

数学推导: 时间复杂度O(n), 空间复杂度O(1)

这题就是简单的推导一下每一行相邻两个位置的间隔,然后编写代码.

```
string convert(string s, int nRows) {
    if(nRows == 1)  return s;
    int len = s.length();
    string ret;
    for(int i = 0; i < nRows; i++){
        int start = i, gap = nRows - i - 1;
        while(start < len){
```

```
                ret.push_back(s[start]);
                if(!gap) gap = nRows - 1 - gap;
                start = start + 2*gap;
                gap = nRows - 1 - gap;
            }
        }
        return ret;
    }
```

这段代码写的很简洁,每一行都有一个自己的*gap*表示这一行相邻两个位置的间隔,除了第一行和最后一行*gap*值一直都会反转,与实际的推导结果一致.

3.3.7 Length of Last Word

问题: Given a string s consists of upper/lower-case alphabets and empty space characters ' ', return the length of last word in the string. If the last word does not exist, return 0.*(leetcode 58)*

举例

Given s = "Hello World",
return 5.

Note A word is defined as a character sequence consists of non-space characters only.

Care : 时间复杂度O(n) , 空间复杂度O(1)

这题其实就是套用spilit操作,过程是一模一样的.

```
int lengthOfLastWord(const char *s) {
    int len = 0;
    if(!s)  return len;
    while(*s){
        while(*s && *s == ' ') s++;
        if(*s == '\0')  return len;
        len = 0;
        while(*s && *s != ' ') {s++; len++;}
    }
    return len;
}
```

3.3.8 Text Justification

问题:

Given an array of words and a length L, format the text such that each line has exactly L characters and is fully (left and right) justified.
You should pack your words in a greedy approach; that is, pack as many words as you can in each line. Pad extra spaces ' ' when necessary so that each line has exactly L characters.
Extra spaces between words should be distributed as evenly as possible. If the number of spaces on a line do not divide evenly between words, the empty slots on the left will be assigned more spaces than the slots on the right.
For the last line of text, it should be left justified and no extra space is inserted between words.*(leetcode 68)*

Note:

Each word is guaranteed not to exceed L in length.
A line other than the last line might contain only one word. What should you do in this case?
In this case, that line should be left-justified.

Care：时间复杂度O(n)，空间复杂度O(L)

这题比较繁琐,需要考虑很多边界情况,弄清楚题目的所有边界情况很重要,代码写的也是根琐碎.

```
vector<string> fullJustify(vector<string> &words, int L) {
    vector<string> ret;
    if(L < 1 || !words.size()){
        string str(L, ' ');
        ret.push_back(str);
        return ret;
    }
    int last = 0, cur = 0, num = 0, left = 0;
    while(cur < words.size()){
        int sum = 0;
        last = cur;
        while(cur < words.size() && sum + words[cur].length() <= L){
            sum = sum + words[cur].length() + 1;
            cur++;
        }
        num = cur - last; // 词数
        sum = sum - num; // 个词总长num
        left = L - sum; // 余下空格长度
        if(num == 1){//一行只有一个词
            string margin(left, ' ');
            words[last] = words[last] + margin;
            ret.push_back(words[last]);
            continue;
        }
        if(cur == words.size()){//最后一行
            string line, s(1, ' ');
            for(int i = 0; i < num - 1; i++){
                line = line + words[last++] + s;
            }
            line = line + words[last++];
            string margin(left - num + 1, ' ');
            line = line + margin;
            ret.push_back(line);
            continue;
        }
        int gap = left / (num - 1);
        int remain = left % (num - 1);
        string line, s1(gap + (remain == 0? 0 : 1), ' '), s2(gap, ' ');
        for(int i = 0; i < remain; i++){
            line = line + words[last++] + s1;
        }
        for(int i = remain; i < num - 1; i++){
            line = line + words[last++] + s2;
        }
        line += words[last];
        ret.push_back(line);
    }
    return ret;
}
```

3.3.9 Compare Version Numbers

问题:

Compare two version numbers version1 and version2.

If version1 ¿ version2 return 1, if version1 ¡ version2 return -1, otherwise return 0.

You may assume that the version strings are non-empty and contain only digits and the . character.
The . character does not represent a decimal point and is used to separate number sequences. For instance, 2.5 is not "two and a half" or "half way to version three", it is the fifth second-level revision of the second first-level revision.
(leetcode 165)

举例:

Here is an example of version numbers ordering:
$0.1 < 1.1 < 1.2 < 13.37$

??? : 时间复杂度O(n), 空间复杂度O(1)

这题主要难度就是字符串的处理，扫描出合法的数，然后逐个比较.

```
int compareVersion(string version1, string version2) {
    int i = 0, j = 0, n1 = version1.size(), n2 = version2.size();
    if(n1 == 0 && n2 == 0)  return 0;
    if(n1 == 0 || n2 == 0)
        return n1 == 0? compareVersion("0", version2) : compareVersion(
            version1, "0");
    while(i < n1 && version1[i] == '.') i++;
    while(j < n2 && version2[j] == '.') j++;
    if(i == n1 && j == n2)  return 0;
    if(i == n1 || j == n2)  return i == n1? -1 : 1;
    int ei = i, ej = j;
    while(ei < n1 && version1[ei] != '.')   ei++;
    while(ej < n2 && version2[ej] != '.')   ej++;
    int v1 = atoi(version1.substr(i, ei - i).c_str());
    int v2 = atoi(version2.substr(j, ej - i).c_str());
    if(v1 == v2)
        return compareVersion(version1.substr(ei, n1 - ei), version2.
            substr(ej, n2 - ej));
    if(v1 < v2) return -1;
    else return 1;
}
```

第 4 章　数组

4.1　变长数组

C++中变长数组就是vector,关于它的实现可以参考我的博客.

4.2 经典问题

4.2.1 Remove Duplicates from Sorted Array

问题:

Given a sorted array, remove the duplicates in place such that each element appear only once and return the new length.

Do not allocate extra space for another array, you must do this in place with constant memory.
(leetcode 26)

举例:

Given input array nums = $[1, 1, 2]$,

Your function should return length = 2, with the first two elements of nums being 1 and 2 respectively. It doesn't matter what you leave beyond the new length.

Two-Pointer : 时间复杂度O(n), 空间复杂度O(1)

维持两个指针(以下代码中分别是i和pos)，一个指向要处理的元素，一个指向有效元素的存储位置

```
int removeDuplicates(vector<int>& nums) {
    int pos = 0;
    for(int i = 0; i < nums.size(); i++){
        if(!(i < nums.size() - 1 && nums[i] == nums[i+1]))
            nums[pos++] = nums[i];
    }
    nums.resize(pos);
    return pos;
}
```

4.2.2 Remove Duplicates from Sorted Array II

问题:

Follow up for "Remove Duplicates":
What if duplicates are allowed at most twice?
(leetcode 80)

举例:

Given sorted array nums = $[1, 1, 1, 2, 2, 3]$,

Your function should return length = 5, with the first five elements of nums being 1, 1, 2, 2 and 3. It doesn't matter what you leave beyond the new length.

Two Pointer : 时间复杂度O(n), 空间复杂度O(1)

```
int removeDuplicates(vector<int>& nums) {
    int pos = 0;
    for(int i = 0; i < nums.size(); i++){
        nums[pos++] = nums[i];
        if(i < nums.size() - 1 && nums[i] == nums[i+1]){
            while(i < nums.size() - 1 && nums[i] == nums[i+1])
                i++;
            nums[pos++] = nums[i];
        }
    }
```

```
11      nums.resize(pos);
12      return pos;
13  }
```

4.2.3 Remove Element

问题:

Given an array and a value, remove all instances of that value in place and return the new length.

The order of elements can be changed. It doesn't matter what you leave beyond the new length.
(leetcode 27)

Two-Pointer : 时间复杂度O(n), 空间复杂度O(1)

与Remove Duplicate of Sort Array类似

```
1   int removeElement(vector<int>& nums, int val) {
2       int pos = 0;
3       for(int i = 0; i < nums.size(); i++){
4           if(nums[i] != val)  nums[pos++] = nums[i];
5       }
6       nums.resize(pos);
7       return pos;
8   }
```

4.2.4 First Missing Positive

问题:

Given an unsorted integer array, find the first missing positive integer.
Your algorithm should run in O(n) time and uses constant space.
(leetcode 41)

举例:

Given $[1, 2, 0]$ return 3,
and $[3, 4, -1, 1]$ return 2.

: 时间复杂度 , 空间复杂度O

这题题目的要求其实就是最大的提示了，即要求O(n)时间复杂度,又要求O(1)空间复杂度，那么思来想去也只有践踏原有数组这一个办法了，至于如何践踏,我们发现数组中元素的位置分别是0,1,2,3,...正好可以一一对应1,2,3,4....于是就可以设定对于元素放在对应位置

```
1   int firstMissingPositive(vector<int>& nums) {
2       int pos = 0;
3       while(pos < nums.size()){
4           if(nums[pos] == pos + 1 || nums[pos] > nums.size()
5               || nums[pos] <= 0  || nums[nums[pos] - 1] == nums[pos])
6               pos++;
7           else swap(nums[pos], nums[nums[pos] - 1]);
8       }
9       for(int i = 0; i < nums.size(); i++)
10          if(nums[i] != i+1)  return i + 1;
11      return nums.size() + 1;
12  }
```

4.2.5 Rotate Image

问题:

You are given an n x n 2D matrix representing an image.
Rotate the image by 90 degrees (clockwise).
(leetcode 48)

Follow Up:

Could you do this in-place?

翻转：时间复杂度O(n^2), 空间复杂度O(1)

这题是编程珠玑中经典问题，可以采用对角翻转和折半翻转的组合完成旋转

```
void rotate(vector<vector<int> > &matrix) {
    int n = matrix.size();
    for(int i = 0; i < n; i++){
        reverse(matrix[i].begin(), matrix[i].end());
    }
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n - 1 - i; j++){
            swap(matrix[i][j], matrix[n-1-j][n-1-i]);
        }
    }
}
```

4.2.6 Rotate Array

问题:

Rotate an array of n elements to the right by k steps.
(leetcode 189)

举例:

With n = 7 and k = 3, the array $[1, 2, 3, 4, 5, 6, 7]$ is rotated to $[5, 6, 7, 1, 2, 3, 4]$.

翻转：时间复杂度O(n) , 空间复杂度O(1)

这题是编程珠玑中的经典例题，可以通过三次翻转实现shift动作

```
void rotate(vector<int>& nums, int k) {
    if(nums.size() == 0 || (k = k % nums.size()) == 0)
        return;
    reverse(nums.begin(), nums.end() - k);
    reverse(nums.end() - k, nums.end());
    reverse(nums.begin(), nums.end());
}
```

4.3 相关问题

4.3.1 Spiral Matrix

问题:

Given a matrix of m x n elements (m rows, n columns), return all elements of the matrix in spiral order.
(leetcode 54)

举例:

Given the following matrix:

[
[1, 2, 3],
[4, 5, 6],
[7, 8, 9]
]
You should return $[1, 2, 3, 6, 9, 8, 7, 4, 5]$.

??? : 时间复杂度O(n^2), 空间复杂度O(1)
从外到内一环一环的处理，需要注意一些边界条件

```
vector<int> spiralOrder(vector<vector<int> > &matrix) {
    vector<int> result;
    int n = matrix.size();
    if(n == 0)  return result;
    int m = matrix[0].size();
    int magrin = 0;
    while(m - 1 - magrin >= magrin && n - 1 - magrin >= magrin){
        for(int j = magrin; j <= m - 1 - magrin; j++)
            result.push_back(matrix[magrin][j]);
        for(int i = magrin + 1; i < n - 1 - magrin; i++)
            result.push_back(matrix[i][m-1-magrin]);
        if(n - 1 - magrin != magrin)
            for(int j = m - 1 - magrin; j >= magrin; j-- )
                result.push_back(matrix[n-1-magrin][j]);
        if(m - 1 - magrin != magrin)
            for(int i = n - 1 - magrin - 1; i > magrin; i--)
                result.push_back(matrix[i][magrin]);
        magrin++;
    }
    return result;
}
```

4.3.2 Spiral Matrix II

问题:

Given an integer n, generate a square matrix filled with elements from 1 to n^2 in spiral order.
(leetcode 59)

举例:

Given n = 3,

You should return the following matrix:
[
[1, 2, 3],

[8, 9, 4],
[7, 6, 5]
]

??? : 时间复杂度O(n^2), 空间复杂度O(1)

```
vector<vector<int> > generateMatrix(int n) {
    vector<vector<int> > matrix(n, vector<int>(n, 0));
    int pos = 1;
    int magrin = 0;
    while(n - 1 - magrin >= magrin && n - 1 - magrin >= magrin){
        for(int j = magrin; j <= n - 1 - magrin; j++)
            matrix[magrin][j] = pos++;
        for(int i = magrin + 1; i < n - 1 - magrin; i++)
            matrix[i][n-1-magrin] = pos++;
        if(n - 1 - magrin != magrin)
            for(int j = n - 1 - magrin; j >= magrin; j-- )
                matrix[n-1-magrin][j] = pos++;
        if(n - 1 - magrin != magrin)
            for(int i = n - 1 - magrin - 1; i > magrin; i--)
                matrix[i][magrin] = pos++;
        magrin++;
    }
    return matrix;
}
```

4.3.3 Set Matrix Zeroes

问题:

Given a m x n matrix, if an element is 0, set its entire row and column to 0. Do it in place. *(leetcode 73)*

Follow Up:

Did you use extra space?
A straight forward solution using O(mn) space is probably a bad idea.
A simple improvement uses O(m + n) space, but still not the best solution.
Could you devise a constant space solution?

??? : 时间复杂度O(n^2), 空间复杂度O(1)

从上到下一行一行的处理，如果上一个存在0,可以先保留上一行的现场，然后根据上一行的原来值更新本行，然后处理上一行．唯一需要注意的就是，如果本行出现新 0 需要更新该 0 所在列的上面所有行.

```
void setZeroes(vector<vector<int> > &matrix) {
    int n = matrix.size();
    if(n == 0)  return;
    int m = matrix[0].size();
    bool lastZero = false;
    for(int i = 0; i < n; i++){
        bool thisZero = false;
        for(int j = 0; j < m; j++){
            if(matrix[i][j] == 0){
                int up = i - 1;
                while(up >= 0){
                    matrix[up][j] = 0;
                    up--;
                }
```

```
                    thisZero = true;
                }
                if(i > 0 && matrix[i-1][j] == 0)
                    matrix[i][j] = 0;
            }
            if(lastZero){
                for(int j = 0; j < m; j++)
                    matrix[i-1][j] = 0;
            }
            lastZero = thisZero;
        }
        if(lastZero){
            for(int j = 0; j < m; j++)
                matrix[n-1][j] = 0;
        }
    }
```

4.3.4 Pascal's Triangle

问题:

Given numRows, generate the first numRows of Pascal's triangle.
(leetcode 118)

举例:

Given numRows = 5,
Return

[
[1],
[1, 1],
[1, 2, 1],
[1, 3, 3, 1],
[1, 4, 6, 4, 1]
]

??? : 时间复杂度O(n^2), 空间复杂度O(1)

```
vector<vector<int> > generate(int numRows) {
    vector<vector<int> > result;
    if(numRows == 0)    return result;
    result.push_back(vector<int>{1});
    for(int i = 1; i < numRows; i++){
        vector<int> cur;
        cur.push_back(1);
        for(int j = 0; j < result[i-1].size() - 1; j++)
            cur.push_back(result[i-1][j] + result[i-1][j+1]);
        cur.push_back(1);
        result.push_back(cur);
    }
    return result;
}
```

4.3.5 Pascal's Triangle II

问题:

Given an index k, return the kth row of the Pascal's triangle.
(leetcode 119)

举例:

Given k = 3,
Return [1, 3, 3, 1].

Note:

Could you optimize your algorithm to use only O(k) extra space?

??? : 时间复杂度O(k^2), 空间复杂度O(k)

这个空间复杂度可以优化，因为每次我们只需要上一行就可以产生本行，所有之前的行可以不存储

```
1       vector<int> getRow(int rowIndex) {
2           rowIndex++;
3           if(rowIndex <= 0)   return vector<int>();
4           vector<int> result{1};
5           for(int i = 1; i < rowIndex; i++){
6               vector<int> cur;
7               cur.push_back(1);
8               for(int j = 0; j < result.size() - 1; j++)
9                   cur.push_back(result[j] + result[j+1]);
10              cur.push_back(1);
11              result.swap(cur);
12          }
13          return result;
14      }
```

第 5 章　栈和队列

5.1　基本概念

栈是一种非常常见的数据结构，因为它具有先进后出的特点，通常被用来模拟递归的实现.

5.2 栈的设计与实现

在STL里面，栈是一个适配器，所谓适配器就是嫁接在其他基本容器之上的容器.像SGI-STL里面栈默认就是内部通过deque来模拟实现，当然我们也可以通过vector, list甚至array来包装成栈.

TODO

5.3 相关问题

5.3.1 Valid Parentheses

问题:

Given a string containing just the characters $'('$,$')'$,$'\{'$,$'\}'$,$'['$,$']'$ and $'and'$, determine if the input string is valid.

The brackets must close in the correct order, "()" and "()[]{}" are all valid but "(}" and "([)]" are not.
(leetcode 20)

栈：时间复杂度O(n), 空间复杂度O(n)

括号匹配的原则是每个右括号都与最左边最近的左括号匹配，所以这和栈的先进后出完全一致，所以使用栈就非常方便.

```
bool isValid(string s) {
    // 0,1,2 represent (, [, {
    stack<int> left;
    for(int i = 0; i < s.size(); i++){
        switch(s[i]){
            case '(': left.push(0); break;
            case '[': left.push(1); break;
            case '{': left.push(2); break;
            case ')': if(left.empty() || left.top() != 0)
                    return false; left.pop(); break;
            case ']': if(left.empty() || left.top() != 1)
                    return false; left.pop(); break;
            case '}': if(left.empty() || left.top() != 2)
                    return false; left.pop(); break;
            default: return false;
        }
    }
    return left.empty();
}
```

5.3.2 Trapping Rain Water

问题:

Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.
(leetcode 42)

举例:

Given [0,1,0,2,1,0,1,3,2,1,2,1], return 6.

栈：时间复杂度O(n), 空间复杂度O(n)

这道题可以这样想，依次遍历$a_1, a_2, ..., a_n$如果$a_i \leq a_{i-1}$那么加入到栈中；反之，进行弹栈处理，因为$a_i > a_{i-1} \leq a_{i-2}$所以可以计算$a_{i-1}$处的积水，依次类推.

```
int trap(vector<int>& height) {
    if(height.size() <= 1)  return 0;
    stack<pair<int,int> > s;
    s.push(make_pair(height[0], 0));
    int sum = 0;
    pair<int,int> cur;
    for(int i = 1; i < height.size(); i++){
        while(height[i] > s.top().first){
            cur = s.top();
            s.pop();
            if(!s.empty())
                sum += (min(height[i], s.top().first) - cur.first)
                        * (i - s.top().second - 1);
            else    break;
        }
        s.push(make_pair(height[i], i));
    }
    return sum;
}
```

DP：时间复杂度O(n), 空间复杂度O(n)

其实栈的方法比较麻烦，这里推荐一种更简单直接的做法，我们来考虑每个元素a_i它能盛放的水量:V_i,我们可以轻易的知道$V_i = min(left[i], right[i])$,其中$left[i], right[i]$分别是$a_i$左右连续最大值.

那么，我们可以通过动态规划计算$left_i right_i$:

$$left[i] = \begin{cases} a_0 & i = 0 \\ max(a_i, left[i-1]) & other \end{cases}$$

$$right[i] = \begin{cases} a_{n-1} & i = n-1 \\ max(a_i, right[i+1]) & other \end{cases}$$

最终$target = \sum_{i=0}^{n-1} V_i$

```
int trap(int A[], int n) {
    vector<int> left(n, 0), right(n, 0);
    left[0] = A[0];
    right[n-1] = A[n-1];
    for(int i = 1; i < n; i++){
        A[i] = A[i] < left[i-1]? left[i-1] : A[i];
    }

    for(int i = n - 2; i >= 0; i--){
        A[i] = A[i] < right[i+1]? right[i+1] : A[i];
    }

    int sum = 0;
    for(int i = 0; i < n; i++){
        min = left[i] < right[i]? left[i] : right[i];
        sum = sum + min - A[i];
    }
    return sum;
}
```

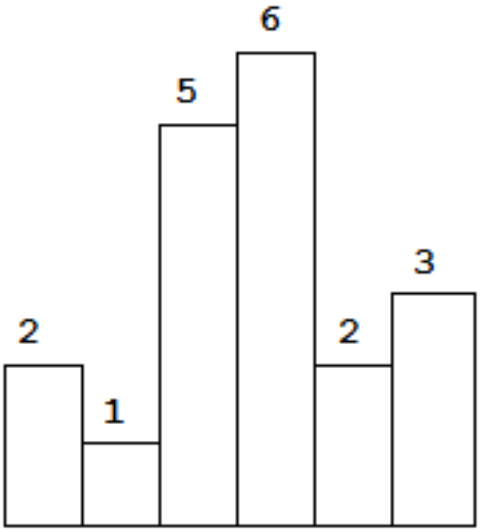
5.3.3 Largest Rectangle in Histogram

问题:

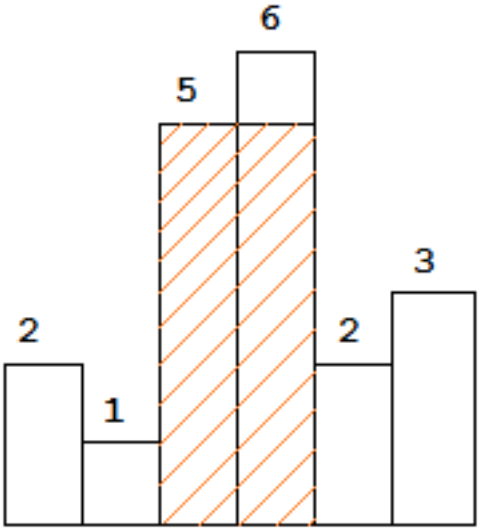
Given n non-negative integers representing the histogram's bar height where the width of each bar is 1, find the area of largest rectangle in the histogram.
(leetcode 84)

举例:

Given height = $[2, 1, 5, 6, 2, 3]$,
Above is a histogram where width of each bar is 1, given height = $[2, 1, 5, 6, 2, 3]$.



The largest rectangle is shown in the shaded area, which has area = 10 unit.



return 10.

栈：时间复杂度O(n), 空间复杂度O(n)

我们可以这样考虑这个问题，我么依次扫描$a_1, a_2, ..., a_n$如果$a_i \geq a_{i-1}$就将a_i加入到栈中；反之，我们就弹栈处理a_{i-1},因为$a_{i-2} < a_{i-1} > a_i$所以我们可以计算以a_{i-1}为基准的histogram大小，依次类推.

```
1  int largestRectangleArea(vector<int> &height) {
2      if(height.size() == 0)  return 0;
3      if(height.size() == 1)  return height[0];
```

```
    stack<pair<int, int> > s;
    s.push(make_pair(height[0], 0));
    int result = INT_MIN;
    for(int i = 1; i < height.size(); i++){
        while(height[i] < s.top().first){
            pair<int, int> cur = s.top();
            s.pop();
            if(!s.empty()){
                if(result < cur.first * (i - s.top().second - 1))
                    result = cur.first * (i - s.top().second - 1);
            }else{
                if(result < cur.first * i)
                    result = cur.first * i;
                break;
            }
        }
        s.push(make_pair(height[i], i));
    }
    int index = s.top().second;
    while(!s.empty()){
        pair<int, int> cur = s.top();
        s.pop();
        if(!s.empty()){
            if(result < cur.first * (index - s.top().second))
                result = cur.first * (index - s.top().second);
        }else{
            if(result < cur.first * (index + 1))
                result = cur.first * (index + 1);
        }
    }
    return result;
}
```

DP : 时间复杂度O(n), 空间复杂度O(n)

同样类似前面的Traip Water问题，分别计算$left[i]right[i]$即可

```
int largestRectangleArea_1st(vector<int> &height) {
    int len = height.size();
    if(len == 0)    return 0;

    vector<int> left(len, 0), right(len, 0);

    for(int i = 1; i < len; i++){
        for(int j = i - 1; j >=0; j--){
            if(height[i] <= height[j]){
                left[i] = left[i] + left[j] + 1;
                j = j - left[j];
            }else{
                break;
            }
        }
    }
    for(int i = len - 2; i >= 0; i--){
        for(int j = i+1; j < len; j++){
            if(height[i] <= height[j]){
                right[i] = right[i] + right[j] + 1;
                j = j + right[j];
            }else{
```

```
                    break;
                }
            }
        }
        int data = 0, tmp;

        for(int i = 0; i < len; i++){
            tmp = height[i] * (left[i] + right[i] + 1);
            if(data < tmp){
                data = tmp;
            }
        }
        return data;
    }
```

5.3.4 Evaluate Reverse Polish Notation

问题:

Evaluate the value of an arithmetic expression in Reverse Polish Notation.

Valid operators are +, -, *, /. Each operand may be an integer or another expression. *(leetcode 150)*

举例:

$["2","1","+","3","*"] \rightarrow ((2+1)*3) -> 9$ $["4","13","5","/","+"] \rightarrow (4+(13/5)) -> 6$

栈：时间复杂度O(n), 空间复杂度O(n)

逆波兰式是典型的栈式问题.

```
bool scanNum(const string &str, int& num){
    if(str.size() == 1 && (str[0] == '+' || str[0] == '-'
        || str[0] == '*' || str[0] == '/') )
        return false;
    bool isMunis = false;
    if(str[0] == '-')   isMunis = true;
    num = 0;
    for(int i = 0; i < str.size(); i++)
        if(str[i] != '+' && str[i] != '-')
            num = num*10 + str[i] - '0';
    num = isMunis? -num : num;
    return true;
}

int evalRPN(vector<string>& tokens) {
    int result = 0, num = 0;
    if(tokens.size() == 0)  return 0;
    scanNum(tokens[0], num);
    if(tokens.size() <= 2)  return num;
    stack<int> s;
    s.push(num);
    scanNum(tokens[1], num);
    s.push(num);
    for(int i = 2; i < tokens.size(); i++){
        if(!scanNum(tokens[i], num)){
            int first = s.top(); s.pop();
            int second = s.top(); s.pop();
            switch(tokens[i][0]){
```

```
                case '+': s.push(second + first); break;
                case '-': s.push(second - first); break;
                case '*': s.push(second * first); break;
                case '/': s.push(second / first); break;
            }
        }else{
            s.push(num);
        }
    }
    result = s.top();
    return result;
}
```

5.3.5 Min Stack

问题:

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

push(x) – Push element x onto stack.
pop() – Removes the element on top of the stack.
top() – Get the top element.
getMin() – Retrieve the minimum element in the stack.
(leetcode 155)

栈：时间复杂度O(n), 空间复杂度O(n)

这里采用经典的双栈法处理，一个栈储存数据，另一个栈储存最小值.

```
class MinStack {

public:

    void push(int x) {
        data.push(x);
        if(track.empty() || x <= track.top())
            track.push(x);
    }

    void pop(){
        if(data.top() == track.top())
            track.pop();
        data.pop();
    }

    int top(){
        return data.top();
    }

    int getMin(){
        return track.top();
    }

private:
    stack<int> data;
    stack<int> track;
};
```

第 6 章　图

6.1　基本概念

图是数据结构中一个非常重要的部分.

6.1.1　图的邻接矩阵表示

TODO

6.1.2　图的邻接表示

TODO

6.2 经典问题

关于图的经典问题我在github上有过总结，请戳这里

6.3 相关问题

6.3.1 Clone Graph

问题:

Clone an undirected graph. Each node in the graph contains a label and a list of its neighbors.

OJ's undirected graph serialization:
Nodes are labeled uniquely.

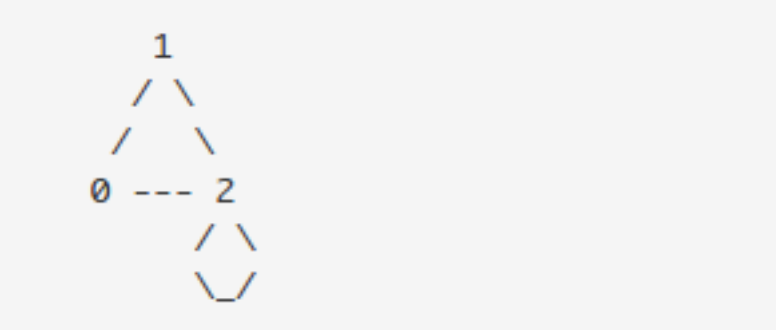
We use ♯ as a separator for each node, and , as a separator for node label and each neighbor of the node.
(leetcode 133)

举例:

As an example, consider the serialized graph $0, 1, 2\sharp 1, 2\sharp 2, 2$.

The graph has a total of three nodes, and therefore contains three parts as separated by ♯.

First node is labeled as 0. Connect node 0 to both nodes 1 and 2.
Second node is labeled as 1. Connect node 1 to node 2.
Third node is labeled as 2. Connect node 2 to node 2 (itself), thus forming a self-cycle.
Visually, the graph looks like the following:



BFS+Hash : 时间复杂度O(E), 空间复杂度O(V)

采用BFS一个个的搜集图的顶点，然后新建顶点和原有顶点通过hash建立一一映射关系方便后来构造neighbors

```
UndirectedGraphNode *cloneGraph(UndirectedGraphNode *node) {
    if(!node)   return NULL;
    unordered_map<UndirectedGraphNode*, UndirectedGraphNode*>   table;
    unordered_set<UndirectedGraphNode*> visited;
    queue<UndirectedGraphNode*> q;
    q.push(node);
    UndirectedGraphNode *new_pos, *pos;
    while(!q.empty()){
        pos = q.front();
        q.pop();
        new_pos = new UndirectedGraphNode(pos->label);
        table[pos] = new_pos;
        for(int i = 0; i < pos->neighbors.size(); i++){
            if(table.count(pos->neighbors[i]) == 0){
                q.push(pos->neighbors[i]);
            }
        }
```

```
18      }
19      q.push(node);
20      visited.insert(node);
21      while(!q.empty()){
22          pos = q.front();
23          q.pop();
24          for(int i = 0; i < pos->neighbors.size(); i++){
25              table[pos]->neighbors.push_back(table[pos->neighbors[i]]);
26              if(visited.count(pos->neighbors[i]) == 0){
27                  q.push(pos->neighbors[i]);
28                  visited.insert(pos->neighbors[i]);
29              }
30          }
31      }
32      return table[node];
33  }
```

6.3.2 Course Schedule

问题:

There are a total of n courses you have to take, labeled from 0 to n - 1.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: $[0, 1]$

Given the total number of courses and a list of prerequisite pairs, is it possible for you to finish all courses?
(leetcode 207)

举例:

2, $[[1, 0]]$
There are a total of 2 courses to take. To take course 1 you should have finished course 0. So it is possible.

2, $[[1, 0], [0, 1]]$
There are a total of 2 courses to take. To take course 1 you should have finished course 0, and to take course 0 you should also have finished course 1. So it is impossible.

BFS : 时间复杂度O(E), 空间复杂度O(V)

这是一题典型的拓扑排序问题，可以使用BFS/DFS进行排序.

```
1   bool canFinish(int numCourses, vector<vector<int>>& prerequisites) {
2       vector<vector<int> > g(numCourses, vector<int>());
3       vector<bool> visited(numCourses, false);
4       vector<int> pre_count(numCourses, 0);
5       for(int i = 0; i < prerequisites.size(); i++){
6           g[prerequisites[i][1]].push_back(prerequisites[i][0]);
7           pre_count[prerequisites[i][0]]++;
8       }
9       queue<int> pre;
10      for(int i = 0; i < numCourses; i++)
11          if(!pre_count[i]){
12              pre.push(i);
13              visited[i] = true;
14      }
15      while(!pre.empty()){
```

```
16          int pos = pre.front();
17          pre.pop();
18          for(int i = 0; i < g[pos].size(); i++){
19              pre_count[g[pos][i]]--;
20              if(pre_count[g[pos][i]] < 0)    return false;
21              if(pre_count[g[pos][i]] == 0){
22                  pre.push(g[pos][i]);
23                  visited[g[pos][i]] = true;
24              }
25          }
26      }
27      for(int i = 0; i < numCourses; i++)
28          if(!visited[i]) return false;
29      return true;
30  }
```

6.3.3 Course Schedule II

问题:

There are a total of n courses you have to take, labeled from 0 to n - 1.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: $[0, 1]$

Given the total number of courses and a list of prerequisite pairs, return the ordering of courses you should take to finish all courses.

There may be multiple correct orders, you just need to return one of them. If it is impossible to finish all courses, return an empty array.
(leetcode 210)

举例:

2, $[[1, 0]]$
There are a total of 2 courses to take. To take course 1 you should have finished course 0. So the correct course order is $[0, 1]$

4, $[[1, 0], [2, 0], [3, 1], [3, 2]]$
There are a total of 4 courses to take. To take course 3 you should have finished both courses 1 and 2. Both courses 1 and 2 should be taken after you finished course 0. So one correct course order is $[0, 1, 2, 3]$. Another correct ordering is$[0, 2, 1, 3]$.

BFS : 时间复杂度O(E), 空间复杂度O(V)
这题和上题一样

```
1   vector<int> findOrder(int numCourses, vector<pair<int, int>>&
        prerequisites) {
2       vector<int> result;
3       vector<int> input(numCourses, 0);
4       vector<vector<int> > graph(numCourses, vector<int>());
5       for(auto ele : prerequisites){
6           input[ele.first]++;
7           graph[ele.second].push_back(ele.first);
8       }
9       queue<int> q;
10      for(int i = 0; i < input.size(); i++)
```

```
11            if(input[i] == 0)    q.push(i);
12        while(!q.empty()){
13            int cur = q.front();
14            q.pop();
15            result.push_back(cur);
16            for(auto ele : graph[cur])
17                if(--input[ele] == 0)    q.push(ele);
18        }
19        return result.size() == numCourses? result : vector<int>();
20    }
```

第 7 章　哈希

7.1　基本概念

TODO

7.2 Hash表的设计与实现

TODO

7.3 相关问题

7.3.1 Substring with Concatenation of All Words

问题:

You are given a string, s, and a list of words, words, that are all of the same length. Find all starting indices of substring(s) in s that is a concatenation of each word in words exactly once and without any intervening characters.
(leetcode 30)

举例:

Given:
s: "barfoothefoobarman"
words: ["*foo*","*bar*"]

You should return the indices: $[0, 9]$.

Hash : 时间复杂度O(n), 空间复杂度O(m)

使用hash表记录已经收集的词及其位置以便及时更新

```
vector<int> findSubstring(string S, vector<string> &L) {
    int n = S.size(), m = L.size();
    vector<int> result;
    if(m == 0 || n == 0 || n < m*L[0].size())   return result;

    unordered_map<string, int> table, find;
    for(int i = 0; i < L.size(); i++){
        if(table.count(L[i])) table[L[i]]++;
        else    table[L[i]] = 1;
    }
    int start = 0, step = L[0].size(), end = 0;

    while(start + m*step <= n){
        string cur = S.substr(end, step);
        if(!table.count(cur) || (find.count(cur) && find[cur] == table[
            cur]) ){
            start++;
            end = start;
            find.clear();
        }else{
            if(!find.count(cur)) find[cur] = 1;
            else find[cur]++;
            end += step;
        }
        if(end - start == m*step)
            result.push_back(start);
    }
    return result;
}
```

7.3.2 Valid Sudoku

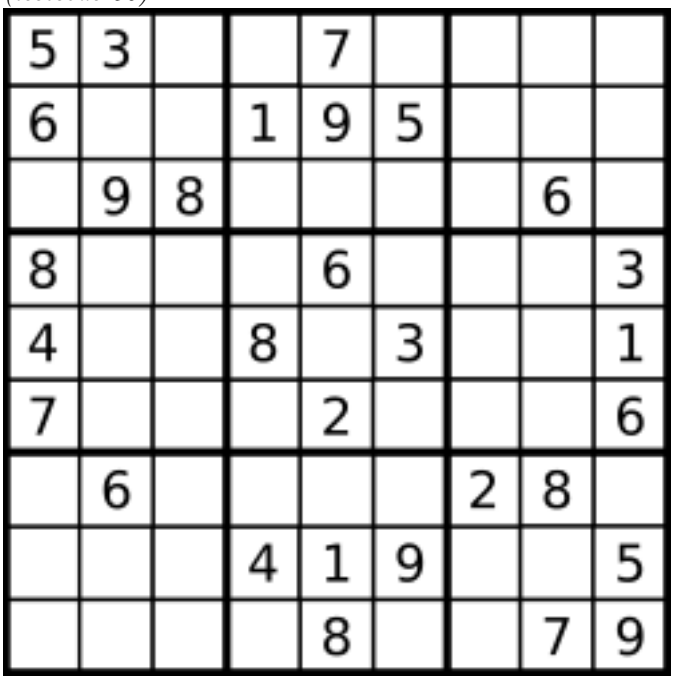
问题:

Determine if a Sudoku is valid, according to: Sudoku Puzzles - The Rules.

The Sudoku board could be partially filled, where empty cells are filled with the character

’.’.

(leetcode 36)



Note:

A valid Sudoku board (partially filled) is not necessarily solvable. Only the filled cells need to be validated.

Hash : 时间复杂度O(???), 空间复杂度O(1)

采用hash来简化行，列和区是否有重复,当然这里采用的hash结构是用vector模拟的

```
bool preprocess(vector<vector<bool> > &row,
                vector<vector<bool> > &col,
                vector<vector<bool> > &area,
                vector<vector<char> > &board){
    for(int i = 0; i < 9; i++){
        for(int j = 0; j < 9; j++){
            if(board[i][j] != '.'){
                if(row[i][board[i][j] - '1'] == false)
                    return false;
                row[i][board[i][j] - '1'] = false;
            }
        }
    }

    for(int j = 0; j < 9; j++){
        for(int i = 0; i < 9; i++){
            if(board[i][j] != '.'){
                if(col[j][board[i][j] - '1'] == false)
                    return false;
                col[j][board[i][j] - '1'] = false;
            }
        }
    }

    for(int i = 0; i < 9; i++){
        for(int j = 0; j < 9; j++){
```

```
27              if(board[i][j] != '.'){
28                  if(area[(i/3)*3 + j/3][board[i][j] - '1'] == false)
29                      return false;
30                  area[(i/3)*3 + j/3][board[i][j] - '1'] = false;
31              }
32          }
33      }
34
35      return true;
36  }
37
38  bool isValidSudoku(vector<vector<char> > &board) {
39      vector<vector<bool> > row(9, vector<bool>(9, true));
40      vector<vector<bool> > col(9, vector<bool>(9, true));
41      vector<vector<bool> > area(9, vector<bool>(9, true));
42
43      return preprocess(row, col, area, board);
44  }
```

7.3.3 Isomorphic Strings

问题:

Given two strings s and t, determine if they are isomorphic.

Two strings are isomorphic if the characters in s can be replaced to get t.

All occurrences of a character must be replaced with another character while preserving the order of characters. No two characters may map to the same character but a character may map to itself.
(leetcode 205)

举例:

Given "egg", "add", return true.

Given "foo", "bar", return false.

Given "paper", "title", return true.

Note:

You may assume both s and t have the same length.

Hash : 时间复杂度O(n), 空间复杂度O(1)

```
1   bool isIsomorphic(string s, string t) {
2       vector<int> table(256, -1), used(256, false);
3       for(int i = 0; i < s.size(); i++){
4           if(table[(unsigned int)s[i]] == -1){
5               if(used[(unsigned int)t[i]])    return false;
6               used[(unsigned int)t[i]] = true;
7               table[(unsigned int)s[i]] = (unsigned int)t[i];
8           }
9           if(table[(unsigned int)s[i]] != (unsigned int)t[i])
10              return false;
11      }
12      return true;
13  }
```

7.3.4 Contains Duplicate

问题:

Given an array of integers, find if the array contains any duplicates. Your function should return true if any value appears at least twice in the array, and it should return false if every element is distinct.
(leetcode 217)

Hash : 时间复杂度O(n), 空间复杂度O(n)

```
bool containsDuplicate(vector<int>& nums) {
    unordered_set<int> table;
    for(int i = 0; i < nums.size(); i++){
        if(table.count(nums[i]))    return true;
        table.insert(nums[i]);
    }
    return false;
}
```

7.3.5 Two Sum

问题:

Given an array of integers, find two numbers such that they add up to a specific target number.

The function twoSum should return indices of the two numbers such that they add up to the target, where index1 must be less than index2. Please note that your returned answers (both index1 and index2) are not zero-based.

You may assume that each input would have exactly one solution.
(leetcode 1)

举例:

Input: numbers=$2, 7, 11, 15$, target=9 Output: index1=1, index2=2

哈希 : 时间复杂度O(n), 空间复杂度O(n)

```
vector<int> twoSum(vector<int> &numbers, int target) {
    unordered_map<int, int> table;
    for(int i = 0; i < numbers.size(); i++)
        table[numbers[i]] = i + 1;
    for(int i = 0; i < numbers.size(); i++){
        if(table.count(target - numbers[i]) && table[target - numbers[i
            ]] != i + 1)
            return vector<int>{i+1, table[target - numbers[i]]};
    }
    return vector<int>{0, 0};
}
```

这题如果是问是否存在，可以采用two pointer夹逼法，这样时间复杂度减少到了O(1)

7.3.6 3Sum

问题:

Given an array S of n integers, are there elements a, b, c in S such that a + b + c = 0? Find all unique triplets in the array which gives the sum of zero.
(leetcode 15)

举例:

Given array S = $-1012-1-4$,

A solution set is:
(-1, 0, 1)
(-1, -1, 2)

Note:

Elements in a triplet (a,b,c) must be in non-descending order. (*ie*, *adbdc*)
The solution set must not contain duplicate triplets.

Hash : 时间复杂度O(n^2), 空间复杂度O(n)

为了避免统计结果出现重复，可以先对原数组进行预处理，排序然后去掉重复元素，再开始利用哈希表排查出3点对

```
1       vector<vector<int> > threeSum(vector<int> &num) {
2           vector<vector<int> > result;
3           if(num.size() < 3)  return result;
4           unordered_map<int, int> table;
5           sort(num.begin(), num.end());
6           vector<pair<int, int> >  shrink;
7           int last = num[0], pos = 0;
8           table[num[0]] = 1;
9           shrink.push_back(make_pair(num[0], 1));
10          for(int i = 1; i < num.size(); i++){
11              if(last == num[i]){
12                  shrink[pos].second++;
13                  table[last]++;
14              }else{
15                  shrink.push_back(make_pair(num[i], 1));
16                  last = num[i];
17                  table[last] = 1;
18                  pos++;
19              }
20          }
21          for(int i = 0; i < shrink.size(); i++)
22              for(int j = i; j < shrink.size(); j++){
23                  if(j == i && shrink[i].second == 1) continue;
24                  int left = 0 - shrink[i].first - shrink[j].first;
25                  if(table.count(left) == 0)  continue;
26                  if(left < shrink[j].first)  break;
27                  if(left == shrink[j].first && j == i
28                          && shrink[j].second == 2)  break;
29                  if(left == shrink[j].first && shrink[j].second == 1)
                        break;
30                  result.push_back(vector<int>{shrink[i].first, shrink[j].
                        first, left});
31              }
32          return result;
33      }
```

7.3.7 4Sum

问题:

Given an array S of n integers, are there elements a, b, c, and d in S such that a + b + c + d = target? Find all unique quadruplets in the array which gives the sum of target.
Elements in a quadruplet (a,b,c,d) must be in non-descending order. (*ie*, *adbdcdd*)

The solution set must not contain duplicate quadruplets.
(leetcode 16)

问题:
Given array S = 10 − 10 − 22, and target = 0.

A solution set is:
(-1, 0, 0, 1)
(-2, -1, 1, 2)
(-2, 0, 0, 2)

Hash : 时间复杂度O(n^2), 空间复杂度O(n^2)
这里同样先排序，然后需要注意重复解

```
vector<vector<int> > fourSum(vector<int> &num, int target) {
    int n = num.size();
    vector<vector<int> > result;
    unordered_map<int,vector<int> > table;
    sort(num.begin(), num.end());
    for(int i = 0; i < n - 1; i++){
        if(num[i] == num[i+1]){
            while(i < n - 1 && num[i] == num[i+1]) i++;
            i--;
            table[num[i] + num[i+1]].push_back(i);
            continue;
        }
        for(int j = i + 1; j < n; j++){
            if(j < n - 1 && num[j] == num[j+1]){
                table[num[i] + num[j]].push_back(i);
                while(j < n - 1 && num[j] == num[j+1]) j++;
                continue;
            }
            table[num[i] + num[j]].push_back(i);
        }
    }
    for(int i = 0; i < n - 3; i++){
        if(num[i] == num[i+1]){
            int left = target - num[i] - num[i+1];
            if(table.count(left)){
                for(int k = 0; k < table[left].size(); k++)
                    if(table[left][k] > i+1){
                        int third = num[table[left][k]];
                        int fourth = left - third;
                        result.push_back(vector<int>{num[i], num[i+1],
                            third, fourth});
                    }
            }
            while(i < n - 1 && num[i] == num[i+1]) i++;
            i--;
            continue;
        }
        for(int j = i + 1; j < n; j++){
            if(j < n - 1 && num[j] == num[j+1]){
                int left = target - num[i] - num[j];
                if(table.count(left)){
                    for(int k = 0; k < table[left].size(); k++)
                        if(table[left][k] > j){
                            int third = num[table[left][k]];
```

```
                                int fourth = left - third;
                                result.push_back(vector<int>{num[i], num[j],
                                    third, fourth});
                            }
                        }
                        while(j < n - 1 && num[j] == num[j+1]) j++;
                        continue;
                    }
                    int left = target - num[i] - num[j];
                    if(table.count(left)){
                        for(int k = 0; k < table[left].size(); k++)
                            if(table[left][k] > j){
                                int third = num[table[left][k]];
                                int fourth = left - third;
                                result.push_back(vector<int>{num[i], num[j],
                                    third, fourth});
                            }
                    }
                }
            }
            return result;
        }
```

7.3.8 Longest Consecutive Sequence

问题:

Given an unsorted array of integers, find the length of the longest consecutive elements sequence.
(leetcode 128)

举例:

Given $[100, 4, 200, 1, 3, 2]$,
The longest consecutive elements sequence is $[1, 2, 3, 4]$. Return its length: 4.

Your algorithm should run in O(n) complexity.

Hash : 时间复杂度O(n), 空间复杂度O(n)

先用hash存好所有数据，然后连续序列就像连通分量一样会被BFS搜出来,只要统计出最大那个连通分量即可

```
int longestConsecutive(vector<int> &num) {
    unordered_set<int> table;
    for(int i = 0; i < num.size(); i++)
        table.insert(num[i]);
    unordered_set<int> used;
    int ret = 0;
    for(int i = 0; i < num.size(); i++){
        if(used.count(num[i]) == 0){
            used.insert(num[i]);
            int con = 1;
            for(int j = 1; ; j++){
                if(table.count(num[i] + j) == 0)    break;
                con++;
                used.insert(num[i] + j);
            }
            for(int j = 1; ; j++){
                if(table.count(num[i] - j) == 0)    break;
```

```
18                      con++;
19                      used.insert(num[i] - j);
20                  }
21                  if(ret < con)   ret = con;
22              }
23          }
24          return ret;
25      }
```

第 8 章　其他数据结构

8.1　堆

堆在STL里面其实是不存在的容器，它是建立在vector基础之上，配合着$make_heap, push_heap pop_heap sort_heap$这四个泛型算法来进行操作的

8.1.1　Implement Heap Algorithm

TODO

8.2 字典树

字典树又称单词查找树，Trie树，是一种树形结构，是一种哈希树的变种
字典树要求根节点不包含字符，除根节点外每一个节点都只包含一个字符； 从根节点到某一节点，路径上经过的字符连接起来，为该节点对应的字符串； 每个节点的所有子节点包含的字符都不相同
字典树的主要用途:
第一：词频统计. 它相对于使用hash记数的方法更加节省空间，因为相同前缀共用.
第二: 前缀匹配. 只要建立好字典树，可以查找出任意公共前缀的字符串，更加高效.

8.2.1 Implement Trie

问题:

Implement a trie with insert, search, and startsWith methods.
(leetcode 208)

Note:

You may assume that all inputs are consist of lowercase letters a-z.

Trie : 时间复杂度O(h), 空间复杂度O(h)

字典树最简单的实现就是看做一个多叉树.

```
1  class TrieNode {
2  public:
3      char val;
4      bool isNode;
5      TrieNode *sons[26];
6  public:
7      // Initialize your data structure here.
8      TrieNode(char val = '-', bool isNode = false)
9      : val(val), isNode(isNode) {
10         for(int i = 0; i < 26; i++)
11             sons[i] = NULL;
12     }
13 };
14
15 class Trie {
16
17 public:
18
19     Trie() {
20         root = new TrieNode();
21     }
22
23     // Inserts a word into the trie.
24     void insert(string s) {
25         TrieNode *pos = root;
26         for(int i = 0; i < s.size(); i++){
27             int index = s[i] - 'a';
28             if(pos->sons[index] == NULL)
29                 pos->sons[index] = new TrieNode(s[i]);
30             pos = pos->sons[index];
31         }
32         pos->isNode = true;
33     }
34
35     // Returns if the word is in the trie.
36     bool search(string key) {
```

```
37          TrieNode *pos = root;
38          for(int i = 0; i < key.size(); i++){
39              int index = key[i] - 'a';
40              if(pos->sons[index] == NULL)
41                  return false;
42              pos = pos->sons[index];
43          }
44          return pos->isNode;
45      }
46
47      // Returns if there is any word in the trie
48      // that starts with the given prefix.
49      bool startsWith(string prefix) {
50          TrieNode *pos = root;
51          for(int i = 0; i < prefix.size(); i++){
52              int index = prefix[i] - 'a';
53              if(pos->sons[index] == NULL)
54                  return false;
55              pos = pos->sons[index];
56          }
57          return true;
58      }
59
60  private:
61      TrieNode* root;
62  };
```

8.3 其他

8.3.1 LRU Cache

问题:

Design and implement a data structure for Least Recently Used (LRU) cache. It should support the following operations: get and set.

get(key) - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.
set(key, value) - Set or insert the value if the key is not already present. When the cache reached its capacity, it should invalidate the least recently used item before inserting a new item.
(leetcode 146)

List+Hash : 时间复杂度O(1), 空间复杂度O(1)

这题是一题非常经典的题目，必须非常熟悉各个数据结构的优劣点才能想到怎么去结合它们设计出O(1)操作的LRU Cache. 我们知道Hash的好处是可以O(1)时间的定位和插入与删除，坏处是数据存储位置难以猜测，彼此独立；数组的优点是O(1)访问，存储连续，但是缺点是无法O(1)的插入和删除;链表和数组相反，插入删除很方便，但是访问就很慢.
回到这个问题，我们想一下首先我们要O(1)时间访问元素即get(key)，那么必须要有哈希结构;然后，我们在insert元素时候，当cache满的时候我们需要花O(1)的时间直接找到哪个是最老的，那我们自然想到了排序，我们可以试着每次插入元素后，都把对应插入时间插入到一个排序的顺序容器中，由于我们还有set操作，所以我们不能使用vector来盛放这个排序序列，因为中间删除代价非常大，只能使用list,当然我们需要在hash表上记录该元素所在list的ListNode的指针方便一次定位.

```
1  class LRUCache {
2  private:
3      struct ListNode{
4          int key;
5          int value;
6          ListNode *pre, *next;
7          ListNode(int key = -1, int value = -1) : key(key), value(value),
               pre(NULL), next(NULL)
8          {}
9      };
10
11     unordered_map<int, ListNode*> table;
12     ListNode seq_head;
13     ListNode seq_end;
14     int capacity;
15     int size;
16
17 private:
18     void seq_push_back(ListNode *pos){
19         seq_end.pre->next = pos;
20         pos->pre = seq_end.pre;
21         seq_end.pre = pos;
22         pos->next = &seq_end;
23     }
24
25     void seq_erase(ListNode *pos){
26         pos->pre->next = pos->next;
27         pos->next->pre = pos->pre;
28     }
29
```

```
30  public:
31
32      LRUCache(int capacity) : capacity(capacity){
33          size = 0;
34          seq_head.next = &seq_end;
35          seq_end.pre = &seq_head;
36      }
37
38      int get(int key){
39          if(table.count(key) == 0)    return -1;
40          seq_erase(table[key]);
41          seq_push_back(table[key]);
42          return table[key]->value;
43      }
44
45
46      void set(int key, int value){
47          if(table.count(key)){
48              table[key]->value = value;
49              seq_erase(table[key]);
50              seq_push_back(table[key]);
51          }else{
52              if(size >= capacity){
53                  ListNode *old = seq_head.next;
54                  table.erase(old->key);
55                  seq_erase(old);
56                  delete old;
57                  size--;
58              }
59              size++;
60              table[key] = new ListNode(key, value);
61              seq_push_back(table[key]);
62          }
63      }
64
65  };
```

第 9 章　排序

9.1　基本概念

排序是最基本的算法,将一个集合里面的元素按照一定顺序排列.排序算法又分为稳定和非稳定的排序算法,不同的排序算法对时间和空间的考虑都不同.

9.2 经典排序算法

经典的排序算法及其各种属性,这里先总结一下:

排序算法	时间复杂度	空间复杂度	稳定性	备注
插入排序	$O(n^2)$	$O(1)$	稳定	
冒泡排序	$O(n^2)$	$O(1)$	稳定	
选择排序	$O(n^2)$	$O(1)$	不稳定	
归并排序	$O(nlgn)$	$O(n)$	稳定	
堆排序	$O(nlgn)$	$O(1)$	不稳定	
快速排序	$O(nlgn)$	$O(1)$	不稳定	最差时间复杂度为$O(n^2)$
计数排序	$O(n)$	$O(m)$		只局限于处于[0,m)的整数排序
桶排序	$O(n)$	$O(n)$		只局限于能通过类似位分割的排序对象

表 9.1: 排序算法比较

9.2.1 Insert Sort

```
1  void insert_sort(int A[], int n){
2      if(n <= 1) return;
3      int cur;
4      for(int i = n - 2; i >= 0; i--){
5          cur = A[i];
6          int j = i + 1;
7          for(; j < n && cur > A[j]; j++)
8              A[j-1] = A[j];
9          A[j-1] = cur;
10     }
11 }
```

9.2.2 Bubble Sort

```
1  void swap(int& a, int& b){
2      //avoid a, b is the same one
3      if(a == b) return;
4      a = a^b;
5      b = a^b;
6      a = a^b;
7  }
8
9  void bubble_sort(int A[], int n){
10     if(n <= 1)  return;
11     for(int i = n - 1; i > 0; i--){
12         for(int j = 0; j < i; j++){
13             if(A[j] > A[j+1])
14                 swap(A[j], A[j+1]);
15         }
16     }
17 }
```

9.2.3 Selection Sort

```
void select_sort(int A[], int n){
    if(n <= 1)  return;
    for(int i = 0; i < n; i++){
        int minPos = i;
        for(int j = i + 1; j < n; j++){
            if(A[minPos] > A[j]){
                minPos = j;
            }
        }
        swap(A[i], A[minPos]);
    }
}
```

9.2.4 Merge Sort

Merge Sort这里分为递归和迭代两种实现,关于迭代的实现可以参考博客归并排序的迭代写法

```
//递归版----------------------merge sort---------
int min(int a, int b){
    return a < b? a : b;
}

void merge_aux(int A[], int begin, int end){
    if(begin >= end)    return;
    int mid = (begin + end)/2;
    merge_aux(A, begin, mid);
    merge_aux(A, mid+1, end);
    int result[end - begin + 1];
    int i = begin, j = mid+1, k = 0;
    while(i <= mid || j <= end){
        int first = i > mid? INT_MAX : A[i];
        int second = j > end? INT_MAX : A[j];
        if(first < second)  i++;
        else    j++;
        result[k++] = min(first, second);
    }
    for(int i = 0; i < k; i++)
        A[begin + i] = result[i];
}

void merge_sort(int A[], int n){
    merge_aux(A, 0, n-1);
}
//-----------------------------------------------
//迭代版----------------------merge sort---------
void merge_iteration(vector<int> &mix, vector<int> &cur){
    vector<int> sum;
    int i = 0, j = 0;
    while(i < mix.size() || j < cur.size()){
        int left = i == mix.size()? INT_MAX : mix[i];
        int right = j == cur.size()? INT_MAX : cur[j];
        if(left < right) i++;
        else    j++;
        sum.push_back(min(left, right));
    }
    cur.clear();
    mix = sum;
```

```
}

void merge_sort(int A[], int n){
    if(n <= 1)  return;
    vector<vector<int> > bucket(64, vector<int>());
    for(int i = 0; i < n; i++){
        int index = 0;
        vector<int> mix(1, A[i]);
        while(!bucket[index].empty()){
            merge_iteration(mix, bucket[index]);
            index++;
        }
        bucket[index] = mix;
    }
    vector<int> mix;
    for(int i = 0; i < 64; i++){
        if(!bucket[i].empty()){
            merge_iteration(mix, bucket[i]);
        }
    }
    for(int i = 0; i < mix.size(); i++)
        A[i] = mix[i];
}
```

9.2.5 Heap Sort

```
void adjust_heap(int A[], int n, int root){
    while(1){
        int left = root*2 + 1 > n - 1? INT_MIN : A[root*2 + 1];
        int right = root*2 + 2 > n - 1? INT_MIN : A[root*2 + 2];
        if(A[root] < max(left, right)){
            if(left < right){
                swap(A[root], A[root*2+2]);
                root = root*2 + 2;
            }else{
                swap(A[root], A[root*2+1]);
                root = root*2 + 1;
            }
        }else
            break;
    }
}

//build max heap
void make_heap(int A[], int n){
    for(int i = n/2 - 1; i >= 0; i--){
        adjust_heap(A, n, i);
    }
}

void pop_heap(int A[], int n){
    if( n <= 1) return;
    swap(A[0], A[n-1]);
    adjust_heap(A, n-1, 0);
}

void heap_sort(int A[], int n){
```

```
    make_heap(A, n);
    for(int i = n; i > 0; i--)
        pop_heap(A, i);
}
```

9.2.6 Quick Sort

```
int partition(int A[], int begin, int end){
    int index = rand()%(end - begin + 1) + begin;
    int last = begin;
    swap(A[index], A[end]);
    for(int i = begin; i < end; i++){
        if(A[i] < A[end]){
            swap(A[last], A[i]);
            last++;
        }
    }
    swap(A[end], A[last]);
    return last;
}

void quick_aux(int A[], int begin, int end){
    if(begin < end){
        int part = partition(A, begin, end);
        quick_aux(A, begin, part - 1);
        quick_aux(A, part + 1, end);
    }
}

void quick_sort(int A[], int n){
    quick_aux(A, 0, n-1);
}
```

9.2.7 Count Sort

```
//assume ele in A is [0, up)
//up is not a very large num
void count_sort(int A[], int n, int up){
    int count[up];
    memset(count, 0, sizeof(count));
    for(int i = 0; i < n; i++)
        count[A[i]]++;
    int pos = 0;
    for(int i = 0; i < up; i++){
        for(int j = 0; j < count[i]; j++){
            A[pos++] = i;
        }
    }
}
```

9.2.8 Bucket Sort

桶排序实现起来比较复杂,这里思路是建立10个头链表和尾链表指针,分别维护0,1,...,9这个10个桶,从低位开始逐渐调整排序,把相应数据放入到相应桶中.

```
struct bucket_node{
    int val;
    bucket_node* next;
};

int march_bucket(int val, int pos){
    return (val / ((int)pow(10,pos))) % 10;
}

void bucket_aux(bucket_node start[], bucket_node end[], int n, int pos){
    bucket_node new_start[10], new_end[10];
    for(int i = 0; i < 10; i++){
        new_start[i].next = 0;
        new_end[i].next = &new_start[i];
    }
    for(int i = 0; i < n; i++){
        bucket_node *head = &start[i], *cur;
        while(head->next){
            cur = head->next;
            head->next = cur->next;
            int id = march_bucket(cur->val, pos);
            (new_end[id].next)->next = cur;
            cur->next = 0;
            new_end[id].next = cur;
        }
    }
    for(int i = 0; i < 10; i++){
        start[i] = new_start[i];
        end[i] = new_end[i];
    }
}

void bucket_pre(int A[], bucket_node node[], int n, int w){
    bucket_node start[10], end[10];
    start[0].next = &node[0];
    end[0].next = &node[n-1];
    for(int i = 1; i < 10; i++){
        start[i].next = 0;
        end[i].next = &start[i];
    }
    for(int i = 0; i < w; i++)
        bucket_aux(start, end, 10, i);
    int pos = 0;
    for(int i = 0; i < 10; i++){
        bucket_node* cur = &start[i];
        while(cur->next){
            cur = cur->next;
            A[pos++] = cur->val;
        }
    }
}

void bucket_sort(int A[], int n, int w){
    if(n <= 1)  return;
    bucket_node node[n];
    bucket_node* pre = &node[0]; ;
    pre->val = A[0];
    for(int i = 1; i < n; i++){
```

```
59          node[i].val = A[i];
60          pre->next = &node[i];
61          pre= pre->next;
62      }
63      pre->next = 0;
64      bucket_pre(A, node, n, w);
65  }
```

9.3 基于排序的问题

9.3.1 Merge Intervals

问题: Given a collection of intervals, merge all overlapping intervals.*(leetcode 56)*

举例:

Given [1,3],[2,6],[8,10],[15,18],
return [1,6],[8,10],[15,18].

排序：时间复杂度O(nlgn), 空间复杂度O(1)

这里先排序,然后逐个的进行合并

```
class mysort{
    public:
        bool operator()(Interval a, Interval b){
            return a.start < b.start;
        }
};

vector<Interval> merge(vector<Interval> &intervals) {
    vector<Interval> result;
    sort(intervals.begin(), intervals.end(), mysort());
    Interval cur;
    int pos = 0;
    while(pos < intervals.size()){
        cur = intervals[pos];
        pos++;
        while(pos < intervals.size() && cur.end >= intervals[pos].start)
            {
            cur.end = max(cur.end, intervals[pos].end);
            pos++;
        }
        result.push_back(cur);
    }
    return result;
}
```

9.3.2 Insert Interval

问题:

Given a set of non-overlapping intervals, insert a new interval into the intervals (merge if necessary).

You may assume that the intervals were initially sorted according to their start times.

(leetcode 57)

举例:

Given intervals [1,3],[6,9], insert and merge [2,5] in as [1,5],[6,9].

Given [1,2],[3,5],[6,7],[8,10],[12,16], insert and merge [4,9] in as [1,2],[3,10],[12,16].

This is because the new interval [4,9] overlaps with [3,5],[6,7],[8,10].

排序：时间复杂度O(nlgn) , 空间复杂度O(1)

先排序,然后找到插入点,插入进去.

```
vector<Interval> insert(vector<Interval> &intervals, Interval
    newInterval) {
```

```
2       vector<Interval> result;
3       Interval cur = newInterval;
4       int pos = 0;
5       bool isMerge = false;
6       while(pos < intervals.size()){
7           if(isMerge || cur.start > intervals[pos].end){
8               result.push_back(intervals[pos]);
9               pos++;
10          }else{
11              if(cur.end >= intervals[pos].start){
12                  cur.start = min(cur.start, intervals[pos].start);
13                  while(pos < intervals.size() && cur.end >= intervals[pos
                        ].start){
14                      cur.end = max(intervals[pos].end, cur.end);
15                      pos++;
16                  }
17              }
18              result.push_back(cur);
19              isMerge = true;
20          }
21      }
22      if(!isMerge)    result.push_back(cur);
23      return result;
24  }
```

9.3.3 Sort Colors

问题:

Given an array with n objects colored red, white or blue, sort them so that objects of the same color are adjacent, with the colors in the order red, white and blue.
Here, we will use the integers 0, 1, and 2 to represent the color red, white, and blue respectively.
(leetcode 75)

Note:

You are not suppose to use the library's sort function for this problem.

Follow up:

A rather straight forward solution is a two-pass algorithm using counting sort.
First, iterate the array counting number of 0's, 1's, and 2's, then overwrite array with total number of 0's, then 1's and followed by 2's.
Could you come up with an one-pass algorithm using only constant space?

排序：时间复杂度O(n)，空间复杂度O(m)

这题第一反应就是使用计数排序,但是Follow up说只能pass一次元素,所以只能采用一点小技巧了:把0一直往前放,把2一直往后放,这样剩余的1就在中间了.

```
1   void sortColors(int A[], int n) {
2       if(n <= 1)  return;
3       int zero_pos = 0, two_pos = n - 1;
4       for(int i = 0; i <= two_pos;){
5           if(A[i] == 0){
6               if(zero_pos != i) swap(A[zero_pos], A[i]);
7               else    i++;
8               zero_pos++;
9           }else if(A[i] == 2){
10              if(two_pos != i) swap(A[two_pos], A[i]);
```

```
            else i++;
            two_pos--;
        }else{
            i++;
        }
    }
}
```

9.3.4 Maximum Gap

问题:

Given an unsorted array, find the maximum difference between the successive elements in its sorted form.
Try to solve it in linear time/space.
Return 0 if the array contains less than 2 elements.
You may assume all elements in the array are non-negative integers and fit in the 32-bit signed integer range.*(leetcode 164)*

排序：时间复杂度O(n) , 空间复杂度O(n)

这题要求O(n)的时间和空间复杂度,而且可以看到元素都是32-bit有符号正数,所以想到使用桶排序.

```
struct bucket_node{
    int val;
    bucket_node* next;
};

int march_bucket(int val, int pos){
    return (val / ((int)pow(10,pos))) % 10;
}

void bucket_aux(bucket_node start[], bucket_node end[], int n, int pos){
    bucket_node new_start[10], new_end[10];
    for(int i = 0; i < 10; i++){
        new_start[i].next = 0;
        new_end[i].next = &new_start[i];
    }
    for(int i = 0; i < n; i++){
        bucket_node *head = &start[i], *cur;
        while(head->next){
            cur = head->next;
            head->next = cur->next;
            int id = march_bucket(cur->val, pos);
            (new_end[id].next)->next = cur;
            cur->next = 0;
            new_end[id].next = cur;
        }
    }
    for(int i = 0; i < 10; i++){
        start[i] = new_start[i];
        end[i] = new_end[i];
    }
}

void bucket_pre(vector<int> &A, bucket_node node[], int n, int w){
    bucket_node start[10], end[10];
    start[0].next = &node[0];
```

```
36      end[0].next = &node[n-1];
37      for(int i = 1; i < 10; i++){
38          start[i].next = 0;
39          end[i].next = &start[i];
40      }
41      for(int i = 0; i < w; i++)
42          bucket_aux(start, end, 10, i);
43      int pos = 0;
44      for(int i = 0; i < 10; i++){
45          bucket_node* cur = &start[i];
46          while(cur->next){
47              cur = cur->next;
48              A[pos++] = cur->val;
49          }
50      }
51  }
52
53  int maximumGap(vector<int> &num) {
54      int n = num.size();
55      if(n <= 1)  return 0;
56      bucket_node node[n];
57      bucket_node* pre = &node[0]; ;
58      pre->val = num[0];
59      for(int i = 1; i < n; i++){
60          node[i].val = num[i];
61          pre->next = &node[i];
62          pre= pre->next;
63      }
64      pre->next = 0;
65      bucket_pre(num, node, n, 10);
66      int gap = INT_MIN;
67      for(int  i = 1; i < n; i++)
68          if(gap < num[i] - num[i-1])
69              gap = num[i] - num[i-1];
70      return gap;
71  }
```

9.3.5 Largest Number

问题:

Given a list of non negative integers, arrange them such that they form the largest number. *(leetcode 179)*

举例: given [3, 30, 34, 5, 9], the largest formed number is 9534330.

Note: The result may be very large, so you need to return a string instead of an integer.

排序：时间复杂度O(nlgn)，空间复杂度O(n*m)

这题最主要是对元素进行排序,排序的关键是如何判断两个元素"谁大谁小",判断的标准是ab与ba大小对比.另外需要注意的是大数据下该用vector/string这样来操作.

```
1   static int march_bucket(int val, int pos){
2       return (val / ((int)pow(10,pos))) % 10;
3   }
4
5   class mysort{
6       public:
7       bool cmp(vector<int> &a, vector<int> &b){
```

```
8           int i = a.size() - 1;
9           for(; i >= 0 && a[i] == b[i]; i--)
10              ;
11          return i < 0? true : a[i] > b[i];
12      }
13
14      // return true if x > y
15      bool operator()(int x, int y){
16          if(x == 0 || y == 0)    return x > y;
17          int i = 9, j = 9;
18          vector<int> a(10,0), b(10,0);
19          while(i >= 0){
20              a[i] = Solution::march_bucket(x, i);
21              b[i] = Solution::march_bucket(y, i);
22              i--;
23          }
24          i = 9;
25          while(i >= 0 && a[i] == 0)
26              i--;
27          while(j >= 0 && b[j] == 0)
28              j--;
29          vector<int> ab, ba;
30          ab.insert(ab.end(), b.begin(), b.begin() + j +1);
31          ab.insert(ab.end(), a.begin(), a.begin() + i + 1);
32          ba.insert(ba.end(), a.begin(), a.begin() + i + 1);
33          ba.insert(ba.end(), b.begin(), b.begin() + j +1);
34          return cmp(ab , ba);
35      }
36  };
37
38  string genString(int ele){
39      string ret;
40      int i = 9;
41      bool start = false;
42      while(i >= 0){
43          int cur = march_bucket(ele, i);
44          if(cur != 0 && !start)  start = true;
45          if(start)   ret.push_back(cur + '0');
46          i--;
47      }
48      if(!start)  return "0";
49      return ret;
50  }
51
52  string largestNumber(vector<int> &num) {
53      sort(num.begin(), num.end(), mysort());
54      string result;
55      for(auto ele : num){
56          if(ele == 0 && result.empty())
57              return "0";
58          else
59              result = result + genString(ele);
60      }
61      return result;
62  }
```

第 10 章　二分查找

10.1　基本概念

二分查找,是一种非常经典的快速查找算法.抽象的来说: 当我们在求满足某个条件C(x)的最小的x时,对于任意满足C(x)的x,如果所有的

$$x_1 \ll x$$

也满足

$$C(x_1)$$

的话就可以使用二分查找来求得最小的x.

10.2 经典问题

二分查找有很多经典实例,比如STL库中常用的lower_bound等等

10.3 相关问题

贪心算法还有很多应用场景，下面介绍几种常见的题目.

10.3.1 Median of Two Sorted Arrays

问题: There are two sorted arrays nums1 and nums2 of size m and n respectively. Find the median of the two sorted arrays. The overall run time complexity should be O(log (m+n)). *(leetcode 4)*

二分查找：时间复杂度O(lgn)，空间复杂度O(1)

这题涉及两个有序数组的查找,主要思想都一样,都是折半查找,去除一部分不合法的候选,然后不断的缩小范围查找.

```
double getMid(int A[], int start, int end){
    if(start > end) return 0;
    int len = end - start + 1;
    return len%2 == 0? (A[start + len/2-1] + A[start + len/2])/2.0 : A[
        start+len/2];
}

double getEle(int A[], int endA, int B[], int endB, int pos){
    int midA = endA / 2;
    int midB = endB / 2;

    if(A[midA] < B[midB]){
        if(midA + midB + 2  > pos + 1){
            if(midB - 1 < 0)    return A[pos];
            return getEle(A, endA, B, midB-1, pos);
        }
        else{
            if(midA + 1 > endA) return B[pos - midA - 1];
            return getEle(&A[midA+1], endA - midA - 1, B, endB, pos -
                midA - 1);
        }
    }else{
        if(midA + midB + 2 > pos + 1){
            if(midA - 1 < 0)    return B[pos];
            return getEle(A, midA-1, B, endB, pos);
        }
    else{
            if(midB+1 > endB)   return A[pos - midB - 1];
            return getEle(A, endA, &B[midB+1], endB - midB - 1, pos -
                midB - 1);
        }
    }
}

double findMedianSortedArrays(int A[], int m, int B[], int n) {
    if(m < 1 && n < 1)  return 0;
    if(m < 1 || n < 1)  return getMid(A, 0, m-1) + getMid(B, 0, n-1);
    if((n+m)%2 == 0){
        int first = getEle(A, m-1, B, n-1, (n+m)/2 - 1);
        int second = getEle(A, m-1, B, n-1, (n+m)/2);
        return (first + second) / 2.0;
    }
    return getEle(A, m-1, B, n-1, (n+m)/2);
```

```
41 }
```

10.3.2 3Sum Closest

问题:

Given an array S of n integers, find three integers in S such that the sum is closest to a given number, target. Return the sum of the three integers. You may assume that each input would have exactly one solution.
(leetcode 16)

举例:

Given array S $=-121-4$, and target $=1$.

The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).

二分查找：时间复杂度O(n^2lgn), 空间复杂度O(1)

这题是找到最近的值，所以hash就力不从心的，我们就可以采用二分搜索的方法

```
1  void binarySearch(vector<int> &num, int begin, int end,
2                    int target, int sum, int &result){
3      if(begin > end) return;
4      int mid = (begin + end) / 2;
5      if(sum + num[mid] == target){
6          result = target;
7          return;
8      }
9      if(result == INT_MAX || abs(target - result) > abs(target - sum -
           num[mid]))
10         result = sum + num[mid];
11     if(sum + num[mid] > target)
12         binarySearch(num, begin, mid-1, target, sum, result);
13     else
14         binarySearch(num, mid+1, end, target, sum, result);
15 }
16
17 int threeSumClosest(vector<int> &num, int target) {
18     sort(num.begin(), num.end());
19     int result = INT_MAX;
20     for(int i = 0; i < num.size() - 2; i++)
21         for(int j = i + 1; j < num.size() - 1; j++){
22             int sum = num[i] + num[j];
23             binarySearch(num, j + 1, num.size() - 1, target, sum, result
                   );
24         }
25     return result;
26 }
```

10.3.3 Search in Rotated Sorted Array

问题:

Suppose a sorted array is rotated at some pivot unknown to you beforehand.*(leetcode 33)*

举例:

(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).

Note:
You are given a target value to search. If found in the array return its index, otherwise return -1.
You may assume no duplicate exists in the array.

二分查找：时间复杂度O(lgn)，空间复杂度O(1)
同样采用二分查找,需要注意的就是如果查找范围已经是排序好的,需要改变查找策略.

```
int binay_search(int A[], int begin, int end, int target){
    if(begin > end) return -1;
    int mid = (begin + end) / 2;
    if(A[mid] == target)    return mid;
    if(A[mid] < target) return binay_search(A, mid+1, end, target);
    else    return binay_search(A, begin, mid-1, target);
}

int search_aux(int A[], int begin, int end, int target){
    if(begin > end) return -1;
    if(begin == end)    return A[begin] == target? begin : -1;
    if(A[end] > A[begin])   return binay_search(A, begin, end, target);
    int mid = (begin + end) / 2;
    if(target == A[mid])    return mid;
    if(A[mid] > A[begin]){
        if(target < A[mid] && target >= A[begin])
            return binay_search(A, begin, mid-1, target);
        return search_aux(A, mid+1, end, target);
    }else if(A[mid] == A[begin]){
        return A[end] == target? end : -1;
    }else{
        if(target > A[mid] && target <= A[end])
            return binay_search(A, mid+1, end, target);
        return search_aux(A, begin, mid-1, target);
    }
}

int search(int A[], int n, int target) {
    return search_aux(A, 0, n-1, target);
}
```

10.3.4 Search in Rotated Sorted Array II

问题:
Follow up for "Search in Rotated Sorted Array":
What if duplicates are allowed?
Would this affect the run-time complexity? How and why?
Write a function to determine if a given target is in the array.
(leetcode 81)

二分查找：时间复杂度O(n)，空间复杂度O(1)
这次旋转数组里面有重复数据,那么切分时候就需要前后判断一下,所以可能在最坏情况下需要O(n)时间复杂度.

```
bool binary_sarch(int A[], int n, int target){
    if(n < 1)   return false;
    int mid = n/2;
    if(A[mid] == target)    return true;
```

```
5       if(A[mid] < target) return binary_sarch(&A[mid+1], n - mid -1,
            target);
6       else    binary_sarch(A, mid, target);
7   }
8
9   bool search(int A[], int n, int target) {
10      if(n < 1)   return false;
11      if(n == 1)  return A[0] == target? true : false;
12      int mid = n/2;
13      if(A[mid] == target)    return true;
14      if(A[mid] > A[0]){
15          if(A[mid] > target && A[0] <= target)
16              return binary_sarch(A, mid, target);
17          return search(&A[mid+1], n - mid - 1, target);
18      }else if(A[mid] < A[0]){
19          if(A[mid] < target && A[n-1] >= target)
20              return binary_sarch(&A[mid+1], n - mid - 1, target);
21          return search(A, mid, target);
22      }else{
23          int pos = 0;
24          while(pos < mid && A[pos] == A[mid])    pos++;
25          if(pos == mid)      return search(&A[mid+1], n - mid - 1, target
                );
26          else    return search(&A[pos], mid - pos, target);
27      }
28  }
```

10.3.5 Search for a Range

问题:Given a sorted array of integers, find the starting and ending position of a given target value.*(leetcode 34)*

举例:

Given [5, 7, 7, 8, 8, 10] and target value 8,
return [3, 4].

Note:

Your algorithm's runtime complexity must be in the order of O(log n).
If the target is not found in the array, return [-1, -1].

二分查找：时间复杂度O(lgn), 空间复杂度O(1)

我这里的方法是先通过二分查找随便找到一个位置,然后再找其上下界.其实可以直接编写类似STL库中的lower_bound和upper_bound就可以了-.-

```
1   int binary_sarch(int A[], int begin, int end, int target){
2       if(begin > end) return -1;
3       int mid = (begin + end) / 2;
4       if(A[mid] == target)    return mid;
5       if(A[mid] > target) return binary_sarch(A, begin, mid-1, target);
6       return binary_sarch(A, mid+1, end, target);
7   }
8
9   int binary_sarchNo(int A[], int begin, int end, int target, bool left){
10      if(begin > end) return left? begin : end;
11      int mid = (begin + end) / 2;
12      int new_begin, new_end;
```

```
13      if(A[mid] == target){
14          new_begin = left? begin : mid + 1;
15          new_end = left? mid - 1 : end;
16      }else if(A[mid] > target){
17          return binary_sarchNo(A, begin, mid - 1, target, left);
18      }else{
19          return binary_sarchNo(A, mid+1, end, target, left);
20      }
21      return binary_sarchNo(A, new_begin, new_end, target, left);
22  }
23
24  vector<int> searchRange(int A[], int n, int target) {
25      vector<int> result{-1, -1};
26      int pos = binary_sarch(A, 0, n-1, target);
27      if(pos != -1){
28          result[0] = pos == 0? 0 : binary_sarchNo(A, 0, pos, target, true
                );
29          result[1] = pos == n-1? n-1 : binary_sarchNo(A, pos, n-1, target
                , false);
30      }
31      return result;
32  }
```

10.3.6 Search Insert Position

问题: Given a sorted array and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.You may assume no duplicates in the array.*(leetcode 35)*

举例:
(1,3,5,6), 5 → 2
(1,3,5,6), 2 → 1
(1,3,5,6), 7 → 4
(1,3,5,6), 0 → 0

二分查找：时间复杂度O(lgn)，空间复杂度O(1)
其实就是lower_bound函数

```
1   int binary_search(int A[], int begin, int end, int target){
2       if(begin > end){
3           return begin;
4       }
5       int mid = (begin + end) / 2;
6       if(A[mid] == target)    return mid;
7       if(A[mid] > target) return binary_search(A, begin, mid-1, target);
8       return binary_search(A, mid+1, end, target);
9   }
10
11  int searchInsert(int A[], int n, int target) {
12      return binary_search(A, 0, n-1, target);
13  }
```

10.3.7 Divide Two Integers

问题: Divide two integers without using multiplication, division and mod operator. If it is overflow, return MAX_INT. *(leetcode 29)*

: 时间复杂度O(lgn) , 空间复杂度O(1)

这题主要思路就是

$$dividend = x * divisor = \sum 2^i divisor$$

,不断的通过二分查找算出一个个

$$2^i$$

,另外需要注意的就是注意溢出问题,这里用的技巧就是全部换算成负数.

```
int div2n(int &dividend, int divisor){
    if(dividend > divisor){
        dividend = 0;
        return 0;
    }
    int i = 1;
    while(divisor > dividend - divisor ){
        divisor +=divisor;
        i = i + i;
    }
    dividend -= divisor;
    return i;
}

int divide(int dividend, int divisor) {
    bool isMinus = (dividend < 0 && divisor > 0) || (dividend > 0 &&
       divisor < 0);
    dividend = dividend < 0? dividend : -dividend;
    divisor = divisor < 0? divisor : -divisor;
    int result = 0;
    while(dividend){
        result -= div2n(dividend, divisor);
    }
    return isMinus? result : (result == INT_MIN? INT_MAX : -result);
}
```

10.3.8 Pow(x, n)

问题: Implement pow(x, n). *(leetcode 50)*

二分查找 : 时间复杂度O(lgn), 空间复杂度O(1)

和除法几乎如出一辙,都是切分成很多个

$$2^i$$

,再用二分法

```
// return x^(count)
double pow2n(double x, int& n){
    int count = 1;
    while(count < n>>1){
        x = x * x;
        count = count<<1;
    }
    n = n - count;
    return x;
}

double pow(double x, int n) {
    if(n == 0)  return 1;
```

```
14      if(n < 0)   return n == INT_MIN? 1.0/(x * pow(x, INT_MAX)) : 1.0/pow
            (x, -n);
15      double result = 1.0;
16      while(n) result = result * pow2n(x, n);
17      return result;
18  }
```

10.3.9 Sqrt(x)

问题: Implement int sqrt(int x). *(leetcode 69)*

二分查找：时间复杂度O(lgn)，空间复杂度O(1)

二分搜索,夹逼方法.这题其实也可以使用牛顿迭代法做.

```
1   int sqrt(int x) {
2       if(x < 2)   return x;
3       int low = 1, high = x/2;
4       while(low <= high){
5           int mid = (low + high) / 2;
6           if(mid == x/mid)    return mid;
7           if(mid > x/mid) high = mid - 1;
8           else    low = mid + 1;
9       }
10      return high;
11  }
```

牛顿迭代法：时间复杂度O(???)，空间复杂度O(1)

```
1   int sqrt(int x) {
2       if(x < 2) return x;
3       int pos = x/2;
4       while(!(pos <= x/pos && (pos+1) >= x/(pos+1))){
5           pos = pos/2 + x/(2*pos);
6       }
7       if((pos+1) == x/(pos+1))    return pos+1;
8       return pos;
9   }
```

10.3.10 Search a 2D Matrix

问题:

Write an efficient algorithm that searches for a value in an m x n matrix. This matrix has the following properties:

Integers in each row are sorted from left to right.
The first integer of each row is greater than the last integer of the previous row.
(leetcode 74)

举例:

Consider the following matrix:
(
(1, 3, 5, 7),
(10, 11, 16, 20),
(23, 30, 34, 50)
(
Given target = 3, return true.

二分查找：时间复杂度O(lgn + lgm)，空间复杂度O(1)

先查找行,再查找列

```
bool searchMatrix(vector<vector<int> > &matrix, int target) {
    int n = matrix.size();
    if(n == 0)  return false;
    int low = 0, high = n - 1;
    int m = matrix[0].size();
    while(low < high){
        int mid = (low + high) / 2;
        if(matrix[mid][m-1] == target)  return true;
        if(matrix[mid][m-1] > target)   high = mid;
        else    low = mid + 1;
    }
    int cur = low;
    low = 0, high = m - 1;
    while(low <= high){
        int mid = (low + high) / 2;
        if(matrix[cur][mid] == target)  return true;
        if(matrix[cur][mid] > target) high = mid - 1;
        else    low = mid + 1;
    }
    return false;
}
```

10.3.11 Find Minimum in Rotated Sorted Array

问题: Suppose a sorted array is rotated at some pivot unknown to you beforehand.
(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).
Find the minimum element.
You may assume no duplicate exists in the array.
(leetcode 153)

二分搜索：时间复杂度O(lgn)，空间复杂度O(1)

这题和search in a rotated array其实解法一样

```
int binary_search(vector<int> &num, int begin, int end){
    if(begin >= end)    return num[begin];
    if(num[begin] < num[end])   return num[begin];
    int mid = (begin + end) / 2;
    if(num[mid] > num[begin]){
        return binary_search(num, mid, end);
    }else if(num[mid] == num[begin]){
        return num[mid+1];
    }else{
        return binary_search(num, begin, mid);
    }
}

int findMin(vector<int> &num) {
    int n = num.size();
    return binary_search(num, 0, n-1);
}
```

10.3.12 Find Minimum in Rotated Sorted Array II

问题:

Follow up for "Find Minimum in Rotated Sorted Array":
What if duplicates are allowed?
Would this affect the run-time complexity? How and why?
(leetcode 154)

二分搜索：时间复杂度O(n)，空间复杂度O(1)

这题和search in a rotated array II其实解法一样

```
int binary_search(vector<int> &num, int begin, int end){
    if(begin >= end)    return num[end];
    if(num[begin] < num[end])   return num[begin];
    int mid = (begin + end) / 2;
    if(num[mid] > num[begin]){
        return binary_search(num, mid+1, end);
    }else if(num[mid] < num[begin]){
        return binary_search(num, begin, mid);
    }else{
        int pos = begin;
        while(pos < mid && num[pos] == num[mid])    pos++;
        if(pos == mid)  return binary_search(num, mid+1, end);
        else    return binary_search(num, begin + pos, mid);
    }
}

int findMin(vector<int> &num) {
    int n = num.size();
    return binary_search(num, 0, n-1);
}
```

10.3.13 Find Peak Element

问题:

A peak element is an element that is greater than its neighbors.
Given an input array where num[i] $\neq$ num[i+1], find a peak element and return its index.
The array may contain multiple peaks, in that case return the index to any one of the peaks is fine.
You may imagine that num[-1] = num[n] = -∞.
(leetcode 162)

举例:

For example, in array [1, 2, 3, 1], 3 is a peak element and your function should return the index number 2.

二分搜索：时间复杂度O(lgn)，空间复杂度O(1)

这题同样采用切分,然后根据丢掉一部分选择范围,我们对于选择范围[s, e],取中间点m = (s + e)/2,判断A[m-1], A[m]和A[m+1]的关系然后选择下一次迭代范围为[s,m-1]还是[m+1,e],然后不断的缩小范围.

```
bool isPeak(const vector<int> &num, int pos){
    return pos == 0?  num[pos] > num[pos+1] : num[pos] > num[pos-1];
}

int binary_sarch(const vector<int> &num, int begin, int end){
    if(begin >= end)    return end;
```

```
7       int  mid = (begin + end) / 2;
8       if(num[mid] > num[mid-1] && num[mid] > num[mid+1])
9           return mid;
10      if(num[mid] >= num[begin] && num[mid] >= num[end]){
11          if(num[mid] < num[mid-1])
12              return binary_sarch(num, begin, mid-1);
13          return binary_sarch(num, mid+1, end);
14      }
15      if(num[mid] < num[begin])
16          return binary_sarch(num, begin, mid-1);
17      else
18          return binary_sarch(num, mid+1, end);
19  }
20
21  int findPeakElement(const vector<int> &num) {
22      int n = num.size();
23      if(n <= 1)  return n-1;
24      if(isPeak(num, 0))  return 0;
25      if(isPeak(num, n-1))    return n - 1;
26      return binary_sarch(num, 1, n-2);
27  }
```

第 11 章　分治

11.1　基本概念

分治算法的基本思想是将一个规模为N的问题分解为K个规模较小的子问题，这些子问题相互独立且与原问题性质相同。求出子问题的解，就可得到原问题的解.
分治法解题的一般步骤:
(1)分解，将要解决的问题划分成若干规模较小的同类问题;
(2)求解，当子问题划分得足够小时，用较简单的方法解决;
(3)合并，按原问题的要求，将子问题的解逐层合并构成原问题的解。

当我们求解某些问题时，由于这些问题要处理的数据相当多，或求解过程相当复杂，使得直接求解法在时间上相当长，或者根本无法直接求出。对于这类问题，我们往往先把它分解成几个子问题，找到求出这几个子问题的解法后，再找到合适的方法，把它们组合成求整个问题的解法。如果这些子问题还较大，难以解决，可以再把它们分成几个更小的子问题，以此类推，直至可以直接求出解为止。这就是分治策略的基本思想。

11.2 经典问题

分治法有很多经典问题，比如归并排序，最近点对问题,最大连续子数组和，网格覆盖问题等

11.3 相关问题

TODO

第 12 章　搜索

12.1　基本概念

搜索,通常是求解问题的一种非常常用的方法,常见的搜索方法有DFS, BFS, A*搜索等等. 有时候大家会混淆搜索和回溯,特别是DFS和回溯的区别,我的理解就是回溯是一种算法,而搜索是一种方法.

12.2 宽度优先搜索

, BFS通常适用于寻找最短路径问题,因为BFS是一步一步的向结果节点靠近的,所以一旦搜到那个此搜索路径必然是最短路径之一.

12.2.1 Word Ladder

问题:

Given two words (beginWord and endWord), and a dictionary, find the length of shortest transformation sequence from beginWord to endWord, such that:

1. Only one letter can be changed at a time
2. Each intermediate word must exist in the dictionary

(leetcode 127)

举例:

Given:
start = "hit"
end = "cog"
dict = ("hot","dot","dog","lot","log")
As one shortest transformation is "hit" - "hot" - "dot" - "dog" - "cog",
return its length 5.

Note:

Return 0 if there is no such transformation sequence.
All words have the same length.
All words contain only lowercase alphabetic characters.

BFS : 时间复杂度O(???), 空间复杂度O(???)

典型的BFS问题,使用队列,一层一层的找到最优解.

```
int ladderLength(string start, string end, unordered_set<string> &dict)
    {
    unordered_set<string> used;
    queue<string> q;
    q.push(start);
    dict.insert(start);
    dict.insert(end);
    used.insert(start);
    int deep = 0, last = 1;
    while(!q.empty()){
        start = q.front();
        q.pop();
        if(start.compare(end) == 0) return deep + 1;
        string tmp = start;
        for(int i = 0; i < start.size(); i++){
            start = tmp;
            for(int j = 'a'; j <= 'z'; j++){
                start[i] = j;
                if(dict.count(start) && !used.count(start)){
                    used.insert(start);
                    q.push(start);
                }
            }
        }
        last--;
```

```
25          if(last == 0){
26              last = q.size();
27              deep++;
28          }
29      }
30      return 0;
31  }
```

12.2.2 Word Ladder II

问题:

Given two words (start and end), and a dictionary, find all shortest transformation sequence(s) from start to end, such that:

Only one letter can be changed at a time
Each intermediate word must exist in the dictionary
(leetcode 126)

举例:

Given:
start = "hit"
end = "cog"
dict = ("hot","dot","dog","lot","log")
Return
(
("hit","hot","dot","dog","cog"),
("hit","hot","lot","log","cog")
)

Note:

All words have the same length.
All words contain only lowercase alphabetic characters.

BFS : 时间复杂度O(???), 空间复杂度O(???)

由于最短路径才可以是最优解,所以不推荐使用DFS,而是使用BFS实现回溯. 这道题需要注意不要访问重复点,还有时间空间要把握好,稍微不慎就会TLE或者MLE

```
1   void genTrack(vector<pair<string, int> > &tree, int pos, vector<string>
        &track){
2       if(pos == 0){
3           track.push_back(tree[0].first);
4           return;
5       }
6       genTrack(tree, tree[pos].second, track);
7       track.push_back(tree[pos].first);
8   }
9
10  /*多一丁点计算都会超时多一丁点空间都会超空间,.....*/
11  vector<vector<string> > findLadders(string start, string end,
12                                      unordered_set<string> &dict){
13      dict.insert(start);
14      vector<vector<string> > result;
15      vector<pair<string, int> > tree;
16      unordered_map<string, int> used;
17      used[start] = 0;
```

```
18      tree.push_back(make_pair(start, -1));
19      int last = 1, pre = -1, pos = 0;
20      bool found = false;
21      string cur, tmp;
22      while(pos < last){
23          cur = tree[pos].first;
24          tmp = cur;
25          for(int i = 0; i < cur.size(); i++){
26              cur = tmp;
27              for(int j = 'a'; j <= 'z'; j++){
28                  cur[i] = j;
29                  if(cur.compare(end) == 0){
30                      found = true;
31                      vector<string> track;
32                      genTrack(tree, pos, track);
33                      track.push_back(end);
34                      result.push_back(track);
35                      continue;
36                  }
37                  if(dict.count(cur) == 1 && (used.count(cur) == 0 || used
                        [cur] >= pre )){
38                      used[cur] = pos;
39                      tree.push_back(make_pair(cur, pos));
40                  }
41              }
42          }
43          pos++;
44          if(!found && pos == last){
45              pre = last;
46              last = tree.size();
47          }
48      }
49      return result;
50  }
```

12.2.3 Number of Islands

问题:

Given a 2d grid map of '1's (land) and '0's (water), count the number of islands. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.
(leetcode 200)

举例:

Example 1:

11110
11010
11000
00000
Answer: 1

Example 2:

11000
11000

00100
00011
Answer: 3

BFS : 时间复杂度O(n*m), 空间复杂度O(n*m)

这题就是通过BFS找到一个个的图连通分量,其实也可以使用DFS进行搜索.另外,需要学习的就是这种对付矩阵四个方向问题,可以在主函数每个方向都调用visit函数,然后在visit函数中判断是否合法,这比直接判断合法要简洁清楚很多.

```
1  void visit(vector<vector<char> > &grid, vector<vector<bool> > &visited,
2             queue<pair<int, int> > &q, int n, int m, int i, int j){
3      if(i < 0 || j < 0 || i > n-1 || j > m-1 ||
4          grid[i][j] == '0' || visited[i][j]) return;
5      visited[i][j] = true;
6      q.push(make_pair(i,j));
7  }
8
9  void bfs(vector<vector<char> > &grid, vector<vector<bool> > &visited,
10          int n, int m, int i, int j){
11     queue<pair<int, int> > q;
12     q.push(make_pair(i, j));
13     pair<int,int> cur;
14     visited[i][j] = true;
15     while(!q.empty()){
16         cur = q.front();
17         q.pop();
18         i = cur.first;
19         j = cur.second;
20         visit(grid, visited, q, n, m, i+1, j);
21         visit(grid, visited, q, n, m, i-1, j);
22         visit(grid, visited, q, n, m, i, j-1);
23         visit(grid, visited, q, n, m, i, j+1);
24     }
25 }
26
27 int numIslands(vector<vector<char> >& grid) {
28     int n = grid.size();
29     if(n == 0)  return 0;
30     int m = grid[0].size();
31     vector<vector<bool> > visited(n, vector<bool>(m, false));
32     int count = 0;
33     for(int i = 0; i < n; i++)
34         for(int j = 0; j < m; j++)
35             if(!visited[i][j] && grid[i][j] == '1'){
36                 count++;
37                 bfs(grid, visited, n, m, i, j);
38             }
39     return count;
40 }
```

12.2.4 Surrounded Regions

问题:

Given a 2D board containing 'X' and 'O', capture all regions surrounded by 'X'.

A region is captured by flipping all 'O's into 'X's in that surrounded region.
(leetcode 130)

举例:

X X X X
X O O X
X X O X
X O X X
After running your function, the board should be:

X X X X
X X X X
X X X X
X O X X

BFS : 时间复杂度O(n*m), 空间复杂度O(1)

这题可以用BFS/DFS进行搜索,但是这里有一个trick: 我们知道O要想存活必须能够扩展到边界,否则就肯定被围,所以我们与其随便找个'O'点开始BFS不如从边界某个'O'开始BFS,因为你随便一点BFS时候在你搜索过程中你还不清楚到底这块会不会被围;但是你从边界的'O'开始BFS那么只要与它连通的肯定是活的,就可以通过这个trcik节省空间开销.

```
1  void visit(vector<vector<char> >& board, queue<pair<int, int> > &search,
2             int n, int m, int i, int j){
3      if(i < 0 || i > n-1 || j < 0 || j > m-1 || board[i][j] != 'O')
           return;
4      search.push(make_pair(i, j));
5      board[i][j] = '-';
6  }
7
8  void bfs(vector<vector<char> > &board, int n, int m, int i, int j){
9      if(board[i][j] != 'O')  return;
10     queue<pair<int,int> > search;
11     pair<int, int> cur;
12     search.push(make_pair(i, j));
13     board[i][j] = '-';
14     while(!search.empty()){
15         cur = search.front();
16         search.pop();
17         i = cur.first;
18         j = cur.second;
19         visit(board, search, n, m, i-1, j);
20         visit(board, search, n, m, i+1, j);
21         visit(board, search, n, m, i, j-1);
22         visit(board, search, n, m, i, j+1);
23 }
24 }
25
26 void solve(vector<vector<char> > &board) {
27     int n = board.size();
28     if(n == 0) return;
29     int m = board[0].size();
30     //从四个边开始搜那么找到的必然是活的,
31     for(int i = 0; i < n; i++){
32         bfs(board, n, m, i, 0);
33         bfs(board, n, m, i, m-1);
34     }
35     for(int j = 0; j < m; j++){
```

```
36          bfs(board, n, m, 0, j);
37          bfs(board, n, m, n-1, j);
38      }
39      for(int i = 0; i < n; i++)
40          for(int j = 0; j < m; j++){
41              if(board[i][j] == 'O') board[i][j] = 'X';
42              if(board[i][j] == '-') board[i][j] = 'O';
43          }
44  }
```

12.2.5 Binary Tree Right Side View

问题:

Given a binary tree, imagine yourself standing on the right side of it, return the values of the nodes you can see ordered from top to bottom.
(leetcode 199)

举例:

Given the following binary tree,
1 ←
/ \
2 3 ←
\ \
5 4 ←
You should return (1, 3, 4).

BFS : 时间复杂度O(n), 空间复杂度O(w)
最普通的宽度优先搜索之队列法.

```
1   vector<int> rightSideView(TreeNode* root) {
2       vector<int> result;
3       if(!root)   return result;
4       queue<TreeNode*> cur, next;
5       cur.push(root);
6       TreeNode* pos;
7       while(!cur.empty()){
8           pos = cur.front();
9           cur.pop();
10          if(pos->left)   next.push(pos->left);
11          if(pos->right)  next.push(pos->right);
12          if(cur.empty()){
13              result.push_back(pos->val);
14              next.swap(cur);
15          }
16      }
17      return result;
18  }
```

12.3 深度优先问题

深度优先搜索在树和回溯那几章已经大量使用,这里就不做专门讨论.

12.4 A*搜索

TODO

第 13 章　回溯

13.1　基本概念

回溯算法实际上一个类似枚举的搜索尝试过程，主要是在搜索尝试过程中寻找问题的解，当发现已不满足求解条件时，就“回溯”返回，尝试别的路径。

回溯法是一种选优搜索法，按选优条件向前搜索，以达到目标。但当探索到某一步时，发现原先选择并不优或达不到目标，就退回一步重新选择，这种走不通就退回再走的技术为回溯法，而满足回溯条件的某个状态的点称为“回溯点”。

在包含问题的所有解的解空间树中，按照深度优先搜索的策略，从根结点出发深度探索解空间树。当探索到某一结点时，要先判断该结点是否包含问题的解，如果包含，就从该结点出发继续探索下去，如果该结点不包含问题的解，则逐层向其祖先结点回溯。（其实回溯法就是对隐式图的深度优先搜索算法）。

有时候需要对搜索空间树进行剪枝,以加快回溯的速度.

13.2 经典问题

回溯法虽然有着"通用解法"的美称,但是一般来说很多问题我们还是优先考虑更优的解法,比如动态规划等方法,但是有些问题就不得不使用回溯法求解了.这里比较著名的就有N皇后问题等

13.2.1 N-Queens

问题:

The n-queens puzzle is the problem of placing n queens on an n×n chessboard such that no two queens attack each other.
Given an integer n, return all distinct solutions to the n-queens puzzle.
Each solution contains a distinct board configuration of the n-queens' placement, where 'Q' and '.' both indicate a queen and an empty space respectively.*(leetcode 51)*

回溯：时间复杂度O(n!), 空间复杂度O(n^2)

其实时间复杂度应该远小于O(n!),这里只是确定一个上界.

```cpp
bool isSafe(vector<pair<int, int> > &queens, int i, int j){
    for(int k = 0; k < queens.size(); k++){
        int x = queens[k].first, y = queens[k].second;
        if(x == i || y == j || abs(i - x) == abs(j - y))
            return false;
    }
    return true;
}

void backtrack(vector<vector<string> > &result, vector<pair<int, int> > &queens,
               vector<string> &track, int row, int row_num){
    if(row == row_num){
        result.push_back(track);
        return;
    }
    string lines(row_num, '.');
    for(int i = 0; i < row_num; i++){
        if(isSafe(queens, row, i)){
            queens.push_back(make_pair(row, i));
            lines[i] = 'Q';
            track.push_back(lines);
            backtrack(result, queens, track, row+1, row_num);
            track.pop_back();
            lines[i] = '.';
            queens.pop_back();
        }
    }
}

vector<vector<string> > solveNQueens(int n) {
    vector<vector<string> > result;
    if(n <= 0)  return result;
    vector<pair<int, int> > queens;
    vector<string> track;
    backtrack(result, queens, track, 0, n);
    return result;
}
```

13.2.2 N-Queens II

问题:

Follow up for N-Queens problem.
Now, instead outputting board configurations, return the total number of distinct solutions.*(leetcode 52)*

回溯：时间复杂度O(n!), 空间复杂度O(n)

同样时间复杂度应该远小于O(n!)

```
bool isSafe(vector<pair<int, int> > &queens, int i, int j){
    for(int k = 0; k < queens.size(); k++){
        int x = queens[k].first, y = queens[k].second;
        if(x == i || y == j || abs(i - x) == abs(j - y))
            return false;
    }
    return true;
}

void backtrack(int &result, vector<pair<int, int> > &queens,
            int row, int row_num){
    if(row == row_num){
        result++;
        return;
    }
    for(int i = 0; i < row_num; i++){
        if(isSafe(queens, row, i)){
            queens.push_back(make_pair(row, i));
            backtrack(result, queens, row+1, row_num);
            queens.pop_back();
        }
    }
}

int totalNQueens(int n) {
    if(n <= 0)  return 0;
    vector<pair<int, int> > queens;
    int result = 0;
    backtrack(result, queens, 0, n);
    return result;
}
```

13.3 相关问题

回溯法可以解决很多需要枚举才能确定解的问题,一般来说会有$a_1, a_2, ..., a_n$的值需要确定,且这些a_i还要满足某种约束. 我们采用的办法是: 从a_1开始分别枚举它的所有可能值,然后采用DFS的做法,依次确定$a_2, ...$,如果确定到a_n,那么这组枚举解是合理解,就放入到搜索结果中,如果枚举到某个a_i发现它已经没有可用解时候,就需要回溯.

另外,我的编码风格是使用result存放最终解,而trace保存一路DFS下来的状态信息,每次进入一个新节点时候trace要压入新节点的状态,每次离开该节点时候就需要pop刚才压入的状态.

13.3.1 Letter Combinations of a Phone Number

问题:

Given a digit string, return all possible letter combinations that the number could represent. A mapping of digit to letters (just like on the telephone buttons) is given below.

(leetcode 17)

举例:

Input:Digit string "23"
Output: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].

Note:

Although the above answer is in lexicographical order, your answer could be in any order you want.

回溯: 时间复杂度O(3^n), 空间复杂度O(n)

```
string table[] = {" ","", "abc", "def", "ghi", "jkl", "mno", "pqrs", "
    tuv", "wxyz"};

void dfs(vector<string> &result, string& digits, string& trace, int pos)
    {
     if(pos == digits.size()){
         result.push_back(trace);
         return;
     }
     string& all = table[digits[pos] - '0'];
     for(int i = 0; i <  all.size(); i++){
         trace.push_back(all[i]);
         dfs(result, digits, trace, pos+1);
         trace.erase(trace.size() - 1);
```

```
    }
}

vector<string> letterCombinations(string digits) {
    vector<string> result;
    int n = digits.size();
    if(n == 0)  return result;
    string trace;
    dfs(result, digits, trace, 0);
    return result;
}
```

13.3.2 Generate Parentheses

问题:

Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.*(leetcode 22)*

举例:

Given n = 3, a solution set is:
"((()))", "(()())", "(())()", "()(())", "()()()"

回溯：时间复杂度O(2^n)，空间复杂度O(n)

这个时间复杂度也是会远低于O(2^n)

```
void dfs(vector<string> &result, int n, string& trace, int left){
    if(left == -1 || (n == 0 && left != 0)) return;
    if(n == 0){
        result.push_back(trace);
        return ;
    }
    trace.push_back('(');
    dfs(result, n-1, trace, left+1);
    trace[trace.size()-1] = ')';
    dfs(result, n-1, trace, left - 1);
    trace.erase(trace.size()-1);
}

vector<string> generateParenthesis(int n) {
    vector<string> result;
    string trace;
    int left = 0;
    dfs(result, 2*n, trace, left);
    return result;
}
```

13.3.3 Sudoku Solver

问题:

Write a program to solve a Sudoku puzzle by filling the empty cells.
Empty cells are indicated by the character '.'.
You may assume that there will be only one unique solution.
(leetcode 37)

举例:

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

A sudoku puzzle...

5	3	4	6	7	8
6	7	2	1	9	5
1	9	8	3	4	2
8	5	9	7	6	1
4	2	6	8	5	3
7	1	3	9	2	4
9	6	1	5	3	7
2	8	7	4	1	9
3	4	5	2	8	6

回溯：时间复杂度O(9!8!7!...1!)，空间复杂度O(1)

时间复杂度也是远远没有O(9!8!7!...1!)这么多

```
unordered_set<char> row[9];
unordered_set<char> col[9];
unordered_set<char> area[9];

bool dfs(vector<vector<char> > &board, int i, int j){
    if(i == 9)  return true;
    if(board[i][j] != '.'){
        i = j == 8? i+1 : i;
        j = j == 8? 0 : j+1;
        return dfs(board, i, j);
    }else{
        for(int k = 1; k <= 9; k++){
            char cur = '0' + k;
            if(!row[i].count(cur) && !col[j].count(cur)
                && !area[(i/3)*3 + j/3].count(cur)){
                row[i].insert(cur);
                col[j].insert(cur);
                area[(i/3)*3 + j/3].insert(cur);
                board[i][j] = cur;
                i = j == 8? i+1 : i;
                j = j == 8? 0 : j+1;
                if(dfs(board, i, j))    return true;
                i = j == 0? i-1 : i;
                j = j == 0? 8 : j-1;
                row[i].erase(cur);
                col[j].erase(cur);
                area[(i/3)*3 + j/3].erase(cur);
            }
        }
        board[i][j] = '.';
        return false;
```

```
32      }
33  }
34
35  void solveSudoku(vector<vector<char> > &board) {
36      for(int i = 0; i < 9; i++){
37          row[i].clear();
38          col[i].clear();
39          area[i].clear();
40      }
41      for(int i = 0; i < 9; i++)
42          for(int j = 0; j < 9; j++){
43              if(board[i][j] != '.'){
44                  row[i].insert(board[i][j]);
45                  col[j].insert(board[i][j]);
46                  area[(i/3)*3+j/3].insert(board[i][j]);
47              }
48          }
49      dfs(board, 0, 0);
50  }
```

13.3.4 Combination Sum

问题:

Given a set of candidate numbers (C) and a target number (T), find all unique combinations in C where the candidate numbers sums to T.
The same repeated number may be chosen from C unlimited number of times.
(leetcode 39)

举例:

Given candidate set 2,3,6,7 and target 7,
A solution set is:
(7)
(2, 2, 3)

Note:

All numbers (including target) will be positive integers.
Elements in a combination (a1, a2, … , ak) must be in non-descending order. (ie, a1 ⩽ a2 ⩽ … ⩽ ak).
The solution set must not contain duplicate combinations.

回溯：时间复杂度O(2^n), 空间复杂度O(n)

这题就是对于每个值包含还是不包含两种选择,然后进行回溯算法.这里需要注意的就是遇到一个值有多个的情况,这里因为每个值可以用无数次,所以就可以直接把原数组数据去重就可以了,具体办法可以每次枚举完a_i后前向枚举下一个第一个不等于a_i的那个.

```
1   void backtrack(vector<int> &num, int target, int pos,
2                  vector<vector<int> > &result, vector<int> &track){
3       if(target == 0){
4           result.push_back(track);
5           return;
6       }
7       if(pos == -1)    return;
8       if(num[pos] <= target){
9           track.push_back(num[pos]);
10          backtrack(num, target - num[pos], pos, result, track);
11          track.pop_back();
12      }
```

```
13      pos--;
14      while(pos >= 0 && num[pos] == num[pos+1]) pos--;
15      if(pos >= 0){
16          backtrack(num, target, pos, result, track);
17      }
18  }
19
20  vector<vector<int> > combinationSum(vector<int> &nums, int target) {
21      sort(nums.begin(), nums.end(), greater<int>());
22      vector<vector<int> > result;
23      vector<int> track;
24      backtrack(nums, target, nums.size()-1, result, track);
25      return result;
26  }
```

13.3.5 Combination Sum II

问题:

Given a collection of candidate numbers (C) and a target number (T), find all unique combinations in C where the candidate numbers sums to T.
Each number in C may only be used once in the combination.
(leetcode 40)

Note:

All numbers (including target) will be positive integers.
Elements in a combination (a1, a2, ... , ak) must be in non-descending order. (ie, a1 $\leqslant$ a2 $\leqslant$ ... $\leqslant$ ak).
The solution set must not contain duplicate combinations.

举例:

For example, given candidate set 10,1,2,7,6,1,5 and target 8,
A solution set is:
(1, 7)
(1, 2, 5)
(2, 6)
(1, 1, 6)

回溯：时间复杂度O(2^n)，空间复杂度O(n)

这题就是对于每个值包含还是不包含两种选择,然后进行回溯算法.这里需要注意的就是遇到一个值有多个的情况,这里通常最简单的处理办法就是进行浓缩,把原数组改成一个个点对,例如(1,1,1,2,3,3)改成((1,3),(2,1),(3,2))这样再每次选择时候消耗一下记数,这样就不会出现重复的结果了.

```
1   void backtrack(vector<int> &key, vector<int> &count, int target,
2               int pos, vector<vector<int> > &result, vector<int> &track){
3       if(target == 0){
4           result.push_back(track);
5           return;
6       }
7       if(pos == -1)    return;
8       if(count[pos] > 0 && key[pos] <= target){
9           track.push_back(key[pos]);
10          count[pos]--;
11          backtrack(key, count, target - key[pos], pos, result, track);
12          count[pos]++;
```

```
13          track.pop_back();
14      }
15      backtrack(key, count, target, pos-1, result, track);
16  }
17
18  vector<vector<int> > combinationSum2(vector<int> &num, int target) {
19      sort(num.begin(), num.end(), greater<int>());
20      vector<int> key, count, track;
21      vector<vector<int> > result;
22      if(num.size() == 0) return result;
23      int last = 0;
24      for(int i = 1; i < num.size(); i++){
25          if(num[i] != num[i-1]){
26              key.push_back(num[last]);
27              count.push_back(i - last);
28              last = i;
29          }
30      }
31      key.push_back(num[last]);
32      count.push_back(num.size() - last);
33      backtrack(key, count, target, key.size() - 1, result, track);
34      return result;
35  }
```

13.3.6 Permutations

问题:Given a collection of numbers, return all possible permutations.*(leetcode 46)*

举例:

(1,2,3) have the following permutations:
(1,2,3), (1,3,2), (2,1,3), (2,3,1), (3,1,2), and (3,2,1).

排序：时间复杂度O(n!), 空间复杂度O(n)

这题本应该使用回溯法求解,特别是对于没有重复元素情况可以非常简单的使用回溯法求解,但是这道题又是经典的组合数学问题: 我们把这些数的组合看成一个数,例如(1,2,3)看成123,(3,2,1)看成321,那么我们可以从最小的数开始逐渐增大这个数直到算到最大的数,那么所有的组合也就求出来了.那么怎么根据现在的数求出下一个更大一点的数的? 假设我们现在的数是$a_1, a_2, ...a_n$,我们找该序列最后一个极大点,设为a_k,我们知道$a_{k-1}<a_k$, $a_k>a_{k+1}>...>a_n$,另外设a_m是$a_k, ..., a_n$中大于a_{k-1}的最小的那个,那么下一个数就是$a_1, a_2, ...a_m, a_k, a_{k+1}, ..., a_{m-1}, a_{k-1}, a_{m+1}, ..., a_n$

```
1   vector<vector<int> > permute(vector<int> &num) {
2       vector<int> data(num);
3       sort(data.begin(), data.end());
4       vector<vector<int> > result;
5       while(1){
6           int peak = data.size() - 1;
7           while(peak > 0 && data[peak] <= data[peak-1])   peak--;
8           result.push_back(data);
9           if(peak == 0)   break;
10          int before_peak = peak - 1;
11          vector<int>::iterator next_peak = lower_bound(data.begin() +
                peak, data.end(),
12                                      data[before_peak], greater<int>());
13          next_peak--;
14          swap(data[before_peak], *next_peak);
15          reverse(data.begin() + peak, data.end());
16      }
```

```
17      return result;
18  }
```

13.3.7 Permutations II

问题:Given a collection of numbers that might contain duplicates, return all possible unique permutations.*(leetcode 47)*

举例:

(1,1,2) have the following unique permutations:
(1,1,2), (1,2,1), and (2,1,1).

排序：时间复杂度O(n!), 空间复杂度O(n)

这题同样还是采用转为大数处理,这里就可以看到,如果采用回溯法,那么还要想办法去掉由于有重复元素带来的问题(压缩原数组变成(元素,记数)对).而这个方法则不需要,是不是很飘逸呢.

```
1   vector<vector<int> > permuteUnique(vector<int>& nums) {
2       vector<int> data(nums);
3       sort(data.begin(), data.end());
4       vector<vector<int> > result;
5       while(1){
6           int peak = data.size() - 1;
7           while(peak > 0 && data[peak] <= data[peak-1])   peak--;
8           result.push_back(data);
9           if(peak == 0)   break;
10          int before_peak = peak - 1;
11          vector<int>::iterator next_peak = lower_bound(data.begin() +
                peak, data.end(),
12                                      data[before_peak], greater<int>());
13          next_peak--;
14          swap(data[before_peak], *next_peak);
15          reverse(data.begin() + peak, data.end());
16      }
17      return result;
18  }
```

13.3.8 Permutation Sequence

问题:

The set [1,2,3,...,n] contains a total of n! unique permutations.

By listing and labeling all of the permutations in order,
We get the following sequence (ie, for n = 3):

"123"
"132"
"213"
"231"
"312"
"321"
Given n and k, return the kth permutation sequence.
(leetcode 60)

Note:

Given n will be between 1 and 9 inclusive.

枚举：时间复杂度O(n), 空间复杂度O(n)

这题其实就是能计算出每种长度的组合有多少个,然后一个个的数就行了.

```
string getPermutation(int n, int k) {
    vector<int> num, count(n+1, 1);
    for(int i = 1; i <= n; i++)
        count[i] = i * count[i-1];
    int pos = n;
    vector<bool> use(n+1, true);
    while(k > 0){
        int i = 0, c = 0;
        for(i = 1; i <= n; i++){
            if(use[i]) c++;
            if( k <= count[pos-1] * c) {
                num.push_back(i);
                use[i] = false;
                k = k - count[pos-1] * (k == count[pos-1] * c? c : c-1);
                break;
            }
        }
        pos--;
    }
    for(int i = n; i >= 1; i--)
        if(use[i]) num.push_back(i);

    string result(n, '0');
    for(int i = 0; i < n; i++)
        result[i] += num[i];
    return result;
}
```

13.3.9 Combinations

问题:

Given two integers n and k, return all possible combinations of k numbers out of 1 ... n. *(leetcode 77)*

举例:

If n = 4 and k = 2, a solution is:

(
(2,4), (3,4), (2,3), (1,2), (1,3), (1,4),)

回溯：时间复杂度O(2^n), 空间复杂度O(n)

从1,2,3,...,n每个枚举有还是没有这两种可能.

```
void backtrack(vector<vector<int> > &result, vector<int> &track,
               int count, int k, int pos, int n){
    if(count == k){
        result.push_back(track);
        return;
    }
    if(pos >= n || k <= 0 || n <= 0)    return;
    track.push_back(pos+1);
    backtrack(result, track, count+1, k, pos+1, n);
    track.pop_back();
    backtrack(result, track, count, k, pos+1, n);
```

```
}

vector<vector<int> > combine(int n, int k) {
    vector<vector<int> > result;
    vector<int> track;
    backtrack(result, track, 0, k, 0, n);
    return result;
}
```

13.3.10 Subsets

问题:

Given a set of distinct integers, nums, return all possible subsets.
(leetcode 78)

举例:

If nums = (1,2,3), a solution is:

(
(3),
(1),
(2),
(1,2,3),
(1,3),
(2,3),
(1,2),
()
)

Note:

Elements in a subset must be in non-descending order.
The solution set must not contain duplicate subsets.

回溯：时间复杂度O(2^n), 空间复杂度O(n)

其实这是一道枚举题目,枚举每个元素包不包含进来两种情况即可.它是没有任何约束条件的回溯,一条道走到黑都是解.

```
void backtrack(vector<vector<int> > &result, vector<int> &track,
               vector<int> &S, int pos, int n){
    if(pos == n){
        result.push_back(track);
        return;
    }
    track.push_back(S[pos]);
    backtrack(result, track, S, pos+1, n);
    track.pop_back();
    backtrack(result, track, S, pos+1, n);
}

vector<vector<int> > subsets(vector<int> &S) {
    sort(S.begin(), S.end());
    int n = S.size();
    vector<vector<int> > result;
    vector<int> track;
    backtrack(result, track, S, 0, n);
    return result;
```

```
20 }
```

13.3.11 Subsets II

问题:

Given a collection of integers that might contain duplicates, nums, return all possible subsets. *(leetcode 90)*

举例:

If nums = (1,2,2), a solution is:

(
(2),
(1),
(1,2,2),
(2,2),
(1,2),
()
)

Note:

Elements in a subset must be in non-descending order.
The solution set must not contain duplicate subsets.

回溯：时间复杂度O(2^n)，空间复杂度O(n)

这题唯一的区别就是含有重复元素,对于这种问题还是那个老办法,先统计重复元素,制作成(元素,记数)对,然后再回溯搜索.

```
1  void backtrack(vector<vector<int> > &result, vector<int> &track, vector<
       int> & data,
2                 vector<int> &count, int pos, int n){
3      if(pos == n){
4          result.push_back(track);
5          return;
6      }
7      for(int i = 0; i < count[pos]; i++){
8          track.push_back(data[pos]);
9          backtrack(result, track, data, count, pos+1, n);
10     }
11     for(int i = 0; i < count[pos]; i++){
12         track.pop_back();
13     }
14     backtrack(result, track, data, count, pos+1, n);
15 }
16
17 vector<vector<int> > subsetsWithDup(vector<int> &S) {
18     sort(S.begin(), S.end());
19     vector<int> data, count;
20     int last = 0;
21     for(int i = 1; i < S.size(); i++){
22         if(S[i] != S[i-1]){
23             data.push_back(S[last]);
24             count.push_back(i - last);
25             last = i;
26         }
27     }
```

```
28      data.push_back(S[last]);
29      count.push_back(S.size() - last);
30      vector<vector<int> > result;
31      vector<int> track;
32      backtrack(result, track, data, count, 0, data.size());
33      return result;
34  }
```

13.3.12 Word Search

问题:

Given a 2D board and a word, find if the word exists in the grid.

The word can be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once.
(leetcode 79)

举例:

Given board =

(
("ABCE"),
("SFCS"),
("ADEE")
)
word = "ABCCED", -¿ returns true,
word = "SEE", -¿ returns true,
word = "ABCB", -¿ returns false.

回溯：时间复杂度($n * m * 4^k$), 空间复杂度O(n*m)

这里从每个点开始出发搜索,不停的向四个方向搜索并回溯.

```
1   bool backtrack(vector<vector<char> > &board, vector<vector<bool> > &used
        , string &word,
2                  int i, int j, int n, int m, int pos, int len){
3       if(pos == len)  return true;
4       if(i < 0 || i >= n || j < 0 || j >= m || used[i][j])    return false
            ;
5       if(board[i][j] == word[pos]){
6           used[i][j] = true;
7           if(backtrack(board, used, word, i+1, j, n, m, pos+1, len))
                return true;
8           if(backtrack(board, used, word, i-1, j, n, m, pos+1, len))
                return true;
9           if(backtrack(board, used, word, i, j+1, n, m, pos+1, len))
                return true;
10          if(backtrack(board, used, word, i, j-1, n, m, pos+1, len))
                return true;
11          used[i][j] = false;
12      }
13      if(pos != 0)    return false;
14      if(backtrack(board, used, word, i, j+1, n, m, pos, len))    return
            true;
15      if(j == m-1 && backtrack(board, used, word, i+1, 0, n, m, pos, len))
                return true;
```

```
16      return false;
17  }
18
19  bool exist(vector<vector<char> > &board, string word) {
20      int n = board.size(), len = word.size();
21      if(n == 0 || len == 0)  return false;
22      int m = board[0].size();
23      vector<vector<bool> > used(n, vector<bool>(m, false));
24      return backtrack(board, used, word, 0, 0, n, m, 0, len);
25  }
```

13.3.13 Gray Code

问题:

The gray code is a binary numeral system where two successive values differ in only one bit. Given a non-negative integer n representing the total number of bits in the code, print the sequence of gray code. A gray code sequence must begin with 0.
(leetcode 89)

举例:

Given n = 2, return [0,1,3,2]. Its gray code sequence is:

00 - 0
01 - 1
11 - 3
10 - 2

Note:

For a given n, a gray code sequence is not uniquely defined.
For example, [0,2,3,1] is also a valid gray code sequence according to the above definition.
For now, the judge is able to judge based on one instance of gray code sequence. Sorry about that.

回溯：时间复杂度O(2^n), 空间复杂度O(2^n)

这道题其实和回溯没多大关系,我的解法就是从第1位开始确定数据,接着确定第二位,...第三位... 第k位的数据就是第k-1位数据集合并上第k-1位数据集合的反序集合.从这个角度上看也算是回溯吧.

```
1   vector<int> grayCode(int n) {
2       vector<int> result;
3       if(n < 0 || n > 31) return result;
4       if(n == 0) return vector<int>(1, 0);
5       vector<int> last{0, 1};
6       result = last;
7       for(int i = 1; i < n; i++){
8           last = result;
9           reverse(last.begin(), last.end());
10          for(int j = 0; j < last.size(); j++){
11              last[j] |= (0x1 << i);
12              result.push_back(last[j]);
13          }
14      }
15      return result;
16  }
```

13.3.14 Restore IP Addresses

问题:

Given a string containing only digits, restore it by returning all possible valid IP address combinations.*(leetcode 93)*

举例:

Given "25525511135",

return ["255.255.11.135", "255.255.111.35"]. (Order does not matter)

回溯：时间复杂度O(C_{11}^3), 空间复杂度O(11)

同样使用回溯,每一步选择IP地址的新的一个字段的值,这个字段可以由1,2或者3个数字组成.

```
int getCon(string &s, int pos, int len){
    if(s[pos] == '0' || pos+1 == len)   return 1;
    if(pos+2 == len)    return 2;
    string substr = s.substr(pos, 3);
    if(substr.compare("256") < 0)   return 3;
    return 2;
}

void backtrack(vector<string> &result, string &track, string &s,
                int pos, int len, int count, int n){
    if(count == n && pos == len){
        result.push_back(track);
        return;
    }
    if(count == n || pos == len) return;
    int con = getCon(s, pos, len);
    string tmp = track;
    for(int i = 1; i <= con; i++){
        track = tmp + (tmp.empty()? "" : ".") + s.substr(pos, i);
        backtrack(result, track, s, pos+i, len, count+1, n);
    }
    track = tmp;
}

vector<string> restoreIpAddresses(string s) {
    int n = s.length();
    vector<string> result;
    string track;
    backtrack(result, track, s, 0, n, 0, 4);
    return result;
}
```

13.3.15 Palindrome Partitioning

问题:

Given a string s, partition s such that every substring of the partition is a palindrome.

Return all possible palindrome partitioning of s.
(leetcode 131)

举例:

Given s = "aab",
Return

(
(”aa”,”b”),
(”a”,”a”,”b”)
)

回溯：时间复杂度 , 空间复杂度O

首先预处理字符串,找到所有回文区间,然后从位置0开始,每个回文区间都尝试一下,采用回溯法搜索结果.

```
void backtrack(vector<vector<string> > &result, vector<string> &track,
                vector<vector<bool> > &pal, string &s, int pos, int n){
    if(pos == n){
        result.push_back(track);
        return;
    }
    for(int j = pos; j < n; j++){
        if(pal[pos][j]){
            track.push_back(s.substr(pos, j - pos + 1));
            backtrack(result, track, pal, s, j+1, n);
            track.pop_back();
        }
    }
}

vector<vector<string> > partition(string s) {
    int n = s.size();
    vector<vector<string> > result;
    vector<string> track;
    vector<vector<bool> > pal(n, vector<bool>(n, false));
    for(int i = 0; i < n; i++){
        for(int j = 0; i - j >= 0 && i + j < n; j++){
            if(s[i-j] != s[i+j]) break;
            pal[i-j][i+j] = true;
        }
    }
    for(int i = 1; i < n; i++){
        for(int j = 1; i - j >= 0 && i + j <= n; j++){
            if(s[i-j] != s[i+j-1]) break;
            pal[i-j][i+j-1] = true;
        }
    }
    backtrack(result, track, pal, s, 0, n);
    return result;
}
```

第 14 章　贪心

14.1　基本概念

贪心法，又称贪心算法，是一种在每一步选择中都采取在当前状态下最好或最优（即最有利）的选择，从而希望导致结果是最好或最优的算法。[1]比如在旅行推销员问题中，如果旅行员每次都选择最近的城市，那这就是一种贪心算法。

贪心算法在有最优子结构的问题中尤为有效。最优子结构的意思是局部最优解能决定全局最优解。简单地说，问题能够分解成子问题来解决，子问题的最优解能递推到最终问题的最优解。

贪心算法与动态规划的不同在于它每对每个子问题的解决方案都做出选择，不能回退。动态规划则会保存以前的运算结果，并根据以前的结果对当前进行选择，有回退功能。

贪心法可以解决一些最优化问题，如：求图中的最小生成树,求哈夫曼编码,Dijkstra算法.....对于其他问题，贪心法一般不能得到我们所要求的答案。一旦一个问题可以通过贪心法来解决，那么贪心法一般是解决这个问题的最好办法。由于贪心法的高效性以及其所求得的答案比较接近最优结果，贪心法也可以用作辅助算法或者直接解决一些要求结果不特别精确的问题。

14.2 经典问题

贪心算法经典问题像Huffman树，最小生成树，Dijkstra算法, 0-1部分背包问题等等.

14.3 相关问题

贪心算法还有很多应用场景，下面介绍几种常见的题目.

14.3.1 Jump Game

问题:

Given an array of non-negative integers, you are initially positioned at the first index of the array.Each element in the array represents your maximum jump length at that position. Determine if you are able to reach the last index.*(leetcode 55)*

举例:

A = [2,3,1,1,4], return true.
A = [3,2,1,0,4], return false.

贪心：时间复杂度O(n), 空间复杂度O(1)

以A = [2,3,1,1,4]举例,从后往前看,其实能到A[4]的有A[1],A[2],A[3],我们知道从A[1]能到A[4]那么必然能从A[1]到A[2]或者A[3]再到A[4],所以我们直接贪心的选取从A[3]到的A[4]. 以这种策略,我们每次只需要选择离目的位置最近的那个跳板就可以了.

```
bool canJump(int A[], int n) {
    int pos = n - 1;
    while(pos > 0){
        int j = pos;
        while(--j >= 0)
            //最近的那个跳板
            if(A[j] + j >= pos) break;
        pos = j;
    }
    return pos == 0;
}
```

14.3.2 Jump Game II

问题:

Given an array of non-negative integers, you are initially positioned at the first index of the array.Each element in the array represents your maximum jump length at that position.Your goal is to reach the last index in the minimum number of jumps.*(leetcode 45)*

举例:

Given array A = [2,3,1,1,4]
The minimum number of jumps to reach the last index is 2. (Jump 1 step from index 0 to 1, then 3 steps to the last index.)

贪心：时间复杂度O(n), 空间复杂度O(1)

我们还是以A=[2,3,1,1,4]为例,我们考虑A[1], A[2], A[3]都能到A[4],如果存在$A[x] \to A[3] \to A[4]$，那么必然存在$A[x] \to A[1] \to A[4]$,因为A[1]在A[3]前面,A[x]必须是先越过A[1]. 所以我们每次选择能到A[4]的所有跳板中最远的那个(A[1])，这样总跳数必然是最少的.

```
int jump(int A[], int n) {
    if(n <= 0)  return -1;
    vector<int> left(n, -1);
    int pos = 0;
    //left[i表示目标位置]的最远跳板位置i
    for(int i = 0; i < n && pos < n - 1; i++){
```

```
7           for(int j = pos + 1; j <= i + A[i] && j < n; j++)
8               left[j] = i;
9           pos = pos > i + A[i]? pos : i + A[i];
10      }
11      int count = 0;
12      pos = n - 1;
13      while(pos > 0){
14          count++;
15          pos = left[pos];
16      }
17      return pos == 0?  count : -1;
18  }
```

这个选取最远跳板还有一个*trick*, 上面代码就是先使用*O(n)*时间产生每个目标位置的最远跳板位置，很具有技巧性

14.3.3 Gas Station

问题:

There are N gas stations along a circular route, where the amount of gas at station i is gas[i]. You have a car with an unlimited gas tank and it costs cost[i] of gas to travel from station i to its next station (i+1). You begin the journey with an empty tank at one of the gas stations. Return the starting gas station's index if you can travel around the circuit once, otherwise return -1. *(leetcode 134)*

Note:

The solution is guaranteed to be unique.

贪心：时间复杂度O(n), 空间复杂度O(1)

首先，我们计算sum(gas) - sum(cost) > 0以确定有解. 我们可以从左到有扫描，初始化sum = 0, 每遇到A[i], 则sum = sum + A[i]，每当sum<0时候就把此时的sum值存起来并令sum=0，那么最后扫描到尾部是sum必然是大于 0 (因为前面的sum都是负数，最后一个再是负数就总和也是负数了)，且那个sum的开始值就是我们要找的起点.
以A = 1, -2, -3, 6, -3, 2为例,一路算出的sum值为-1,-3,5. 那么起点就是最后一个sum的起点即6.

```
1   int canCompleteCircuit(vector<int> &gas, vector<int> &cost) {
2       int size = gas.size();
3       if(size == 0 || gas.size() != cost.size())
4           return -1;
5       if(accumulate(gas.begin(), gas.end(), 0) < accumulate(cost.begin(),
            cost.end(), 0))
6           return -1;
7       int sum = 0;
8       int start = -1;
9       for(int i = 0; i < size; i++){
10          sum += gas[i] - cost[i];
11          if(sum >= 0 && start == -1)
12              start = i;
13          if(sum < 0){
14              sum = 0;
15              start = -1;
16          }
17      }
18      return start;
19  }
```

这道题挺费脑筋的，想到这样还要证明为何这是对的还是挺麻烦的

14.3.4 Candy

问题:

There are N children standing in a line. Each child is assigned a rating value. You are giving candies to these children subjected to the following requirements:
Each child must have at least one candy.
Children with a higher rating get more candies than their neighbors.
What is the minimum candies you must give? *(leetcode 135)*

贪心：时间复杂度O(n), 空间复杂度O(n)

我们对每个位置设两个变量left[i]和right[i]分别表示位于i左边/右边连续的小于A[i]的数目. 然后我们可以使用递推公式left[i] = A[i] > A[i-1]? left[i-1]+1 : 0来求left数组，对应right[i] = A[i] > A[i+1]? right[i+1]+1 : 0来求right数组.

这题老实说感觉应该算动态规划，不过贪心也算是一种特殊的动态规划了，而且left和right数组的求值都是直接通过最优子问题求出，不是多个选出的，从这种角度来说应该算是贪心算法.

```
int candy(vector<int> &ratings) {
    int size = ratings.size();
    vector<int> left(size, 0), right(size, 0);
    for(int i = 1; i < size; i++)
        if(ratings[i] > ratings[i-1]) left[i] = left[i-1] + 1;
        else    left[i] = 0;
    for(int i = size - 2; i >= 0; i--)
        if(ratings[i] > ratings[i+1]) right[i] = right[i+1] + 1;
        else    right[i] = 0;
    int sum = 0;
    for(int i = 0; i < size; i++)
        sum += max(left[i], right[i]) + 1;
    return sum;
}
```

*left[i]*和*right[i]*是一种常用的方法

14.3.5 Best Time to Buy and Sell Stock I

问题: Say you have an array for which the ith element is the price of a given stock on day i. If you were only permitted to complete at most one transaction (ie, buy one and sell one share of the stock), design an algorithm to find the maximum profit. *(leetcode 121)*

贪心：时间复杂度O(n), 空间复杂度O(1)

这道题只需要找到一个最大值和一个最小值，且最小值在最大值左边就可以了.我们可以从右往前遍历并使用rightMax记录当前位置及其右边的最大值，然后那这个最大值减去当前值就算出当前买可以获得的极大值，然后遍历一遍后这些极大值中的最大值就是全局最大值.

```
int maxProfit(vector<int> &prices) {
    if(prices.size() == 0)  return 0;
    int rightMax = prices[prices.size() - 1];
    int get = 0;
    for(int i = prices.size() - 2; i >= 0; i--){
        rightMax = max(prices[i], rightMax);
        if(get < rightMax - prices[i])
            get = rightMax - prices[i];
    }
    return get;
}
```

14.3.6 Best Time to Buy and Sell Stock II

问题: Say you have an array for which the ith element is the price of a given stock on day i. Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times). However, you may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again). *(leetcode 122)*

贪心： 时间复杂度O(n), 空间复杂度O(1)

这道题的选择策略就是一旦开涨就买，一旦下跌就卖，所以只需要记住每次开涨的值，然后一遇到下跌就出手，然后等待下一次开涨.

```
int maxProfit(vector<int> &prices) {
    int last = -1;
    int sum = 0;
    int size = prices.size();
    for(int i = 0; i < size - 1; i++){
        if(prices[i] < prices[i+1] && last == -1)
            last = prices[i];
        if(prices[i] > prices[i+1] && last != -1){
            sum +=  prices[i] - last;
            last = -1;
        }
    }
    if(last != -1){
        sum += prices[size - 1] - last;
    }
    return sum;
}
```

使用*last*值为*-1*标志还没开始入手，还用来过滤一开始就下跌这种情况

14.3.7 Container With Most Water

问题: Given n non-negative integers a1, a2, ..., an, where each represents a point at coordinate (i, ai). n vertical lines are drawn such that the two endpoints of line i is at (i, ai) and (i, 0). Find two lines, which together with x-axis forms a container, such that the container contains the most water.*(leetcode 11)*

Note:You may not slant the container.

贪心： 时间复杂度O(n) , 空间复杂度O(1)

这题采用夹逼法，首先选取左边板为A[0],右边板为A[n-1],这两个边假设可以盛放水量为water，那么我们选取左边板和右边板中的较小的那个，因为这个较少的值决定盛水槽的高度，所以在宽度已经最大的情况下，要想增加盛水量只能试着升高高度，那么只能使得这个两个边板较小的那个向中间移动，以寻找最大盛水量.

假设我们的最优解为中间的某两个边板子，设为A[i]和A[j]，且设A[i] < A[j](反之一样)，我们目前的两个边板设为A[l], A[r],那么在两个边板都没到达最大值的边板位置之前必然有minA[l], A[r] < minA[i], A[j], 接着有一个边到达最大值的位置，假设是A[l] == A[i], 那么必然有A[r] < minA[l], A[j], 所以会一直移动到A[r] == A[j]，即是最大值.

```
int maxArea(vector<int> &height) {
    int water = 0;
    int left = 0, right = height.size() - 1;
    while(left < right){
```

```
5           water = max(water, min(height[left], height[right])*(right -
                left));
6           if(height[left] < height[right])    left++;
7           else    right--;
8       }
9       return water;
10  }
```

这道题虽然叫夹逼法，其实也是贪心的一种体现，因为每次内收较矮的一边.

第 15 章　动态规划

15.1　基本概念

动态规划（英语：Dynamic programming，DP）[1]是一种在数学、计算机科学和经济学中使用的，通过把原问题分解为相对简单的子问题的方式求解复杂问题的方法。 动态规划常常适用于有重叠子问题[2]和最优子结构性质的问题，动态规划方法所耗时间往往远少于朴素解法。

动态规划背后的基本思想非常简单。大致上，若要解一个给定问题，我们需要解其不同部分（即子问题），再合并子问题的解以得出原问题的解。 通常许多子问题非常相似，为此动态规划法试图仅仅解决每个子问题一次，从而减少计算量： 一旦某个给定子问题的解已经算出，则将其记忆化存储，以便下次需要同一个子问题解之时直接查表。 这种做法在重复子问题的数目关于输入的规模呈指数增长时特别有用

动态规划的实现大体分为自顶向下的记忆化递归搜索和自底向上的迭代实现.

15.2 经典问题

动态规划的经典问题像0-1背包问题，完全背包问题，多重背包问题，DAG上的DP问题，最长递增子序列，LCS,矩阵链乘,最大子矩形问题等等等. 可以参考戴方勤的著作...

15.2.1 Maximum Subarray

问题:

Find the contiguous subarray within an array (containing at least one number) which has the largest sum.*(leetcode 53)*

举例:

For example, given the array [2,1,3,4,1,2,1,5,4],
the contiguous subarray [4,1,2,1] has the largest sum = 6.

DP : 时间复杂度O(n) , 空间复杂度O(n)

设动态规划变量为dp[i]表示以A[i]为结尾的最长连续和,那么target=max{dp[i]}

dp的递推关系式:

$$dp[i] = \begin{cases} 0 & i = 0 \\ max\{A[i], A[i] + dp[i-1]\} & other \end{cases}$$

```
int maxSubArray(int A[], int n) {
    if(n <= 0)  return 0;
    int maxCon = A[0];
    vector<int> con(n, A[0]);
    for(int i = 1; i < n; i++){
        con[i] = max(A[i], con[i-1] + A[i]);
        if(maxCon < con[i]) maxCon = con[i];
    }
    return maxCon;
}
```

这题是非常经典的动态规划题目，这类问题通常是以某某为结尾或者某某为开始作为规划目标

15.2.2 Maximal Rectangle

问题:

Given a 2D binary matrix filled with 0's and 1's, find the largest rectangle containing all ones and return its area.*(leetcode 85)*

DP : 时间复杂度O(n*m) , 空间复杂度O(n*m)

这便是经典的最大子矩形问题的一个变种，最大子矩形问题中，允许边界有障碍点，但是本题是不允许边界有障碍点(0),不过本题的解法和最大子矩形问题的解法一样，都是对矩形中每点求其悬线，悬线左边距离和右边距离，然后在每个悬线左右扫描形成的极大子矩形中选取一个为最大子矩形.

我们首先来声明几个定义:

- 悬线height[i][j]: 矩阵中点A[i][j]的悬线指从该点出发向上走，走到第一个障碍点或者边界的距离(含本点).

- 左边距离left[i][j]: 矩阵中点A[i][j]的左边距离指的是该点的悬线向左移动在不会碰到障碍点或者边界的情况下能移动的最大距离(含本点).
- 右边距离right[i][j]: 矩阵中点A[i][j]的右边距离指的是该点的悬线向右移动在不会碰到障碍点或者边界的情况下能移动的最大距离(含本点).

我们以矩阵A为例:

$$A = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

那么点A[2][1]位置的悬线高度为2,左边距离为1,右边距离为2. 所以点A[2][1]悬线左右扫面形成的矩形面积为2*(1+2-1) = 4

于是我们target = max{height[i][j]*(left[i][j] + right[i][j] - 1)}

另外对于height,left,right的求值都有递推公式可以算出:

$$height[i][j] = \begin{cases} 0 & A[i][j] = 0 \\ 1 & A[i][j]! = 0 \ \ \& \ \ i = 0 \\ 1 + height[i-1][j] & A[i][j]! = 0 \ \ \& \ \ i! = 0 \end{cases}$$

假设left_zero为A[i][j]同行中左边最近的那个0的位置

$$left[i][j] = \begin{cases} 0 & A[i][j] = 0 \\ i - left_zero & A[i][j]! = 0 \ \ \& \ \ i = 0 \\ min(i - left_zero, left[i-1][j]) & A[i][j]! = 0 \ \ \& \ \ i! = 0 \end{cases}$$

假设right_zero为A[i][j]同行中右边最近的那个0的位置

$$right[i][j] = \begin{cases} 0 & A[i][j] = 0 \\ right_zero - i & A[i][j]! = 0 \ \ \& \ \ i = 0 \\ min(right_zero - i, right[i-1][j]) & A[i][j]! = 0 \ \ \& \ \ i! = 0 \end{cases}$$

```
int maximalRectangle(vector<vector<char> > &matrix){
    int n = matrix.size();
    if(n == 0)  return 0;
    int m = matrix[0].size();
    vector<vector<int> > height(n, vector<int>(m, 0));
    //悬线包含本元素,
    vector<vector<int> > left(n, vector<int>(m, 0));
    //左边包含本元素,
    vector<vector<int> > right(n, vector<int>(m, 0));
    //右边包含本元素,
    //计算悬线
    for(int i = 0; i < n; i++)
        for(int j = 0; j < m; j++){
            if(matrix[i][j] != '0'){
                height[i][j] = i == 0? 1 : height[i-1][j] + 1;
            }
        }
    //计算左边界
    for(int i = 0; i < n; i++){
        int left_zero = -1;
        for(int j = 0; j < m; j++){
            if(matrix[i][j] != '0'){
```

```
23                  if(j == 0) left[i][j] = 1;
24                  else{
25                      if(height[i][j] == 1){
26                          left[i][j] = j - left_zero;
27                      }else{
28                          left[i][j] = min(j - left_zero, left[i-1][j]);
29                      }
30                  }
31              }else{
32                  left_zero = j;
33              }
34          }
35      }
36      //计算右边界
37      for(int i = 0; i < n; i++){
38          int right_zero = m;
39          for(int j = m-1; j >= 0; j--){
40              if(matrix[i][j] != '0'){
41                  if(j == m-1)    right[i][j] = 1;
42                  else{
43                      if(height[i][j] == 1){
44                          right[i][j] = right_zero - j;
45                      }else{
46                          right[i][j] = min(right_zero - j, right[i-1][j])
                                ;
47                      }
48                  }
49              }else{
50                  right_zero = j;
51              }
52          }
53      }
54      //计算每个悬线左右扫描确定的极大子矩阵
55      int maxRect = 0;
56      for(int i = 0; i < n; i++){
57          for(int j = 0; j < m; j++){
58              if(matrix[i][j] != '0'){
59                  int cur = height[i][j]*(left[i][j] + right[i][j] - 1);
60                  maxRect = max(maxRect, cur);
61              }
62          }
63      }
64      return maxRect;
65  }
```

这题是非常经典的动态规划题目，这类问题的更详细解释可以参考王知昆的论文

15.2.3 Triangle

问题:

Given a triangle, find the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below.*(leetcode 120)*

举例:

given the following triangle

2,
3,4,

6,5,7,
4,1,8,3

The minimum path sum from top to bottom is 11 (i.e., 2 + 3 + 5 + 1 = 11).

Note:Bonus point if you are able to do this using only O(n) extra space, where n is the total number of rows in the triangle.

DP : 时间复杂度O(n^2) , 空间复杂度O(n)
设动态规划变量为dp[i][j]表示处于triangle[i][j]到顶点的最短距离,那么target=min{dp[n-1][j]}
dp的递推关系式:

$$dp[i][j] = \begin{cases} triangle[i][j] & i = 0 \\ dp[i-1][j] & j = 0 \\ dp[i-1][j-1] & j = triangle[i].size() - 1 \\ min(dp[i-1][j-1], dp[i-1][j]) + triangle[i][j] & other \end{cases}$$

```
1  int minimumTotal(vector<vector<int> > &triangle) {
2      int n = triangle.size();
3      if( n == 0) return 0;
4      vector<int> odp(triangle[0]);
5      vector<int> ndp(triangle[0]);
6      for(int i = 1; i < n; i++){
7          odp = ndp;
8          ndp.clear();
9          ndp.resize(i+1);
10         for(int j = 0; j < triangle[i].size(); j++){
11             ndp[j] = triangle[i][j] + min((j == 0? INT_MAX : odp[j-1]),
12                                 (j == triangle[i].size()-1? INT_MAX : odp[j
                                       ]));
13         }
14     }
15     auto iter = min_element(ndp.begin(), ndp.end());
16     return *iter;
17 }
```

我们可以看到*dp[i]*行变量只依赖*dp[i-1]*变量，所以可以使用滚动数组将*O(n^2)*空间复杂度降为*O(n)*

15.2.4 House Robber

问题:
You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security system connected and it will automatically contact the police if two adjacent houses were broken into on the same night.

Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight without alerting the police.
(leetcode 198)

DP : 时间复杂度O(n), 空间复杂度O(1)
这题就是经典的最大无连续子序列问题，我们假设dp[i]表示nums[0..i]可以rob的最大价值，target

= dp[n-1]
所以我们有递推公式:

$$dp[i] = \begin{cases} nums[0] & i = 0 \\ max(nums[0], nums[1]) & i = 1 \\ max(nums[i] + dp[i-2], dp[i-1]) & other \end{cases}$$

我们可以在编写代码时候对dp进行状态压缩以减少空间复杂度

```
int rob(vector<int>& nums) {
    int n = nums.size();
    if(n == 0)  return 0;
    if(n == 1)  return nums[0];
    int dp1 = nums[0], dp2 = max(nums[0], nums[1]);
    for(int i = 2; i < n; i++){
        int cur = max(nums[i] + dp1, dp2);
        dp1 = dp2;
        dp2 = cur;
    }
    return dp2;
}
```

15.3 相关问题

动态规划算法还有很多应用场景，下面介绍几种常见的题目.

15.3.1 Longest Valid Parentheses

问题:

Given a string containing just the characters '(' and ')', find the length of the longest valid (well-formed) parentheses substring.
For "(()", the longest valid parentheses substring is "()", which has length = 2.
Another example is ")()())", where the longest valid parentheses substring is "()()", which has length = 4. *(leetcode 32)*

DP : 时间复杂度O(n) , 空间复杂度O(n)

设动态规划变量为dp[i]表示以s[i]为开始的最长匹配串长度.以")()())"为例,dp[i]分别为0,4,0,2,0,0.
所以target = max{dp[i]}即为4.

dp的递推关系式:

$$dp[i] = \begin{cases} 0 & s[i] = ')' \,||\, i = n-1 \\ 2 + dp[i+2] & s[i, i+1] = "()" \\ 0 & s[i, i+1] = "((" \quad \& \quad i + dp[i+1] = n-1 \\ 0 & s[i, i+1] = "((" \quad \& \quad s[i + dp[i+1]] = '(' \\ 2 + dp[i+1] + dp[i + dp[i+1] + 2] & s[i, i+1] = "((" \quad \& \quad s[i + dp[i+1]] = ')' \end{cases}$$

```
int longestValidParentheses(string s) {
    int size = s.size();
    vector<int> dp(size + 1, 0);
    int max = 0;
    for(int i = size - 2; i >= 0; i--){
        if(s[i] == '('){
            if(s[i+1] == ')'){
                dp[i] = 2 + dp[i+2];
            }else{
                if(i + dp[i+1] < size - 1){
                    if(s[i + dp[i+1] + 1] == ')')
                        dp[i] = 2 + dp[i+1] + dp[i + dp[i+1] + 2];
                }
            }
        }
        if(max < dp[i])
            max = dp[i];
    }
    return max;
}
```

15.3.2 Unique Paths

问题:

A robot is located at the top-left corner of a m x n grid (marked 'Start' in the diagram below).
The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).
How many possible unique paths are there? *(leetcode 62)*

DP : 时间复杂度O(n*m) , 空间复杂度O(n*m)
设动态规划变量为dp[i][j]表示从起点到i,j位置的路径数目，那么target = dp[n-1][m-1]

dp的递推关系式:

$$dp[i][j] = \begin{cases} 1 & i = 0 \,||\, j = 0 \\ dp[i-1][j] + dp[i][j-1] & other \end{cases}$$

```
int uniquePaths(int m, int n) {
    if(m < 1 || n < 1)  return 0;
    vector<vector<int> > dp(m, vector<int>(n, 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];
}
```

15.3.3 Unique Paths II

问题:
Follow up for "Unique Paths":
Now consider if some obstacles are added to the grids. How many unique paths would there be?
An obstacle and empty space is marked as 1 and 0 respectively in the grid.*(leetcode 63)*

DP : 时间复杂度O(n*m) , 空间复杂度O(n*m)
设动态规划变量为dp[i][j]表示从起点到i,j位置的路径数目，那么target = dp[n-1][m-1]

dp的递推关系式:

$$dp[i][j] = \begin{cases} 0 & A[i][j] = 1 \\ 1 & A[0][0] = 0 \ \& \ i = 0 \ \& \ j = 0 \\ dp[i][j-1] & A[i][j] = 0 \ \& \ i = 0 \ \& \ j! = 0 \\ dp[i-1][j] & A[i][j] = 0 \ \& \ i! = 0 \ \& \ j = 0 \\ dp[i-1][j] + dp[i][j-1] & A[i][j] = 0 \ \& \ i! = 0 \ \& \ j! = 0 \end{cases}$$

```
int uniquePathsWithObstacles(vector<vector<int> > &obstacleGrid) {
    int m = obstacleGrid.size();
    int n = obstacleGrid[0].size();
    vector<vector<int> > dp(m, vector<int>(n, 1));
    for(int i = 0; i < m; i++)
        for(int j = 0; j < n; j++){
            if(obstacleGrid[i][j])  dp[i][j] = 0;
            else{
                if(i + j != 0)
                    dp[i][j] = (i > 0? dp[i-1][j] : 0) + (j > 0? dp[i][j
                        -1] : 0);
            }
        }
    return dp[m-1][n-1];
}
```

这题和*Unique Paths*唯一的区别就是注意路障不能选

15.3.4 Minimum Path Sum

问题:

Given a m x n grid filled with non-negative numbers, find a path from top left to bottom right which minimizes the sum of all numbers along its path.*(leetcode 64)*

Note:

You can only move either down or right at any point in time.

DP : 时间复杂度O(n*m) , 空间复杂度O(n*m)

设动态规划变量为dp[i][j]表示从起点到i,j位置的最短路径长度，那么target = dp[n-1][m-1]

dp的递推关系式:

$$dp[i][j] = \begin{cases} A[0][0] & i = 0 \ \ \& \ \ j = 0 \\ A[i][j] + dp[i][j-1] & i = 0 \ \ \& \ \ j! = 0 \\ A[i][j] + dp[i-1][j] & i! = 0 \ \ \& \ \ j = 0 \\ min(dp[i-1][j], dp[i][j-1]) + A[i][j] & other \end{cases}$$

```
int minPathSum(vector<vector<int> > &grid) {
    int n = grid.size();
    int m = grid[0].size();
    vector<vector<int> > dp(grid);
    for(int i = 0; i < n; i++)
        for(int j = 0; j < m; j++){
            if(i + j != 0){
                dp[i][j] = grid[i][j] +
                        min((i > 0? dp[i-1][j] : INT_MAX), (j > 0? dp[i
                            ][j-1] : INT_MAX));
            }
        }
    return dp[n-1][m-1];
}
```

15.3.5 Climbing Stairs

问题:

You are climbing a stair case. It takes n steps to reach to the top.
Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?*(leetcode 70)*

DP : 时间复杂度O(n) , 空间复杂度O(1)

其实本题就是著名的斐波那契数列问题，设动态规划变量为dp[i]为到阶梯i的所有可能的路数，那么target=dp[n-1].
dp的递推关系式:

$$dp[i] = \begin{cases} 1 & i = 0 \\ 2 & i = 1 \\ dp[i-2] + dp[i-1] & other \end{cases}$$

```
int climbStairs(int n) {
    if(n <= 1)  return n;
    int fn1 = 2, fn2 = 1, fn = 2;
    for(int i = 3; i <= n; i++){
        fn = fn1 + fn2;
```

```
6           fn2 = fn1;
7           fn1 = fn;
8       }
9       return fn;
10  }
```

虽然我们递推公式中是使用*dp[i]*一个数组来解释，但是因为*dp[i]*只依赖*dp[i-1]*和*dp[i-2]*,那么可以压缩状态存储空间到*O(1)*,这是节约空间的一种常用手段，像轮转数组等等

15.3.6 Scramble String

问题:

Given a string s1, we may represent it as a binary tree by partitioning it to two non-empty substrings recursively.
Below is one possible representation of s1 = "great":

great
/ \
gr eat
/ \ / \
g r e at
/ \
a t
To scramble the string, we may choose any non-leaf node and swap its two children.

For example, if we choose the node "gr" and swap its two children, it produces a scrambled string "rgeat".

rgeat
/ \
rg eat
/ \ / \
r g e at
/ \
a t
We say that "rgeat" is a scrambled string of "great".

Similarly, if we continue to swap the children of nodes "eat" and "at", it produces a scrambled string "rgtae".

rgtae
/ \
rg tae
/ \ / \
r g ta e
/ \
t a
We say that "rgtae" is a scrambled string of "great".

Given two strings s1 and s2 of the same length, determine if s2 is a scrambled string of s1.*(leetcode 87)*

DP : 时间复杂度O(n^4) , 空间复杂度O(n^3)
我们假设dp[i][j][l]表示$s1_i, ...s1_{i+l-1}$和$s2_j, ...s2_{j+l-1}$是否互为可转置的.

dp的递推关系式:

$$dp[i][j][l] = \begin{cases} s1[i] == s2[j] & l = 1 \\ \prod_{k=1}^{l-1}(dp[i][j][k] \ \& \ dp[i+k][j+k][l-k]) || (dp[i][j+l-k][k] \ \& \ dp[i+k][j][l-k]) & other \end{cases}$$

其中上面的求积符号代表或运算

```
bool isScramble(string s1, string s2) {
    int n = s1.size();
    if(n == 0 || s1.size() != s2.size() ) return false;
    vector<vector<vector<bool> > > dp(n, vector<vector<bool> >(n, vector
        <bool>(n+1, false)));
    for(int i = 0; i < n; i++)
        for(int j = 0; j < n; j++)
            dp[i][j][1] = s1[i] == s2[j];
    for(int l = 2; l < n+1; l++){
        for(int i = 0; i <= n - l; i++)
            for(int j = 0; j <= n - l; j++){
                for(int k = 1; k < l; k++){
                    dp[i][j][l] = dp[i][j][l]
                                || (dp[i][j][k] && dp[i+k][j+k][l-k])
                                || (dp[i][j+l-k][k] && dp[i+k][j][l-k]);
                }
            }
    }
    return dp[0][0][n];
}
```

这道题困难之处在于敢不敢这么简单粗暴的递推

15.3.7 Decode Ways

问题:

A message containing letters from A-Z is being encoded to numbers using the following mapping:

'A' → 1
'B' → 2
...
'Z' → 26
Given an encoded message containing digits, determine the total number of ways to decode it.*(leetcode 91)*

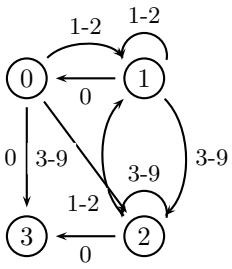
举例:

Given encoded message "12", it could be decoded as "AB" (1 2) or "L" (12).
The number of ways decoding "12" is 2.

DP : 时间复杂度O(n) , 空间复杂度O(n)

本题有两个需要注意的地方一个是检测非法输入，比如00这样的. 另外就是算出多少种解密方法.

- 检测非法输入: 非法输入的检测使用NFA自动机来实现，这样是最优美和最简单的，状态转移图如下,其中起始状态为0,非法状态为3:



- 计算解密数：设dp[i]表示s[0..i-1]可解密的数目，那么我们有递推公式如下:

$$dp[i] = \begin{cases} 1 & i = 0 \\ dp[i-2] & i! = 0 \ \ \& \ \ s[i-1] =' 0' \\ dp[i-1] + dp[i-2] & i > 1 \ \ \& \ \ s[i-2] =' 1' || (s[i-2] =' 2' \ \ \& \ \ s[i-1] <=' 6') \\ dp[i-1] & other \end{cases}$$

```
bool isValid(string &s){
    int size = s.size();
    int dfa[4][10] = {
        {3,1,1,2,2,2,2,2,2,2},
        {0,1,1,2,2,2,2,2,2,2},
        {3,1,1,2,2,2,2,2,2,2},
        {3,3,3,3,3,3,3,3,3,3}
    };
    int status = 0;
    for(int i = 0; i < size; i++)
        status = dfa[status][s[i]-'0'];
    return status != 3;
}

int numDecodings(string s) {
    int size = s.size();
    if(size == 0 || s[0] == '0' || !isValid(s)) return 0;
    vector<int> dp(size+1, 1);
    for(int i = 1; i < size; i++){
        if(s[i] == '0')
            dp[i+1] = dp[i-1];
        else
            if(s[i-1] == '1' || (s[i-1] == '2' && s[i] <= '6'))
                dp[i+1] = dp[i] + dp[i-1];
            else
                dp[i+1] = dp[i];
    }
    return dp[size];
}
```

这题递推公式很简单，但是考虑到非法输入比较麻烦，最简单的做法就是先检测一边，使用有限自动机是最优雅的检测手段

15.3.8 Interleaving String

问题:

Given s1, s2, s3, find whether s3 is formed by the interleaving of s1 and s2.*(leetcode 97)*

举例:

Given:
s1 = "aabcc",
s2 = "dbbca",

When s3 = "aadbbcbcac", return true.
When s3 = "aadbbbaccc", return false.

DP : 时间复杂度O(n*m) , 空间复杂度O(n*m)
我们假设dp[i][j]表示s1[0..i-1]与s2[0...j-1]是否能交叉构造出s3[0..i+j-1]

dp的递推关系式:

$$dp[i][j] = \begin{cases} s2[0..j-1] == s3[0..j-1] & i = 0 \\ s1[0..i-1] == s3[0..i-1] & j = 0 \\ (s1[i-1] == s3[i+j-1] \ \& \ dp[i-1][j]) || (s2[j-1] == s3[i+j-1] \ \& \ dp[i][j-1]) & other \end{cases}$$

```
bool isInterleave(string s1, string s2, string s3) {
    int n1 = s1.size(), n2 = s2.size(), n3 = s3.size();
    if(n1 + n2 != n3) return false;
    vector<vector<int> > dp(n1+1, vector<int>(n2+1, false));
    dp[0][0] = true;
    for(int i = 0; i < n1; i++)
        if(s1[i] == s3[i])  dp[i+1][0] = true;
        else    break;
    for(int i = 0; i < n2; i++)
        if(s2[i] == s3[i])  dp[0][i+1] = true;
        else    break;
    for(int i = 0; i < n1; i++)
        for(int j = 0; j < n2; j++){
            dp[i+1][j+1] = (s1[i] == s3[i+j+1] && dp[i][j+1])
                           || (s2[j] == s3[i+j+1] && dp[i+1][j]);
        }
    return dp[n1][n2];
}
```

15.3.9 Distinct Subsequences

问题:
Given a string S and a string T, count the number of distinct subsequences of T in S.

A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (ie, "ACE" is a subsequence of "ABCDE" while "AEC" is not).*(leetcode 115)*

举例:
S = "rabbbit", T = "rabbit"

Return 3.

DP : 时间复杂度O(n*m) , 空间复杂度O(n*m)
我们假设dp[i][j]表示S[0..i-1]能抽出T[0..j-1]的种类数,那么target = dp[n][m]
dp的递推关系式:

$$dp[i][j] = \begin{cases} 0 & i < j \\ 1 & j = 0 \\ dp[i-1][j-1] + dp[i-1][j] & S[i-1] == T[j-1] \\ dp[i-1][j] & other \end{cases}$$

```
int numDistinct(string S, string T) {
    int n = S.size();
    int m = T.size();
    vector<vector<int> > dp(n+1, vector<int>(m+1, 0));
    for(int j = 0; j < m; j++)
        for(int i = j; i < n; i++){
            if(S[i] == T[j]){
                dp[i+1][j+1] = dp[i][j+1] + (j == 0? 1 : dp[i][j]);
            }else{
                dp[i+1][j+1] = dp[i][j+1];
            }
        }
    return dp[n][m];
}
```

这题和*LCS*问题很像

15.3.10 Best Time to Buy and Sell Stock III

问题:

Say you have an array for which the ith element is the price of a given stock on day i.
Design an algorithm to find the maximum profit. You may complete at most two transactions.*(leetcode 123)*

Note:

You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

DP : 时间复杂度O(n) , 空间复杂度O(n)

我们可以试着这样考虑问题: 由于能买两次，所以可以考虑将prices[0...n-1]插入n+1个分界点，每选取一个都可以将prices数组分为两份，我们可以在这两份子数组中分别求出一次购买的最大利润，然后两者相加就是此分界点确定的两次购买最大利润． 我们的最优解也必然是先是一次购买又是一次购买(如果只需要一次购买，可以想成是有一个第二次购买且是当天购买当天卖出收益0这种情况),最优解的购买方式也必然是被某个(或者多个)分界点确定的，所以我们的目标就是求出每个分界点确定的最大利润，然后在这里面取最大的那个就行了.

我们假设dp[i]为分界点i,将prices数组分为prices[0..i-1]和prices[i..n-1],这两个子数组的单次购买最大利润值分别标记为left[i]和right[i]

那么有递推关系式:
假设left_min为prices[0...i-1]中的最小值

$$left[i] = \begin{cases} 0 & i = 0 \\ max(left[i-1], prices[i-1] - left\_min) & other \end{cases}$$

假设right_max为prices[i..n-1]中的最大值

$$right[i] = \begin{cases} 0 & i = n \\ max(right[i+1], right\_max - prices[i-1]) & other \end{cases}$$

$$dp[i] = left[i] + right[i]$$

```
int maxProfit(vector<int> &prices) {
    int n = prices.size();
    vector<int> left(n+1, 0);
```

```
4       //left[i表示]a[0],...a[i单次购买最大利润-1]
5       vector<int> right(n+1, 0);
6       //right[i表示]a[i],...a[n单次购买最大利润-1]
7       int left_min = INT_MAX, right_max = INT_MIN;
8       for(int i = 1; i <= n; i++){
9           left_min = min(prices[i-1], left_min);
10          left[i] = max(left[i-1], prices[i-1] - left_min);
11      }
12      for(int i = n-1; i >= 0; i--){
13          right_max = max(prices[i], right_max);
14          right[i] = max(right_max - prices[i], right[i+1]);
15      }
16      //以为切割点，求左右两部分各自最大利润再综合i
17      int maxEarn = INT_MIN;
18      for(int i = 0; i <= n; i++){
19          maxEarn = max(maxEarn, left[i] + right[i]);
20      }
21      return maxEarn;
22  }
```

这道题先采用分治的做法把区间两次股票问题转为区间单次股票问题，区间单次股票问题很简单，参见*Best_Time_Buy_and_Sell_Stock_I*

15.3.11 Best Time to Buy and Sell Stock IV

问题:

Say you have an array for which the ith element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete at most k transactions.
(leetcode 188)

Note:

You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

DP : 时间复杂度O(kn), 空间复杂度O(n)

首先需要考虑一种特殊情况: $k > price.size()/2$, 这种情况就是等同于你可以购买无数次，也就是你需要把每个增长区间都计算进去就可以了.这样就可以在O(n)时间解决这种情况而不是O(kn).
对于其他情况，我们假设left[i]表示prices[i]的左边连续最小值的位置，dp[i][p]表示至多购买p次并且以prices[i]为最后一次卖出点的最大利润，r[i][p]表示至多购买p次并且最后一次卖出点在prices[i]之前的最大利润，所以target = r[n-1][k]
我们可以有递推公式

$$left[i] = \begin{cases} 0 & i = 0 \\ left[i-1] & prices[i] >= prices[i-1] \\ i & prices[i] < prices[i-1] \end{cases}$$

$$dp[i][p] = \begin{cases} prices[i] - prices[0] & left[i] = 0 \\ max(prices[i] - prices[j] + r[j-1][p-1], prices[i] - prices[j-1] + dp[j-1][p]) & other \end{cases}$$

其中$j = left[i]$

$$r[i][p] = \begin{cases} 0 & p = 0 || r = 0 \\ max(r[i-1][p], dp[i][p]) & other \end{cases}$$

从递推公式可以看出,dp和r可以状态压缩以节省空间

```
int maxProfit(int k, vector<int>& prices) {
    int n = prices.size();
    // simple case, very important
    if (k > n/2){
        int ans = 0;
        for (int i=1; i<n; ++i){
            ans += max(prices[i] - prices[i-1],0);
        }
        return ans;
    }
    k = min(k, n/2);
    if(n <= 1)  return 0;
    vector<int> dp(n, 0), r(n, 0), left(n,0);
    for(int i = 1; i < n; i++){
        if(prices[i] >= prices[i-1]){
            left[i] = left[i-1];
        }
        else{
            left[i] = i;
        }
    }
    for(int p = 1; p <= k; p++){
        for(int i = 1; i < n; i++){
            int j = left[i];
            if(j == 0)  dp[i] = prices[i] - prices[0];
            else
                dp[i] = max(prices[i] - prices[j-1] + dp[j-1],
                            prices[i] - prices[j] + r[j-1]);
        }
        for(int i = 1; i < n; i++)
            r[i] = max(r[i-1], dp[i]);
    }
    return r[n-1];
}
```

15.3.12 Palindrome Partitioning II

问题:

Given a string s, partition s such that every substring of the partition is a palindrome. Return the minimum cuts needed for a palindrome partitioning of s.*(leetcode 132)*

举例:

given s = "aab",
Return 1 since the palindrome partitioning ["aa","b"] could be produced using 1 cut.

DP : 时间复杂度O(n^2) , 空间复杂度O(n^2)

我们假设dp[i]表示s[0..i-1]最小切分下的段数,那么tgarget = dp[n] - 1.

dp的递推关系式:

$$dp[i] = \begin{cases} 0 & i = 0 \\ mindp[k] + 1v - 0 <= k < i\tilde{}s[k+1...i]ispalindrome & other \end{cases}$$

```cpp
int minCut(string s) {
    int n = s.size();
    vector<vector<bool> > palin(n, vector<bool>(n, false));
    for(int i = 0; i < n; i++){
        for(int j = 0; i + j < n && i - j >= 0; j++){
            if(s[i-j] == s[i+j]) palin[i-j][i+j] = true;
            else break;
        }
    }
    for(int i = 0; i < n-1; i++){
        for(int j = 0; i + j < n - 1 && i - j >= 0; j++){
            if(s[i-j] == s[i+j+1]) palin[i-j][i+j+1] = true;
            else break;
        }
    }
    vector<int> dp(n+1, 0);
    for(int i = 0; i < n; i++){
        int minCur = INT_MAX;
        for(int j = 0; j <= i; j++){
            if(palin[j][i]){
                minCur = min(minCur, dp[j] + 1);
            }
        }
        dp[i+1] = minCur;
    }
    return dp[n] - 1;
}
```

这题很有普世性，都是一维的*dp*,但是对于*dp[i]*需要选取前面所有合适的候选中的最佳候选而不是直接得出最佳候选，所以时间复杂度就是$O(n^2)$

15.3.13 Word Break

问题:

Given a string s and a dictionary of words dict, determine if s can be segmented into a space-separated sequence of one or more dictionary words.*(leetcode 139)*

举例:

s = "leetcode",
dict = ["leet", "code"].
Return true because "leetcode" can be segmented as "leet code".

DP : 时间复杂度O(n^2) , 空间复杂度O(n)

假设dp[i]表示s[0...i-1]能否由词典dict里面的词构成，target = dp[n]

dp的递推关系式:

$$dp[i] = \begin{cases} true & i = 0 \\ \prod_{k=0}^{i-1} dp[k] \ \ \& \ \ s[k+1..i] \in dict & other \end{cases}$$

其中上面的求积符号表示逻辑或.

```cpp
bool wordBreak(string s, unordered_set<string> &dict) {
    int n = s.size();
    vector<bool> dp(n+1, false);
    dp[0] = true;
    for(int i = 0; i < n; i++){
```

```
6           for(int j = i; j >= 0; j--){
7               if(dict.count(s.substr(j, i - j + 1)) != 0){
8                   dp[i+1] =  dp[j];
9                   if(dp[i+1]) break;
10              }
11          }
12      }
13      return dp[n];
14  }
```

这题也是一维*dp*问题，对于每个*dp[i]*,同样也是综合所有候选方案确定，时间复杂度为$O(n^2)$

15.3.14 Word Break II

问题:

Given a string s and a dictionary of words dict, add spaces in s to construct a sentence where each word is a valid dictionary word.
Return all such possible sentences.*(leetcode 140)*

举例:

s = "catsanddog",
dict = ["cat", "cats", "and", "sand", "dog"].

A solution is ["cats and dog", "cat sand dog"].

DP : 时间复杂度O(n^2) , 空间复杂度O(n^2)

假设dp[i]表示s[0...i-1]由词典dict里面的词构成所有可能，target = dp[n]

dp的递推关系式:

$$dp[i] = \begin{cases} "" & i = 0 \\ \bigcup_{k=0}^{i-1} ele + s[k+1..i], & ele \in dp[k] \ \ \& \ \ s[k+1..i] \in dict \quad other \end{cases}$$

```
1   vector<string> wordBreak(string s, unordered_set<string> &dict) {
2       int n = s.size();
3       vector<vector<string> > dp(n+1, vector<string>());
4       dp[0].push_back("");
5       for(int i = 0; i < n; i++)
6           for(int j = i; j >= 0; j--){
7               string tail = s.substr(j , i - j + 1);
8               if(dict.count(tail) != 0){
9                   for(int k = 0; k < dp[j].size(); k++){
10                      dp[i+1].push_back(dp[j][k] + " " + tail);
11                  }
12              }
13          }
14      for(int i = 0; i < dp[n].size(); i++)
15          dp[n][i].erase(dp[n][i].begin());
16      return dp[n];
17  }
```

这题这样解在leetcode上通不过，因为空间复杂度太大...,于是可以使用下面的这种方法

DP+DFS : 时间复杂度O($n!$) , 空间复杂度O(n)

假设dp[i]表示s[0...i-1]由词典dict里面的词构成的每种可能，但是不像上面记下整个切分的

词，而是只记录最后一个切分词的位置

dp的递推关系式:

$$dp[i] = \begin{cases} -1 & i = 0 \\ \bigcup_{k=0}^{i} k, & dp[k] \notin \emptyset \ \& \ s[k+1..i] \in dict \quad other \end{cases}$$

有了上面的切分之后就可以使用DFS来自下而上的构造最终解，虽然这里面可能会有很多重复计算，但是空间复杂度很低了...

```
void genPath(string &s, vector<vector<int> > &dp, vector<string> &ret,
        int pos, vector<string>& path){
    if(pos < 0){
        int size = path.size();
        string cur = path[size-1];
        for(int i = size - 2; i >= 0; i--){
            cur += " " + path[i];
        }
        ret.push_back(cur);
        return;
    }
    for(int i = 0; i < dp[pos+1].size(); i++){
        path.push_back(s.substr(dp[pos+1][i], pos - dp[pos+1][i] + 1));
        genPath(s, dp, ret, dp[pos+1][i] - 1, path);
        path.pop_back();
    }
}

vector<string> wordBreak(string s, unordered_set<string> &dict) {
    int n = s.size();
    vector<vector<int> > dp(n+1, vector<int>());
    dp[0].push_back(-1);
    for(int i = 0; i < n; i++)
        for(int j = i; j >= 0; j--){
            string tail = s.substr(j , i - j + 1);
            if(dict.count(tail) != 0 && dp[j].size() != 0){
                dp[i+1].push_back(j);
            }
        }
    vector<string> ret;
    vector<string> path;
    genPath(s, dp, ret, n-1, path);
    return ret;
}
```

这题按理说还是使用第一种方法比较好，这是空间换取时间的做法，无奈*leetcode*上限制了内存使用只能使用第二种方法了...

15.3.15 Maximum Product Subarray

问题:

Find the contiguous subarray within an array (containing at least one number) which has the largest product.*(leetcode 152)*

举例:

given the array [2,3,-2,4],
the contiguous subarray [2,3] has the largest product = 6.

DP : 时间复杂度O(n)，空间复杂度O(n)

我们假设maxCon[i]表示以A[i]为结尾的最大连续子串积，minCon[i]表示以A[i]为结尾的最小连续子串积． 那么target = max{maxCon[i]}.

maxCon和minCon的递推关系式:

$$maxCon[i] = \begin{cases} A[0] & i = 0 \\ max(A[i], maxCon[i-1] * A[i]) & i! = 0 \ \ \& \ \ A[i] >= 0 \\ max(A[i], minCon[i-1] * A[i]) & other \end{cases}$$

$$minCon[i] = \begin{cases} A[0] & i = 0 \\ min(A[i], minCon[i-1] * A[i]) & i! = 0 \ \ \& \ \ A[i] >= 0 \\ min(A[i], maxCon[i-1] * A[i]) & other \end{cases}$$

```
int maxProduct(int A[], int n) {
    if(n <= 0)  return 0;
    vector<int> maxCon(n, A[0]);
    vector<int> minCon(n, A[0]);
    for(int i = 1; i < n; i++){
        if(A[i] >= 0){
            maxCon[i] = max(maxCon[i-1]*A[i], A[i]);
            minCon[i] = min(minCon[i-1]*A[i], A[i]);
        }else{
            maxCon[i] = max(minCon[i-1]*A[i], A[i]);
            minCon[i] = min(maxCon[i-1]*A[i], A[i]);
        }
    }
    return *max_element(maxCon.begin(), maxCon.end());
}
```

这题和最大连续子串和几乎一样，唯一区别是需要维护两个动规量，思路都是列出所有结尾的可能得到的值，取其中最大的一个

15.3.16 Dungeon Game

问题:

The demons had captured the princess (P) and imprisoned her in the bottom-right corner of a dungeon. The dungeon consists of M x N rooms laid out in a 2D grid. Our valiant knight (K) was initially positioned in the top-left room and must fight his way through the dungeon to rescue the princess.

The knight has an initial health point represented by a positive integer. If at any point his health point drops to 0 or below, he dies immediately.

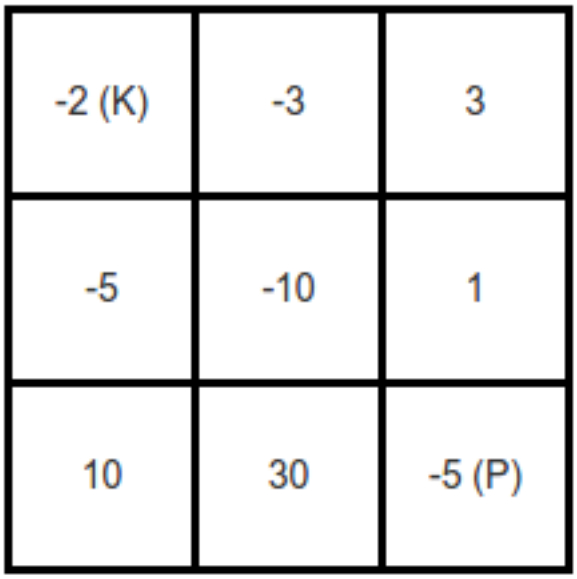
Some of the rooms are guarded by demons, so the knight loses health (negative integers) upon entering these rooms; other rooms are either empty (0's) or contain magic orbs that increase the knight's health (positive integers).

In order to reach the princess as quickly as possible, the knight decides to move only rightward or downward in each step.

Write a function to determine the knight's minimum initial health so that he is able to rescue the princess.
(leetcode 174)

举例:

Given the dungeon below, the initial health of the knight must be at least 7 if he follows the optimal path RIGHT → RIGHT → DOWN → DOWN.



Note:

The knight's health has no upper bound.

Any room can contain threats or power-ups, even the first room the knight enters and the bottom-right room where the princess is imprisoned.

DP : 时间复杂度O(n^2), 空间复杂度O(n^2)

这道题很显然会想到用DP来做，但是我一开始是从出发点开始构造DP推导公式，但是发现这样会有问题，因为你无法仅仅根据开始点到当前点的情况的就能判断出每步该选择向右还是向下，所以稍微改变一下思路，可以从结束点倒过来DP,这样做的好处是，对于每个待判断的点，它已经知道向右或者向下的将来代价，只需要选这两个较小的那个就好，因为无论向右还是向下，之前的路都是一样的，所以全局最优解就向右和向下这个两个子最优解的综合.

我们假设dp[i][j]代表从(i, j)到达终点的所需最小血值，所以我们有target = dp[0][0]

递推公式如下:

$$dp[i][j] = \begin{cases} makeDp(dungeon_{i,j}) & i = n-1, j = m-1 \\ makeDp(dungeon_{i,j} - dp[i][j+1]) & i = n-1 \\ makeDp(dungeon_{i,j} - dp[i+1][j]) & j = m-1 \\ min(makeDp(dungeon_{i,j} - dp[i][j+1]), makeDp(dungeon_{i,j} - dp[i+1][j])) & other \end{cases}$$

makeDp is a function as the follow:

```
int makeDp(int data){
    return data < 0? -data : 1;
}

int calculateMinimumHP(vector<vector<int> > &dungeon) {
    int n = dungeon.size();
    if(n == 0)  return 0;
    int m = dungeon[0].size();
    vector<vector<int> > dp(dungeon);
    for(int i = n - 1; i >= 0; i--)
```

```
11          for(int j = m - 1; j >= 0; j--){
12              if( i == n - 1 && j == m - 1)
13                  dp[i][j] = dungeon[i][j] < 0? -dungeon[i][j] + 1 : 1;
14              else{
15                  if(i == n - 1){
16                      dp[i][j] = makeDp(dungeon[i][j] - dp[i][j+1]);
17                  }else if(j == m - 1){
18                      dp[i][j] = makeDp(dungeon[i][j] - dp[i+1][j]);
19                  }else{
20                      int right = makeDp(dungeon[i][j] - dp[i][j+1]);
21                      int down = makeDp(dungeon[i][j] - dp[i+1][j]);
22                      dp[i][j] = min(right, down);
23                  }
24              }
25          }
26      return dp[0][0];
27  }
```

第 16 章　位操作

16.1　基本概念

位操作包括左移，右移，与，或，非和亦或等.其中C++中的右移都是算术右移，即首位补符号位．还有一点需要注意的是，我们的这些位操作的操作对象都是补码，比如$-1 \wedge 2$其实就是$0xffffffff \wedge 0x2$.

16.2 经典问题

位操作有很多经典问题，比如将最低位的1设置为0问题: 我们有变量x,现在我们希望设置x最低位的1为0,比如x = 0xff0ff010, 那么设置之后就是x = 0xff0ff000. 最简单的做法就是$x = x \ \& (x - 1)$;

16.3 相关问题

位操作通常适于那些每个位上都有显著特征的问题，比如每个位上的0,1分布是成对出现啊，0和1出现的哪个多就取哪个啊等等.

16.3.1 Single Number

问题:

Given an array of integers, every element appears twice except for one. Find that single one.
(leetcode 136)

Note: Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?

亦或：时间复杂度O(n), 空间复杂度O(1)

这道题是很经典的一道题，主要是需要想出一个或者一组操作使得相同的元素能够彼此清除掉而只剩下那一个唯一的操作.我们可以想一下，相同的元素在每个位上都是相同的要么都是0要么都是1,所以我们想到了亦或操作.

```
int singleNumber(vector<int>& nums) {
    int result = 0;
    for(int i = 0; i < nums.size(); i++)
        result ^= nums[i];
    return result;
}
```

16.3.2 Single Number II

问题:

Given an array of integers, every element appears three times except for one. Find that single one.
(leetcode 137)

Note:

Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?

：时间复杂度O(n), 空间复杂度O(1)

这题和上题不同，这题相同的元素是有三个．我们不得不寻找一个更加普适的方法，我们知道三个一样的数在每一位上要么都是1要么都是0,所以我们可以对于每个位,求出这些1和0的和，然后模上3,那么三三一样的就会被剔除，剩余的余数就是那个单独的数．这个方法不仅适用于3个数一样的问题，还适用于4,5,...个数一样的问题，当然两个数一样的问题，只不过那题可以用更简洁的亦或做.

```
int singleNumber(vector<int>& nums) {
    int result = 0;
    vector<int> b(32, 0);
    for(int i = 0; i < 32; i++){
        for(int j = 0; j < nums.size(); j++){
            b[i] += (((nums[j] >> i) & 0x1) != 0);
        }
        b[i] %= 3;
        result += (b[i] << i);
    }
    return result;
}
```

16.3.3 Majority Element

问题:

Given an array of size n, find the majority element. The majority element is the element that appears more than $n/2$ times.

You may assume that the array is non-empty and the majority element always exist in the array.
(leetcode 169)

记数法：时间复杂度O(n), 空间复杂度O(1)

这题我最先了解到的解法就是记数法，因为主元素出现的次数多于总个数的一半，所以我们可以放着两个椅子，然后让元素去争抢：对于每个新加入的元素，如果椅子上有与它相同的元素，那么该椅子记数+1;如果没有，此时如果有空闲椅子，则该元素抢占该椅子，然后置此椅子记数为1;如果没有空闲椅子，那么舍弃该元素，并且两个椅子的记数都-1,如果此时某个椅子记数为0,则置此椅子为空闲.
最后，椅子上记数较大的那个元素就是主元素.
这个方法的证明也很简单，我们可以试想一下，由于主元素个数超过一半，又有两个椅子，所以主元素必然在最终的一个之一，而且另一个椅子的记数也绝不会超过它.

```
int majorityElement(vector<int>& nums) {
    if(nums.size() == 0)    return -1;
    int p1 = nums[0], p2, c1 = 1, c2 = 0;
    for(int i = 1; i < nums.size(); i++){
        if(c1 && nums[i] == p1) c1++;
        else    if(c2 && nums[i] == p2) c2++;
        else    if(!c1){
            p1 = nums[i];
            c1++;
        }
        else    if(!c2){
            p2 = nums[i];
            c2++;
        }else{
            c1--;
            c2--;
        }
    }
    return c1 > c2? p1 : p2;
}
```

位操作：时间复杂度O(n), 空间复杂度O(1)

这题还可以这么考虑：我们对于数组中每个元素分析它们的每个位，由于主元素个数超半，所以每个位的0,1个数必然被主元素控制，如果主元素该位为1那么1的个数必然超过一半，反之亦然.所以我们可以基于这个原则设计算法.

```
int majorityElement(vector<int>& nums) {
    int result = 0, n = nums.size();
    for(int i = 0; i < 32; i++){
        int bit = 0;
        for(int j = 0; j < n; j++)
            bit += ((nums[j] & (0x1 << i))? 1 : 0);
        result += ((bit > n/2? 1 : 0) << i);
    }
    return result;
}
```

16.3.4 Reverse Bits

问题:

Reverse bits of a given 32 bits unsigned integer.
(leetcode 190)

举例:

Given input 43261596 (represented in binary as 00000010100101000001111010011100), return 964176192 (represented in binary as 00111001011110000010100101000000).

Follow Up:

If this function is called many times, how would you optimize it?

位操作：时间复杂度O(1), 空间复杂度O(1)

这题的Follow Up很值得注意，最笨的做法就是建立索引表，下次查找就不用再计算了.

```
uint32_t reverseBits(uint32_t n) {
    uint32_t result = 0;
    for(int i = 0; i < 16; i++){
        bool front = n & (0x1 << (31 - i));
        bool back = n & (0x1 << i);
        result = result | (back << (31 - i));
        result = result | (front << i);
    }
    return result;
}
```

位操作：时间复杂度O(1), 空间复杂度O(1)

```
uint32_t reverseBits(uint32_t n) {
    n=((n & 0xffff0000) >> 16)| ((n & 0x0000ffff) << 16);
    n=((n & 0xff00ff00) >> 8) | ((n & 0x00ff00ff) << 8);
    n=((n & 0xf0f0f0f0) >> 4) | ((n & 0x0f0f0f0f) << 4);
    n=((n & 0xcccccccc) >> 2) | ((n & 0x33333333) << 2);
    n=((n & 0xaaaaaaaa) >> 1) | ((n & 0x55555555) << 1);
    return n;
}
```

16.3.5 Number of 1 Bits

问题:

Write a function that takes an unsigned integer and returns the number of '1' bits it has (also known as the Hamming weight).
(leetcode 191)

举例:

The 32-bit integer '11' has binary representation 00000000000000000000000000001011, so the function should return 3.

位操作：时间复杂度O(1), 空间复杂度O(1)

这题用经典问题x & $(x-1)$即可

```
int hammingWeight(uint32_t n) {
    int count = 0;
    while(n){
        n = (n & (n - 1));
        count++;
```

```
6       }
7       return count;
8   }
```

16.3.6 Bitwise AND of Numbers Range

问题:

Given a range $[m, n]$ where 0 ¡= m ¡= n ¡= 2147483647, return the bitwise AND of all numbers in this range, inclusive.
(leetcode 201)

举例:

Given the range $[5, 7]$, you should return 4.

位操作：时间复杂度O(n), 空间复杂度O(1)

首先声明高位区间:高位都一样,低位从0x0到0xff..f区间.比如$[0xc1a000, 0xc1afff]$
然后我们有一个规律，在区间$[a, b]$中如果gap = b - a + 1小于某个高位区间的大小，那么a和b的对应高位是一样的. 比如区间[13, 16]中gap = 16 - 13 + 1 = 4 ≤ 8,所以高29位是一样的.
基于上面的规律，不断的扩大高位区间直到gap大于高位区间大小，这样就确定了哪些高位是一样的.

```
1   int rangeBitwiseAnd(int m, int n) {
2       if(m == 0 && n == INT_MAX)  return 0;
3       int gap = n - m + 1;
4       if(gap <= 1)    return n;
5       int pos = 0;
6       while(pos < 32 && ((1 << pos) - (m & (~(0xffffffff - ((1 << pos) -
            1)))) < gap))
7           pos++;
8       return n & (0xffffffff - ((1 << pos) - 1));
9   }
```

16.3.7 Repeated DNA Sequences

问题:

All DNA is composed of a series of nucleotides abbreviated as A, C, G, and T, for example: "ACGAATTCCG". When studying DNA, it is sometimes useful to identify repeated sequences within the DNA.

Write a function to find all the 10-letter-long sequences (substrings) that occur more than once in a DNA molecule.
(leetcode 187)

举例:

Given s = "AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT",

Return:
["*AAAAACCCCC*","*CCCCCAAAAA*"].

Hash+bit：时间复杂度O(n), 空间复杂度O(n)

这题一般会想到使用hash来统计是否重复，问题是hash的key值怎么表示才是体现水准的关键，笔者一开始想到的就是把A,C,G,T分别编号问0,1,2,3然后将序列化成4进制的数. 但是还有一种非常巧妙的做法，它的原理如下:

A,C,G,T的Asci码分别是0x41, 0x43, 0x47和0x54.这几个数的二进制表示中倒数第三和倒数第二位分别是00, 01, 11和10,所以可以利用这两位产生key

```
vector<string> findRepeatedDnaSequences(string s) {
    vector<string> result;
    unordered_map<int, int> table;
    for(int i = 0; i + 9 < s.size(); i++){
        unsigned int code = 0x0;
        for(int j = 0; j < 10; j++)
            code |= (((s[i+j] & 0x6) >> 1) << (2*j));
        if(table[code] == 1) result.push_back(s.substr(i,10));
        table[code]++;
    }
    return result;
}
```

第 17 章　数学

17.1　基本概念

TODO

17.2 经典问题

17.2.1 Gcd

问题:

Given you two integer, return the greatest common divisor of them.

欧几里得算法：时间复杂度O(1), 空间复杂度O(1)

欧几里得算法又称辗转相除法.假设a ¿= b, 那么$gcd(a,b) = gcd(a \mod b, b)$直到a == 0 或者b == 0时候另一个非零值就是gcd(a,b)

```
//assume a >= b
unsigned int gcd(unsigned int a, unsigned int b){
    if (b == 0) return a;
    return gcd(b, a % b);
}
```

17.2.2 Lcm

问题:

Given you two integer, return the lowwest common multiple of them.

欧几里得算法：时间复杂度O(1), 空间复杂度O(1)

我们有结论lcm(a.b) = a*b / gcd(a, b), 所以实质上还是求gcd

```
//assume a >= b
unsigned int gcd(unsigned int a, unsigned int b){
    if (b == 0) return a;
    return gcd(b, a % b);
}
unsigned int lcm(unsigned int a, unsigned int b){
    if(a < b)   return lcm(b, a);
    return a / gcd(a, b) * b;
}
```

17.2.3 Prime Number

问题:

Given you one positive integer, determine if it is a prime number.

Eratosthenes 筛法：时间复杂度O(???), 空间复杂度O(1)

这里介绍的是使用素数表筛选判断整数n是否是素数，那么问题来了：怎么构造素数表呢？这里就是采用边筛边构造素数表的方法．从素数表初始化集合只有元素 2 ，然后使用 2 筛掉$[3,n]$区间中的所有2的倍数，然后判断3是否被筛掉，如果没有则将3加入素数表，继续新一轮筛；如果被筛掉了，则继续判断4,5...直到$\sqrt{n}$

```
#define MAXN 30000

/** prime_table[i]==1 表示 i 是素数等于, 0 则不是素数 */
int prime_table[MAXN+1];

void compute_prime_table() {
    int i, j;
    const int upper = sqrt(MAXN);
    for (i = 2; i <= MAXN; i++)
```

```
        prime_table[i] = 1;
    prime_table[0] = 0;
    prime_table[1] = 0;
    for (i = 2; i < upper; i++)
        if(prime_table[i]) {
            for (j = 2; j * i <= MAXN; j++)
                prime_table[j*i] = 0;
        }
}

int is_prime(unsigned int n) {
    return prime_table[n];
}
```

17.2.4 Next Permutation

问题:

Implement next permutation, which rearranges numbers into the lexicographically next greater permutation of numbers.

If such arrangement is not possible, it must rearrange it as the lowest possible order (ie, sorted in ascending order).

The replacement must be in-place, do not allocate extra memory.
(leetcode 31)

举例:

Inputs are in the left-hand column and its corresponding outputs are in the right-hand column.
1,2,3 → 1,3,2
3,2,1 → 1,2,3
1,1,5 → 1,5,1

组合数学：时间复杂度O(n), 空间复杂度O(1)

这是组合数学中的经典问题，假设我们现在的数是$a_1, a_2, ...a_n$,我们找该序列最后一个极大点,设为a_k,我们知道$a_{k-1}<a_k$，$a_k>a_{k+1}>...>a_n$,另外设a_m是$a_k, ..., a_n$中大于a_{k-1}的最小的那个,那么下一个数就是$a_1, a_2, ...a_m, a_k, a_{k+1}, ..., a_{m-1}, a_{k-1}, a_{m+1}, ..., a_n$

```
void nextPermutation(vector<int> &num) {
    int pos = num.size() - 1;
    while(pos > 0 && num[pos - 1] >= num[pos])  pos--;
    if(pos != 0){
        int pre = pos - 1;
        auto index = lower_bound(num.begin() + pos, num.end(), num[pre],
             greater<int>());
        swap(num[pre], num[index - num.begin() - 1]);
    }
    reverse(num.begin() + pos, num.end());
}
```

17.3 相关问题

17.3.1 Reverse Integer

问题:

Reverse digits of an integer.
(leetcode 7)

举例:

Example1: x = 123, return 321
Example2: x = -123, return -321

Note:

Here are some good questions to ask before coding. Bonus points for you if you have already thought through this!

If the integer's last digit is 0, what should the output be? ie, cases such as 10, 100.

Did you notice that the reversed integer might overflow? Assume the input is a 32-bit integer, then the reverse of 1000000003 overflows. How should you handle such cases?

For the purpose of this problem, assume that your function returns 0 when the reversed integer overflows.

Math : 时间复杂度O(1), 空间复杂度O(1)

这题有两个点需要注意，一个是正数与负数怎么表示？我这里采用全化为负数处理，简单明了；另一个问题就是溢出预先判断，可以在乘之前判断一下

```
int reverse (int x) {
    if(x > 0)   return -reverse(-x);
    int rever = 0;
    while(x != 0){
        if(rever < (INT_MIN - x%10) / 10)
            return 0;
        rever = rever * 10 + x % 10;
        x = x / 10;
    }
    return rever;
}
```

17.3.2 Palindrome Number

问题:

Determine whether an integer is a palindrome. Do this without extra space.
(leetcode 9)

Math : 时间复杂度O(1), 空间复杂度O(1)

我们可以只反转低半部分，然后判断反转的低半部分是否与高半部分相符即可

```
bool isPalindrome(int x) {
    if(x < 0 || (x != 0 && x % 10  == 0)) return false;
    int rev = 0;
    while(x > rev){
        rev = rev * 10 + (x % 10);
        x = x / 10;
```

```
7       }
8       return (x == rev) || (x == rev/10);
9   }
```

17.3.3 Multiply Strings

问题:

Given two numbers represented as strings, return multiplication of the numbers as a string. *(leetcode 43)*

Note:

Note: The numbers can be arbitrarily large and are non-negative.

大数：时间复杂度O(n*m), 空间复杂度O(n*m)

这题是典型的大数问题，对于大数问题一般采用vector或者string来存储大数，需要注意高低位的顺序，其运算方式完全按照小学学习的加减乘除计算方法计算就可以了.

```
1   // num1为低位[0],num2为低位[0], num1 = num1 + (num2 << left)
2   void bigAdd(string &num1, string &num2, int left){
3       string fill = string(left, '0');
4       num2 = fill + num2;
5       int cin = 0;
6       for(int i = 0; i < num1.size(); i++){
7           int out = cin + (num1[i] - '0') + (num2[i] - '0');
8           num1[i] = (out % 10) + '0';
9           cin = out / 10;
10      }
11  }
12
13  // num1为低位[0]
14  void bigMul(string &result, const string &num1, const char num2){
15      int cin = 0;
16      int mul = num2 - '0';
17      for(int i = 0; i < num1.size(); i++){
18          int out = cin + mul * (num1[i] - '0');
19          result[i] = (out % 10) + '0';
20          cin = out / 10;
21      }
22      if(cin) result[num1.size()] = cin + '0';
23  }
24
25  string multiply(string num1, string num2) {
26      int n1 = num1.size(), n2 = num2.size();
27      if(n1 < n2) return multiply(num2, num1);
28      reverse(num1.begin(), num1.end());
29      reverse(num2.begin(), num2.end());
30      int n = n1 + n2;
31      vector<string> line(n2, string(n, '0'));
32      string result(n, '0');
33      for(int i = 0; i < n2; i++){
34          bigMul(line[i], num1, num2[i]);
35          bigAdd(result, line[i], i);
36      }
37      reverse(result.begin(), result.end());
38      int nozero = 0;
39      while(nozero < result.size() && result[nozero] == '0')
40          nozero++;
```

```
41      return nozero == result.size()? "0" : result.substr(nozero, result.
            size() - nozero);
42  }
```

17.3.4 Plus One

问题:

Given a non-negative number represented as an array of digits, plus one to the number. The digits are stored such that the most significant digit is at the head of the list.
(leetcode 66)

大数：时间复杂度O(n), 空间复杂度O(n)

典型的大数问题

```
1   vector<int> plusOne(vector<int> &digits) {
2       int cin = 1, n = digits.size();
3       vector<int> result;
4       for(int i = n - 1; i >= 0; i--){
5           result.push_back((digits[i] + cin) % 10);
6           cin = (digits[i] + cin) / 10;
7       }
8       if(cin) result.push_back(1);
9       reverse(result.begin(), result.end());
10      return result;
11  }
```

17.3.5 Max Points on a Line

问题:

Given n points on a 2D plane, find the maximum number of points that lie on the same straight line.
(leetcode 149)

Hash+Gcd：时间复杂度O(n^2), 空间复杂度O(n)

这题思路就是选取每个点作为直线上的点，然后计算其他点与该点形成的斜率，斜率一样的说明这些点和该点在同一条直线上．问题就编程了怎么记录这些斜率一样的信息，我们想到使用hash记录，那么这个斜率自然就是key值，这个key值不能是整数，也不能是浮点数(会有误差)，只能用最简分数表示，如何化简分数那就使用gcd

```
1   //x,最大公约数，如果或为yxy0返回,或yx
2   int gcd(int x, int y){
3       x = abs(x);
4       y = abs(y);
5       while(x && y){
6           if(x < y) y = y - x;
7           else x = x - y;
8       }
9       return x != 0? x : y;
10  }
11
12  struct hashKey{
13      std::size_t operator()(const pair<int,int> &x) const{
14          using std::hash;
15          return x.first ^ x.second;
16      }
```

```
};

//pair<int,int表示斜率>,(0,0)表示重点,(0,1)表示平行与轴y,(1,0)表示平行于轴x
int maxPoints(vector<Point> &points) {
    int result = 0;
    for(int i = 0; i < points.size(); i++){
        unordered_map<pair<int,int>, int, hashKey> table;
        int cur = 0, base = 0;
        for(int j = 0; j < points.size(); j++){
            int dx = points[j].x - points[i].x;
            int dy = points[j].y - points[i].y;
            if(dx == 0 && dy == 0){
                base++;
                continue;
            }
            pair<int, int> k(make_pair(dx/gcd(dx, dy), dy/gcd(dx,dy)));
            if(table.count(k))  table[k]++;
            else table[k] = 1;
        }
        for(auto iter = table.begin(); iter != table.end(); iter++)
            if(cur < iter->second) cur = iter->second;
        result = max(result, base + cur);
    }
    return result;
}
```

17.3.6 Fraction to Recurring Decimal

问题:

Given two integers representing the numerator and denominator of a fraction, return the fraction in string format.
If the fractional part is repeating, enclose the repeating part in parentheses.
(leetcode 166)

举例:

Given numerator = 1, denominator = 2, return "0.5".
Given numerator = 2, denominator = 1, return "2".
Given numerator = 2, denominator = 3, return "0.(6)".

Math+Hash：时间复杂度O(1), 空间复杂度O(1)

注意正负号，注意溢出，注意如何判断是循环小数(采用hash)

```
string num2str(int num){
    if(num == 0)    return "0";
    if(num < 0) return "-" + num2str(-num);
    string result = "";
    while(num){
        result.push_back((num % 10) + '0');
        num /= 10;
    }
    reverse(result.begin(), result.end());
    return result;
}

struct hashPair{
    size_t operator()(const pair<int, int> key) const{
        return key.first ^ key.second;
```

```
16      }
17  };
18
19  string fractionToDecimal(int numerator, int denominator) {
20      string result;
21      if(denominator == 0)    return result;
22      if(numerator == 0)  return "0";
23      bool isMunis = true;
24      if((numerator < 0 && denominator < 0) || (numerator > 0 &&
            denominator > 0)){
25          isMunis = false;
26      }
27      numerator = numerator < 0? numerator : -numerator;
28      denominator = denominator < 0? denominator : -denominator;
29      if(numerator == INT_MIN && denominator == -1){
30          result = "2147483648";
31          numerator = 0;
32      }
33      else{
34          int ret = 0;
35          while(numerator <= denominator){
36              ret = ret + numerator / denominator;
37              numerator = numerator - denominator * (numerator /
                    denominator);
38          }
39          result = num2str(ret);
40      }
41      if(numerator){
42          result.push_back('.');
43          int pos = result.size() - 1;
44          unordered_map<pair<int, int>, int, hashPair> table;
45          double numerator_d = numerator, denominator_d = denominator;
46          while(int(numerator_d)){
47              if(table.count(make_pair(int(numerator_d), int(denominator_d
                    )))){
48                  int index = table[make_pair(int(numerator_d), int(
                        denominator_d))];
49                  result.insert(index, 1, '(');
50                  result.push_back(')');
51                  break;
52              }
53              pos++;
54              table[make_pair(int(numerator_d), int(denominator_d))] = pos
                    ;
55              numerator_d *= 10;
56              int cur = numerator_d / denominator_d;
57              numerator_d = numerator_d - denominator_d * cur;
58              result.push_back('0' + cur);
59          }
60      }
61      return isMunis? "-" + result : result;
62  }
```

17.3.7 Excel Sheet Column Title

问题:

Given a positive integer, return its corresponding column title as appear in an Excel sheet. *(leetcode 168)*

举例:

1 → A
2 → B
3 → C
...
26 → Z
27 → AA
28 → AB

递归：时间复杂度O(n), 空间复杂度O(1)

这是典型的进制转换问题，可以采用递归实现一行代码解决

```
string convertToTitle(int n) {
    return n <= 0? "" : convertToTitle((n-1)/26) + string(1,((n-1) % 26)
        + 'A');
}
```

17.3.8 Excel Sheet Column Number

问题:

Related to question Excel Sheet Column Title

Given a column title as appear in an Excel sheet, return its corresponding column number. *(leetcode 171)*

举例:

A → 1
B → 2
C → 3
...
Z → 26
AA → 27
AB → 28

递归：时间复杂度O(n), 空间复杂度O(1)

进制转换，使用递归一行代码解决

```
int titleToNumber(string s) {
    return s.size()? titleToNumber(s.substr(0, s.size() - 1))*26 + s[s.
        size()-1] - 'A' + 1: 0;
}
```

17.3.9 Factorial Trailing Zeroes

问题:

Given an integer n, return the number of trailing zeroes in n!. *(leetcode 172)*

Note:

Note: Your solution should be in logarithmic time complexity.

Math：时间复杂度O($lg_5 n$), 空间复杂度O(1)

这题其实就是统计$1*2*...*n$中包含5因子的个数,因为2因子个数肯定比5的多，所以它们组合成10的倍数

```
int trailingZeroes(int n) {
    return abs(n) < 5?  0 : abs(n/5) + trailingZeroes(n/5);
}
```

17.3.10 Happy Number

问题:

Write an algorithm to determine if a number is "happy".

A happy number is a number defined by the following process: Starting with any positive integer, replace the number by the sum of the squares of its digits, and repeat the process until the number equals 1 (where it will stay), or it loops endlessly in a cycle which does not include 1. Those numbers for which this process ends in 1 are happy numbers.
(leetcode 202)

举例:

19 is a happy number

$1^2 + 9^2 = 82$
$8^2 + 2^2 = 68$
$6^2 + 8^2 = 100$
$1^2 + 0^2 + 0^2 = 1$

Hash : 时间复杂度O(1), 空间复杂度O(1)

使用hash记录是否重现

```
int change(int n){
    return n == 0? 0 : (n % 10) * (n % 10) + change(n/10);
}

bool isHappy(int n) {
    unordered_set<int> table;
    while(n != 1){
        table.insert(n);
        n = change(n);
        if(table.count(n))
            return false;
    }
    return true;
}
```

17.3.11 Count Primes

问题:

Count the number of prime numbers less than a non-negative number, n.
(leetcode 204)

筛选法 : 时间复杂度O(n), 空间复杂度O(m)

采用前面提到的筛选法即可

```
int countPrimes(int n) {
    if(n < 3)   return 0;
    vector<bool> table(n+1, true);
    int pos = 2, count = 0;
```

```
5       while(pos < n){
6           while(pos < n && !table[pos])   pos++;
7           if(pos < n) count++;
8           for(int i = 2; i * pos < n; i++)
9               table[i*pos] = false;
10          pos++;
11      }
12      return count;
13  }
```