Table of Contents

Leetcode题解(JAVA版)

在线阅读及下载

https://www.gitbook.com/book/lidinghao/leetcode-java

介绍

算法与数据结构

数据结构是为了满足算法有效访问数据访问的要求。

数学知识

卡特兰数

基本运算

关于数的各种操作技巧

得到整数的基（即十进制位数）

```
int d = 1; //与x同位数的整数,如x = 312, d =100
int k = 1; //x的位数，如x=321,k=3
while(x/d >= 10){
  d *= 10;
  k++;
}
```

线性表

链表

判断链表是否有环

分析

第一反应是遍历链表，并用HashMap将访问过的节点记录下来，如果出现重复访问，则说明有环，这样的时间复杂度是O(N),空间复杂度也是O(N)。 另外一种方式就是设置快慢两个指针，p1和p2,一开始都指向表头，p1每次移动一步，p2每次移动两步，如果p1和p2相遇，则说明有环。 代码如下：

树

图

排序

排序算法的实际实现（**Arrays.sort**源码分析）

我们将使用迭代改进的方式来描述现实中排序算法的实现，主要参考java中的Arrays.sort源码。 baseline是quicksort，然后在此基础上，进行迭代改进。

Out-of-Core算法

排序、查找、join

100 亿个数字中找出最大的 **1000** 个

这是最基本问题了，答案是用堆，但是用大根堆，还是小根堆，是有区别的。 正确答案是用小根堆，首先读取前1000个数，建立一个1000元素的小根堆，然后每次读取一个新的数，将其与堆顶的元素（即目前扫描的数中最大的1000元素中最小的，即第1000大的）比较，如果新元素比堆顶元素大，则删除堆顶元素，插入新元素，调整堆，否则什么也不做。 相反，同理，如果是找出最小的1000，则采用大根堆。

用**2G**内存对**10**亿数进行排序

这类大数据排序，或大数据统计（找出top-K， Middle-K）都要利用分治的思想，或空间搜索的思想一步步缩小搜索空间。 大数据排序，基本上是利用归并排序，先把文件分成很多块，然后对每块进行排序，然后进行归并。

Internal Sort

Quicksort Tournament Sort

利用找出40亿个非负整数中出现两次的数

利用Bitmap来压缩空间，32位无符号整数为0 ~ 4294967295, 而0-3次可以用2个bit表示（因为要求两次，所以至少要有0,1,2,3四种状态）。所以可以利用4294967295 *2* 个比特位的空间， 即 *(2^32* 2) / 8 = 1GB的内存来完成运算。

找出100亿中重复的URL

利用hash函数的特性和divide and conquer算法，对于相同输入，任何hash函数值也相同，反之不一定，（不过话说所有函数都有这个特性吧？）。可以hash函数吧URLdivide到各个比较小的文件，然后分而治之。

找出100亿个整数的中位数（或者进行其他区间统计）

利用桶排序，将数据按大小归类，放到不同的桶中，然后找出中位数所在的那个桶，然后再对通过进行进一步的桶排序，当剩余数小到一定时候，就可以选用内存排序了。 其实是一个逐步缩小搜索空间的过程。

B-tree 、B+ tree、 B-link tree

动态规划

找零钱

换钱的最少货币数

1.给一组不同面值的货币，一个金额M，求组成M的最少货币数，其中，每种面值的货币都可以使用任意张。 2.给给一组各种面值的货币（里面可能有相同面值的货币），一个金额M，求组成M的最少货币数，其中，每个货币只能使用一次。

换钱的方法数

3.给一组不同面值的货币，一个金额M，求组成M的最少货币数，其中，每种面值的货币都可以使用任意张。

0-1背包问题

找零钱和背包问题有很多共同之处。

动态规划的实际应用

join optimization

字符串匹配

KMP算法

给定两个字符串str和pattern,长度分别为N,M。实现一个算法，如果字符串str中含有子串pattern,则返回pattern在str中的开始位置，不含则返回-1。 简单KMP，时间复杂度为O(n),空间复杂度为O(M*A),A为字符的数目，英语为26。 解法:首先，对pattern构建一个自动机，用26*M的数组来表示一个自动机。构建自动机的步骤很巧妙，在构造自动机的过程中，利用目前的自动机来构造下一步自动机。

10. Regular Expression Matching

分析

首先第一印象是采用递归的方式，首先匹配str[i]与exp[j]，然后再递归匹配str[i+1...]与exp[j+1...]。如果exp[j+1]是普通字符串或"."的话，上面的递归显然是正确的。但如果exp[j+1]是"*"的话，上面的递归就不正确了。例如，str="aabc",exp="a*bc",结果应该是true,但是按上面的递归会在i=0,j=0后,匹配str[1..]和exp[1...],显然会返回错误结果。

因此我们要读"*"特殊处理，"*"的特殊性在于，它会匹配或者叫消耗0或多个str[i]字符。因此当exp[j+1] == "*", 要采用backtracking的方式，对所有情况（"*"匹配0到n-i个）进行尝试，如果有一种情况能返回true,则整个匹配就返回true。 因此，代码如下：

数组

18. 4Sum

问题描述

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.

Note:

- Elements in a quadruplet (a,b,c,d) must be in non-descending order. (ie, a ≤ b ≤ c ≤ d)
- The solution set must not contain duplicate quadruplets.

```
For example, given array S = {1 0 -1 0 -2 2}, and target = 0.

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

分析

k-sum问题都可以通过双指针左右夹逼来解决，一般步骤是，先排序，然后外层做k-2层循环，最内层循环采用双指针左右夹逼。排序时间复杂度为O(nlgn),共有k-1层循环，所以时间复杂度为O(max{nlgn,n^k-1})。

对于k >= 4的情况，采用左右夹逼容易超时。因此可以采用预计算的方式，先缓存所有两个数组合及他们的和，在最内层循环时直接取出所有和等于target减去外面两层和的组合。

由于数组经过排序，也可以采用剪枝的方式，首先在最外面层循环中去掉太大或太小的数，再把它转为3Sum,再剪枝，再转为2Sum。

代码实现

cache版

```
public class FourSum {
    public List<List<Integer>> fourSum(int[] nums, int target) {
        Arrays.sort(nums);
        Map<Integer, List<int[]>> cache = new HashMap<>();
        for (int i = 0; i < nums.length - 1; i++) {
            for (int j = i + 1; j < nums.length; j++) {
                int sum = nums[i] + nums[j];
                if (cache.containsKey(sum)) {
                    List<int[]> list = cache.get(sum);
                    list.add(new int[]{i, j});
                } else {
                    List<int[]> list = new ArrayList<>();
                    list.add(new int[]{i, j});
                    cache.put(sum, list);
                }
            }
        }
        Set<String> used = new HashSet<>();
        List<List<Integer>> result = new ArrayList<>();
        for (int i = 0; i < nums.length - 3; i++) {
            if (i > 0 && nums[i] == nums[i - 1]) continue;
            for (int j = i + 1; j < nums.length - 2; j++) {
                if (j > i + 1 && nums[j] == nums[j - 1]) continue;
                int key = target - nums[i] - nums[j];
                if (!cache.containsKey(key)) {
                    continue;
                }
                for (int[] ints : cache.get(key)) {
                    if (j >= ints[0]) continue;
                    Integer[] solution = {nums[i], nums[j], nums[ints[0]], nums[ints[1
]]};
                    if (!used.contains(Arrays.toString(solution))) {
                        used.add(Arrays.toString(solution));
                        List<Integer> list = new ArrayList<>(Arrays.asList(solution));

                        result.add(list);

                    }

                }
            }
        }
        return result;
    }
```

pruning版，效率更高

```
public List<List<Integer>> fourSum(int[] nums, int target) {
    List<List<Integer>> result = new ArrayList<>();
    if (nums.length == 0) return result;
    Arrays.sort(nums);
    int max = nums[nums.length - 1];
    int min = nums[0];
    //去除太大或太小
    if (4*min > target || 4*max < target) return result;
    for (int i = 0; i < nums.length -3; i++) {
        //去除太大
        if (4*nums[i]> target) return result;
        //去除重复
        if (i > 0 && nums[i] == nums[i-1]) continue;
        for (int j = i+1; j < nums.length - 2; j++) {
        //去除太大
         if ((nums[i] + 3*nums[j])> target) break;
           //去除重复
            if (j > i+1 && nums[j] == nums[j-1]) continue;
            int p = j + 1, q = nums.length - 1;
            while (p < q) {
                int sum = nums[i] + nums[j] + nums[p] + nums[q];
                if (sum == target) {
                    result.add(Arrays.asList(nums[i], nums[j], nums[p], nums[q]));
                    p++;
                    q--;
                    //去除重复
                    while (nums[q] == nums[q + 1] && p<q) q--;
                    while (nums[p] == nums[p-1] && p<q) p++;
                } else if (sum > target) {
                    q--;
                    //去除重复
                    while (nums[q] == nums[q + 1] && p<q) q--;
                } else {
                    p++;
                    //去除重复
                    while (nums[p] == nums[p-1] && p<q) p++;
                }
            }


        }
    }
    return result;
}
```

Remove Duplicates from Sorted Array

问题描述

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.

For example, 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.

分析

维护一个边界，边界左边是已经目前得到的不同的数，初始为num[0]。遍历数组，将当前值与上一个不同值即使边界处的值比较，不同，则说明又有一个不同的值，将插入边界处，并扩大边界。

代码实现

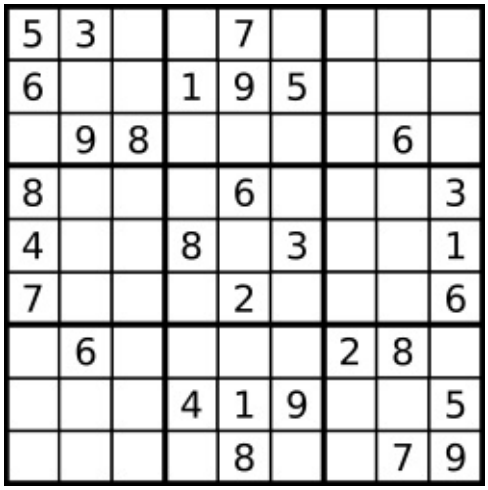
```
    public int removeDuplicates(int[] nums) {
        if (nums.length == 0) return 0;
        int id = 1;
        for (int i = 1; i < nums.length; i++) {
            if (nums[i]!=nums[id-1]) {
                nums[id++] = nums[i];
            }
        }
        return id;

    }
```

Remove Element

Valid Sudoku

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 '.'.



A partially filled sudoku which is valid.

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

分析

采用一个大小为9的数组做hash table/symbol table记录出现的元素，对于所有行，所有列以及9个3*3的subbox进行验证.

代码如下

```java
public boolean isValidSudoku(char[][] board) {
    boolean[] used = new boolean[9];
    for (int i = 0; i < 9; i++) {
        fillFalse(used);
        for (int j = 0; j < 9; j++) {
            char k = board[i][j];
            if (!check(used,k)) return false;
        }
        fillFalse(used);
        for (int j = 0; j < 9; j++) {
            char k = board[j][i];
            if (!check(used,k)) return false;
        }
    }
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (!checkSubbox(board, i, j)) return false;
        }
    }
   return  true;
}
public void fillFalse(boolean[] used) {
    for (int i = 0; i < used.length; i++) {
        used[i] = false;

    }
}

public boolean check(boolean[] used, char ch) {
    if (ch == '.') return true;
    if (used[ch - '1']) return false;
    else {
        used[ch - '1'] = true;
        return true;
    }
}
public boolean checkSubbox(char[][] board, int i, int j) {
    boolean[] used = new boolean[9];
    for (int p = i * 3; p < i * 3 + 3; p++) {
        for (int q = j * 3; q < j * 3 + 3; q++) {
            char k = board[p][q];
            if (!check(used,k)) return false;
        }
    }
    return true;
}
```

链表

Merge Two Sorted Lists

描述

Merge two sorted linked lists and return it as a new list. The new list should be made by splicing together the nodes of the first two lists.

分析

设置两个指针，分别指向两个链表，比较元素大小，交替地把较小元素连接到返回链表的尾部。直到到某个链表的尾部，然后把剩下的那个链表链接到返回链表的尾部，结束。

代码实现

```java
public class MergeTwoSortedLists {
    // Definition for singly-linked list.
    public static class ListNode {
        int val;
        ListNode next;
        ListNode(int x) {
            val = x;
        }
    }

    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        if (l1 == null || l2 == null) {
            return l1 != null ? l1 : l2;
        }
        ListNode head = new ListNode(-1);
        ListNode cur = head;
        while (l1 != null && l2 != null) {
            if (l1.val < l2.val) {
                cur.next = l1;
                cur = cur.next;
                l1 = l1.next;
            } else {
                cur.next = l2;
                cur = cur.next;
                l2 = l2.next;
            }
        }
        cur.next = l1 != null ? l1 : l2;

        return head.next;
    }
```

Merge k Sorted Lists

问题描述

Merge k sorted linked lists and return it as one sorted list. Analyze and describe its complexity.

分析

这个归并排序的第二个步骤是一样的，对k个链表进行两两归并，然后在对产生的k/2个链表再进行两两归并，直到最终归并到一个链表。时间复杂度为(nlgk),n为链表的最大长度 或者维护一个k大小的小根堆，不断地取走堆顶元素，并将该元素的next节点插入堆中，直至堆为空，即所有链表都遍历完。时间复杂度为(nlgk),n为链表的最大长度。

代码实现

递归两两归并版,比小根堆效率更高

```
    public ListNode mergeKLists(ListNode[] lists) {
        if (lists.length == 0) return null;
        if (lists.length == 1) return lists[0] ;
        ListNode[] newLists = new ListNode[(lists.length + 1) / 2];
        for (int i = 0; i < lists.length / 2; i++) {
            ListNode node = mergeTwoLists(lists[2*i], lists[2*i + 1]);
            newLists[i] = node;
        }
        if (lists.length %2 != 0) newLists[(lists.length + 1) / 2 - 1] = lists[lists.l
ength - 1];
        return mergeKLists(newLists);
    }

    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        if (l1 == null || l2 == null) {
            return l1 != null ? l1 : l2;
        }
        ListNode head = new ListNode(-1);
        ListNode cur = head;
        while (l1 != null && l2 != null) {
            if (l1.val < l2.val) {
                cur.next = l1;
                cur = cur.next;
                l1 = l1.next;
            } else {
                cur.next = l2;
                cur = cur.next;
                l2 = l2.next;
            }
        }
        cur.next = l1 != null ? l1 : l2;

        return head.next;
    }
```

最小堆版

```
public ListNode mergeKListsV2(ListNode[] lists) {
        if (lists.length == 0) {
            return null;
        }
        PriorityQueue<ListNode> minHeap = new PriorityQueue<>(lists.length,
        new Comparator<ListNode>() {
            @Override
            public int compare(ListNode o1, ListNode o2) {
                if (o1.val < o2.val) {
                    return  -1;
                } else if (o1.val == o2.val) {
                    return 0;
                } else {
                    return 1;
                }
            }
        });

        for (ListNode list : lists) {
            if (list != null) {
                minHeap.add(list);
            }
        }
        ListNode head = new ListNode(-1);
        ListNode tail = head;
        while (!minHeap.isEmpty()) {
            tail.next = minHeap.poll();
            tail = tail.next;
            if (tail.next != null)
            minHeap.add(tail.next);
        }
        return head.next;
    }
```

Swap Nodes in Pairs

分析

同反转链表等题目一样，仔细分析，设置好指针的交换顺序。这种题目一般都会有递归和迭代两种解法。

代码实现

```
public ListNode swapPairs(ListNode head) {
    ListNode dummy = new ListNode(-1);
    dummy.next = head;
    ListNode prev = dummy;
    ListNode first = dummy.next;
    while (first!= null && first.next != null) {
        prev.next = first.next;
        first.next = first.next.next;
        prev.next.next = first;

        prev=first;
        first = first.next;
    }
    return dummy.next;
}
```

Reverse Nodes in k-Group

问题描述

Given a linked list, reverse the nodes of a linked list k at a time and return its modified list.

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.

For example, Given this linked list: 1->2->3->4->5

For k = 2, you should return: 2->1->4->3->5

For k = 3, you should return: 3->2->1->4->5

分析

有递归和迭代两种解法，递归解法每次反转k个节点的子链表，返回新子链表的头部，递归是从后向前进行的，迭代是从前向后进行的。

代码实现

迭代版

```
    public ListNode reverseKGroup(ListNode head, int k) {
         if (head ==null || head.next == null || k < 2) return head;
         ListNode dummy = new ListNode(-1);
         dummy.next = head;
         ListNode prev = dummy;
         ListNode end = prev;
         while (end != null) {
             for (int i = 0; i < k && end != null; i++) {
                 end = end.next;
             }
             if (end == null) {
                 break;
             }
             prev = reverse(prev, prev.next, end);
             end = prev;
         }
         return dummy.next;
     }

     private ListNode reverse(ListNode prev, ListNode begin, ListNode end) {
         ListNode  end_next = end.next;
         for (ListNode p = begin, cur = p.next, next = cur.next;
                 cur != end_next;
                 p = cur, cur = next, next = next != null ? next.next : null) {
                   cur.next = p;

          }
         begin.next = end_next;
         prev.next = end;
         return begin;
     }
```

递归版

扩展

这种题目，考察的主要是控制结构，这样在复杂的控制流程中写出正确，清晰的代码。 最主要的是要把控制流程的代码和正常的业务逻辑代码分开。首先要通过函数把不同的逻辑分开，其次要在代码中，把复杂的控制逻辑写在for中，而不是用while把业务逻辑和控制逻辑写在一起。

Search in Rotated Sorted Array

问题描述

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).

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.

分析

仍然采用二分搜索的方法，由于至少有一半的元素是有序的，因此，每次将数组重新划分后，我们将low和mid比较，以此pivot是否落在左边区间，然后对两中情况分别处理。

1. 如果pivot不在左边区间，则左边是有序的，通过target >= A[low] && target < A[mid] 判断target的是否在左分区，如果是则hi= mid -1,如果不是，则lo = mid+1;
2. 如果pivot在左边区间，则右边是有序的，通过target > A[mid] && target <= A[hi] 判断target的是否在右分区，如果是则lo= mid + 1,如果不是，则hi = mid -1; 可见，该方法和二分搜索是类似的，只不过判断target是在左分区还是在右分区，需要根据pivot是否在左边做特殊处理。

代码实现

```
public int search(int[] nums, int target) {
     int lo = 0, hi = nums.length -1;
     while (lo <= hi) {
         int mid = (lo + hi) / 2;
         if (target == nums[mid]) return mid;
         if (nums[lo] <= nums[mid]) {
             if (target >= nums[lo] && target < nums[mid]) {
                 hi = mid - 1;
             } else {
                 lo = mid + 1;
             }
         } else {
             if (target > nums[mid] && target <= nums[hi]) {
                 lo = mid + 1;
             } else {
                 hi = mid - 1;
             }
         }
     }
   return -1;
 }
```

Search for a Range

问题描述

Given a sorted array of integers, find the starting and ending position of a given target value.

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].

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

分析

对数组做两次二分搜索，分别找出上界和下界。

代码实现

```
public int[] searchRange(int[] nums, int target) {
        int lo = 0;
        int hi = nums.length -1;
        while (lo < hi) {
            int mid = (lo + hi) / 2;
            if (nums[mid]< target)
                lo = mid + 1; //不停地向右移动lo,直到nums[mid] >= target
            else
                hi = mid; //向左移动hi,不加1是为了避免mid = left的情况。
        }
        int left;
        if(nums[lo] != target)
            return new int[] {-1,-1};
        else
            left = lo;
        hi = nums.length -1;
        while (lo < hi) {
            int mid = (lo + hi) / 2+1; // 避免([2,2],2)这种情况时出现死循环。
            if (nums[mid] > target)
                hi = mid - 1;
            else
                lo = mid;
        }
        return new int[]{left, lo};
    }
```

扩展

mid = (low + high) / 2 和 **mid = low + (high - low) / 2** 有何不同 ？ 前一种写法在low 和high 都比较大时会造成 low + high 大于Integer.MAX_VALUE,从而溢出。

关于二分搜索**low** 和 **high**指针移动的问题

1. 搜索特定值

```
while (lo < hi) {
    int mid = (lo + hi) / 2;
    if (nums[mid] == target) {
        return mid;
    } else if (nums[mid] < target) {
        lo = mid + 1;
    } else {
        hi = mid -1;
    }
}
```

Search Insert Position

问题描述

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.

Here are few examples. [1,3,5,6], 5 → 2 [1,3,5,6], 2 → 1 [1,3,5,6], 7 → 4 [1,3,5,6], 0 → 0

分析

和search for a range 的解答类似，都是基于二分搜索的变形。对于这类问题，都可以在二分搜索的基础上，对于不同的情形采用不同的判断条件和不同的移动方式，从而得到适合问题的解法。

代码实现

```
        if (target > nums[nums.length -1]) return nums.length;
         int lo= 0, hi = nums.length - 1;
         while (lo < hi) {
            int mid = (lo + hi) / 2 ;
            if (nums[mid] >= target) {
                hi = mid;
            } else {
                lo = mid + 1;
            }
         }
        return  lo ;
```

扩展

[1,3,5,6], 7 → 4 和[1,3,5,6], 0 → 0 这两个corner case决定了必须使用 >=的判断条件，并对target >nums[nums.length -1]的情况做特殊处理。 if (nums[mid] >= target) j = mid; 相当于用target值索引设立了一个边界，能够保证j不越过该边界。

栈和队列

20. Valid Parentheses

问题描述

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

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

分析

利用栈，将输入的字符依次入栈，发现对应的括号就从栈中弹出一个元素，最后栈为空，则Parentheses is valid,否者 invalid。 实现复杂度为O(n),空间复杂度也为O(n)。

代码实现

```
    public boolean isValid(String s) {
        char[] chars = s.toCharArray();
        Stack<Character> stack = new Stack<>();
        for (char ch : chars) {
            if (!stack.isEmpty() && isPair(stack.peek(), ch)) {
                stack.pop();
            } else {
                stack.push(ch);
            }
        }
        return stack.isEmpty();

    }

    public boolean isPair(char left, char right) {
        switch (left) {
            case '(':
                return right == ')';
            case '{':
                return right == '}';
            case '[':
                return right == ']';
            default:
                return false;
        }
    }
```

Longest Valid Parentheses

问题描述

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.

分析

遍历字符串，并用栈记录当前索引，对于每个字符：

1. 如果当前字符为'(',则将索引入栈。
2. 如果当前字符为')', 栈顶对应的字符为'('，则将栈顶元素弹出，并用当前索引减去栈顶索引，得到一个新的长度curLen，将curLen与原先保存的最大长度比较，更新最大长度。
3. 如果当前字符为')',栈为空，或栈顶对应的字符为')',则将索引压栈。

代码实现

```
public int longestValidParentheses(String s) {
    if (s.length()==0) {
        return 0;
    }
    char[] chars = s.toCharArray();
    Stack<Integer> stack = new Stack<>();
    int maxLen = 0;
    for (int i = 0; i < chars.length; i++) {
        if (chars[i] == '(') {
            stack.push(i);
        } else if (!stack.isEmpty() && chars[stack.peek()] == '(') {
            stack.pop();
            int curLen = 0;
            if (!stack.isEmpty()) {
                curLen = i - stack.peek();
            } else {
                curLen = i + 1;
            }
            maxLen = Math.max(maxLen, curLen);
        } else {
            stack.push(i);
        }
    }
     return maxLen;
}
```

位运算

Java位操作运算符

符号右移(">>")

负数在最高位填充"1"，正数在最高位填充"0", 右移一位相当于除2,ArrayList中每次扩容1.5倍，所使用的代码就是：

```
newCapacity = oldCapacity + (oldCapacity >> 1);
```

左移("<<")

在最低位填充"0", 左移一位相当于乘2。

无符号右移(">>>")

在最高位填充"0"。

位与("&")

按位与运算符，常见应用是用来做bitmask运算：

```
    int bitmask = 0x000F;
    int val = 0x2222;
    val & bitmask == 2
```

位异或("^")

"bitwise exclusive OR"，两个操作数的第n位相同则结果第n位为0，不同即相异则结果第n位为1。

位或("|")

"bitwise inclusive OR"，两个操作数的第n位有一个为1，则结果第n位为0，如果都为0则结果第n位为1。

注意，如果用位运算来进行逻辑运算的话，会导致两边的逻辑表达式都会被运算，如下面所示：

```
int array[] = new int[]{1,2,3};
int i = 0;
//这里不会发生数组越界
while(i < array.length && array[i] != 4){
    i++;
}

i = 0;
//这里会发生数组越界
while(i < array.length & array[i] != 4){
    i++;
}
```

因此，逻辑运算符也叫做"short-circuit" 布尔运算符， 位运算符叫做"non-short-circuiting"运算符。 因为当左边表达式就可以决定整个表达式的值时，右边的就不会被计算。而为位运算符总是计算左右两个表达式。

经典算法

对于整型变量a、b，如何不经中间变量，来交换a与b的值？ a = a^b; b = b ^ a; a = b ^ a; 或者 a = a + b; b = a - b; a = a - b;

Divide Two Integers

问题描述

Divide two integers without using multiplication, division and mod operator.

If it is overflow, return MAX_INT.

分析

首先通过计算结果符号，把问题reduce到两个整数相除。然后把两个数字转为long型，以免溢出。 最后把被除数减去除数，并通过将被除数每次加倍来加快速度。当被除数大于剩余的除数时，再从初始的除数开始，重复以上步骤。

代码实现

```
public int divide(int dividend, int divisor) {
        int sign = (( dividend<0) ^ (divisor <0))  ? -1 : 1;
        long dvd = Math.abs((long)dividend);
        long dvs = Math.abs((long)divisor);
        long result = 0;
        while (dvd > dvs) {
            long temp = dvs;
            for (int i = 0; dvd >= temp; i++, temp <<= 1) {
                dvd -= temp;
                result += 1<<i;
            }
        }
        result = sign*result > Integer.MAX_VALUE ? Integer.MAX_VALUE : sign*result;
        return (int) result ;
    }
```

扩展

判断两个数a,b相除或相乘结果的符号？ 最直接写法 if((a>0 && b<0) ||="" a<0="" &&="" b="">0) sign = -1; else sign = 1; **通过异或操作来写 sign = (a < 0) ^ (b < 0) ? -1 : 1;

位移加异或 sign = (a^b)>>>31 == 0 ? 1 : -1;

通过布尔逻辑 sing = a < 0 != b < 0 ? -1 : 1;

字符串

Implement strStr()

问题描述

Implement strStr().

Returns the index of the first occurrence of needle in haystack, or -1 if needle is not part of haystack.

分析

经典的字符串搜索算法，有暴力搜索，KMP，、Boyer-Mooer和Rabin-Karp等。

代码实现

暴力搜索

Substring with Concatenation of All Words

问题描述

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.

For example, given: s: "barfoothefoobarman" words: ["foo", "bar"]

You should return the indices: [0,9]. (order does not matter).

分析

设置一个words.length * wordLen长度的滑动窗口，将窗口从左向右滑动，并将窗口再划分为长度为wordlen的块，依次检查每个块是否包含在words中，并且出现次数也是符合要求的。

代码实现

```
public List<Integer> findSubstring(String s, String[] words) {
    List<Integer> result = new LinkedList<>();
    if (s.length() == 0 || words.length == 0) return result;
    Map<String, Integer> wordCount = new HashMap<>();
    for (String word : words) {
        wordCount.put(word, wordCount.getOrDefault(word, 0) + 1);
    }
    int wordLen = words[0].length();
    int winSize = words.length * wordLen;
    for (int i = 0; i <= s.length() - winSize; i++) {
        Map<String, Integer> unused = new HashMap<>(wordCount);
        for (int j = 0; j < winSize; j+=wordLen) {
            String str = s.substring(i + j, i + j + wordLen);
            int count = unused.getOrDefault(str, -1);
            if (count == 1) {
                unused.remove(str);
            } else if (count > 1) {
                unused.put(str, count - 1);
            } else {
                break;
            }


        }
        if (unused.size() == 0) result.add(i);
    }
    return result;
}
```

回溯

概念

回溯是一种类似枚举的搜索尝试方法，在搜索解的过程中，如果发现答案不正确或已到达边界，就退回到上一步重新选择。这种走不通就退回再走的技术称为回溯法，而满足回溯条件的某个状态的被称为“回溯点”。

基本思想

对于能用回溯解决的问题，它的所有解构成了一个解空间树。回溯法采用深度优先搜索(DFS)的策略，从根节点出发，评估每个节点是否属于问题的解，如果是，就继续向下搜索，如果不是，则退回到父节点重新选择搜索路径。

若用回溯法求问题的所有解时，要回溯到根，且根结点的所有可行的子树都要已被搜索遍才结束。

而若使用回溯法求任一个解时，只要搜索到问题的一个解就可以结束。

一般步骤

1. 分析所给问题，确定问题的解空间
2. 制定节点的扩展搜索规则
3. 采用DFS的方式搜索解空间，并注意使用剪枝函数避免无效搜索。

算法框架

回溯与递归的区别

递归是指函数自己调用自己的一种形式。

回溯是一种系统性地搜索问题解答的方法。

回溯和DFS,BFS, 动态规划一样是算法，可以用递归方式实现，也可以不用递归方式实现，而递归和循环，迭代一样是一种方法。

17. 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.


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

分析

可以采用两种方式来分析思考本题，一种是回溯，一种是组合。 采用回溯时，首先构建搜索树，可以认为digits中的第一个元素对于的letters为根节点的子节点，第二个元素对应的letters为第一层节点的子节点，以此类推。 然后我们制定规则来DFS搜索这棵树，并注意保存状态，即从根节点到当前节点的路径，当满足条件时（到达叶子节点时），再回溯到上一个节点，重新选择路径来搜索。 采用组合时，可以采用incremental的方式来构建组合，即首先构建一个空集，然后构建包含digitals[0]对于letters的组合，然后在构建包含digitals[0]，digitals[1]对于letters的组合。同78.Subsets的解法类似。

代码实现

回溯

```
    public Map<Integer, List<Character>> buildMap() {
        String alphabet = "abcdefghijklmnopqrstuvwxyz";
        Map<Integer, List<Character>> map = new HashMap<>();
        for (int i = 2,j = 0; i <= 9&&j< 26; i++) {
            List<Character> list = new ArrayList<>();
            list.add(alphabet.charAt(j++));
            list.add(alphabet.charAt(j++));
            list.add(alphabet.charAt(j++));
            if (i == 7 || i == 9) {
                list.add(alphabet.charAt(j++));
            }
            map.put(i, list);
        }
        return map;
    }
    public List<String> letterCombinations(String digits) {
        List<String> result = new ArrayList<>();
        Map<Integer, List<Character>> map = buildMap();
        if (digits.length()==0) {
            result.add("");
        }
        combine(digits,0,"",result, map);
        return result;
    }

    public void combine(String digits, int index, String prefix, List<String> result,
Map<Integer, List<Character>> map){
        if (index ==digits.length()){
            result.add(prefix);
            return;
        };
        List<Character> chars = map.get(Character.getNumericValue(digits.charAt(index)
));
        for (Character ch : chars) {
            combine(digits, index+1, prefix+ch, result, map);
        }
    }
```

组合

```
    public List<String> letterCombinationsV2(String digits) {
        String digitletter[] = {"","","abc","def","ghi","jkl","mno","pqrs","tuv","wxyz"
};
        List<String> result = new ArrayList<>();
        if (digits.length() == 0) return result;
        result.add("");
        for (int i = 0; i < digits.length(); i++) {
            char[] chars = digitletter[digits.charAt(i) - '0'].toCharArray();
            List<String> temp = new ArrayList<>();
            for (String s : result) {
                for (char ch : chars) {
                    temp.add(s + ch);
                }
            }
            result = temp;
        }
        return result;
    }
```

Generate Parentheses

问题描述

Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

For example, given n = 3, a solution set is:

"((()))", "(()())", "(())()", "()(())", "()()()"

分析

采用递归的方式构造括号串，当左括号数量 < n 时，添加左括号，直到=n,然后添加右括号，并不断回溯递归，直到右括号数量=n。

代码实现

```
    public List<String> generateParenthesis(int n) {
        List<String> result = new ArrayList<>();
        backtrack(result,"", 0, 0, n);
        return result;
    }

    private void backtrack(List<String> result, String str, int open, int close, int n)
{
        if (str.length() == 2 * n) {
            result.add(str);
            return;
        }
        if (open < n) {
            backtrack(result, str + "(", open + 1, close, n);
        }
        if (close < open) {
            backtrack(result, str + ")", open, close + 1, n);
        }

    }
```

Sudoku Solver

问题描述

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.

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

...and its solution numbers marked in red.

分析

应该是采用回回溯的方式来对每个空白的地方进行不断的尝试。注意，在尝试之前可以进行一些预处理，去掉那些本行本列中以及存在的数，只尝试那些没出现的数。

代码实现

```
public void solveSudoku(char[][] board) {
    if (board == null || board.length == 0)
        return;
    solve(board);

}

public static boolean solve(char[][] board) {
    for (int i = 0; i < 9; i++) {
        for (int j = 0; j < 9; j++) {
            if (board[i][j] == '.') {
                for (char k = '1'; k <= '9'; k++) {
                    board[i][j] = k;
                    if (isValid(board, i, j) && solve(board))
                        return true;
                    else
                        board[i][j] = '.';
                }
                return false;
            }

        }
    }
     //当最终求得正确解时,会走到这一段代码,因为所有的空格都被填满,所以if语句不会被执行。
    return true;

}

public static boolean isValid(char[][] board, int i, int j) {
    for (int col = 0; col < 9; col++) {
        if (col != j && board[i][col] == board[i][j])
            return false;
    }
    for (int row = 0; row < 9; row++) {
        if (row != i && board[row][j] == board[i][j])
            return false;
    }
    for (int row = (i / 3) * 3; row < (i / 3) * 3 + 3; row++) {
        for (int col = (j / 3) * 3; col < (j / 3) * 3 + 3; col++) {
            if ((row != i || col != j) && board[row][col] == board[i][j])
                return false;
        }
    }
    return true;
}
```

复杂度分析

Many recursive searches can be modeled as a tree. In Sodoku, you have 9 possibilities each time you try out a new field. At maximum, you have to put solutions into all 81 fields. At this point it can help drawing it up in order to see, that the resulting search space is a tree with a depth of 81 and a branching factor of 9 at each node of each layer, and each leaf is a possible solution. Given these numbers, the search space is 9^81.

Since you don't check if the solution is correct after trying 81 times but after each try, the actual search space is way smaller. I don't know how to put it into numbers really, maybe the branching factor gets smaller by 1 each time you set n tries, as a rough estimate. But given any Sodoku with k pre-set numbers, you can with 100% certainty say, that you will need at most $n^{(n^2-k)}$ tries.

细节实现

分支限界

类似于回溯法，也是一种在问题的解空间树T上搜索问题解的算法。但在一般情况下，分支限界法与回溯法的求解目标不同。回溯法的求解目标是找出T中满足约束条件的所有解，而分支限界法的求解目标则是找出满足约束条件的一个解，或是在满足约束条件的解中找出使某一目标函数值达到极大或极小的解，即在某种意义下的最优解。
http://www.cnblogs.com/steven_oyj/archive/2010/05/22/1741378.html

贪心

选择子问题最优 从子问题最优到全局最优

数学

数学面试题

面试中经常出现概率与组合问题

概率面试题

1、有一苹果，两个人抛硬币来决定谁吃这个苹果，先抛到正面者吃。问先抛者吃到苹果的概率是多少？ 先抛者必然假设概率为P,则 2、一副扑克牌54张，现分成3等份每份18张，问大小王出现在同一份中的概率是多少？ 思路一,采用排列组合的方式： 54份牌分为3份，共有M=(C54取18)*(C36取18)*(C18取18)种分法。 其中大小王在同一份的分法有N=(C3取1)*(C52取16)*(C36取18)*(C18取18)种，即首先选大小王出现哪份，共有(C3取1)中，然后大小王出现的那份中，剩下的16张牌有(C52取16)种，剩下的两份依次有(C36取18)，(C18取18)种。因此所求概率为P=N /M=17/53。 http://blog.csdn.net/rudyalwayhere/article/details/7349957

Next Permutation

分析

主要是分析并发现数字中的规律。

0. Initial sequence

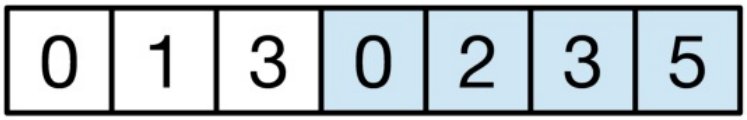
1. Find longest non-increasing suffix

2. Identify pivot

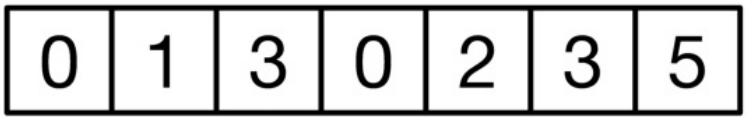
3. Find rightmost successor to pivot in the suffix

4. Swap with pivot

5. Reverse the suffix



6. Done



代码实现

```
public void nextPermutation(int[] nums) {
    if (nums==null || nums.length <= 1) return;
    int i = nums.length -2;
    while (i >= 0 && nums[i] >= nums[i+1]) i--;
    if (i >= 0) {
    int j = nums.length -1;
        while (nums[j] <= nums[i]) j--;
        swap(nums, i, j);

    }
    reverse(nums, i+1, nums.length -1);
}

private void reverse(int[] nums, int i, int j) {
    while (i < j) {
        swap(nums, i,j);
        i++;
        j--;
    }
}

private void swap(int[] nums, int i, int j) {
    nums[i] ^= nums[j];
    nums[j] ^= nums[i];
    nums[i] ^= nums[j];
}
```

大数据算法

Out-of-Core算法

排序、查找、join

用2G内存对10亿数进行排序

这类大数据排序，或大数据统计（找出top-K， Middle-K）都要利用分治的思想，或空间搜索的思想一步步缩小搜索空间。 大数据排序，基本上是利用归并排序，先把文件分成很多块，然后对每块进行排序，然后进行归并。

Internal Sort

Quicksort Tournament Sort

利用找出40亿个非负整数中出现两次的数

利用Bitmap来压缩空间，32位无符号整数为0 ~ 4294967295, 而0-3次可以用2个bit表示（因为要求两次，所以至少要有0,1,2,3四种状态）。所以可以利用4294967295 *2* 个比特位的空间， 即 *(2^32* 2) / 8 = 1GB的内存来完成运算。

找出100亿中重复的URL

利用hash函数的特性和divide and conquer算法，对于相同输入，任何hash函数值也相同，反之不一定，（不过话说所有函数都有这个特性吧？）。可以hash函数吧URL divide到各个比较小的文件，然后分而治之。

找出100亿个整数的中位数（或者进行其他区间统计）

利用桶排序，将数据按大小归类，放到不同的桶中，然后找出中位数所在的那个桶，然后再对通过进行进一步的桶排序，当剩余数小到一定时候，就可以选用内存排序了。 其实是一个逐步缩小搜索空间的过程。

topK问题

在N个数字中找出最大或最小的K个数字，N通常以亿为单位。 这是最常见的问题了，答案是用堆，但是用大根堆，还是小根堆，是有区别的。选出最小的k个元素，要用大根堆，选出最大的k个元素，要用小根堆。 时间复杂度为O(nlgk)，空间复杂度为O(1)

100 亿个数字中找出最大的 1000 个

答案是用小根堆，首先读取前1000个数，建立一个1000元素的小根堆，然后每次读取一个新的数，将其与堆顶的元素（即目前扫描的数中最大的1000元素中最小的，即第1000大的）比较，如果新元素比堆顶元素大，则删除堆顶元素，插入新元素，调整堆，否则什么也不做。相反，同理，如果是找出最小的1000，则采用大根堆。

外排序

用2G内存对10亿数进行排序

这类大数据排序，或大数据统计（找出top-K， Middle-K）都要利用分治的思想，或空间搜索的思想一步步缩小搜索空间。 大数据排序，基本上是利用归并排序，先把文件分成很多块，然后对每块进行排序，然后进行归并。