算法解题笔记

郑华

2019 年 5 月 23 日

目录

第一章 算法基础 **27**

1.1 时间复杂度 . . . 27

1.2 解题提醒 . . . 27

1.2.1 等价意思.. 简化题目 . . . 27

1.2.2 有无最优子结构 . . . 27

1.2.3 想通可行再实践 . . . 27

1.3 分治思想 . . . 27

1.3.1 基本概念 . . . 27

1.3.2 基本思想及策略 . . . 28

1.3.3 分治法适用的情况 . . . 28

1.3.4 基本步骤 . . . 28

1.3.5 可使用分治法求解的一些经典问题 . . . 29

1.3.6 依据分治法设计程序时的思维过程 . . . 29

1.4 动态规划思想 . . . 29

1.4.1 基本概念 . . . 29

1.4.2 基本思想和策略 . . . 29

1.4.3 适用的情况 . . . 30

1.4.4 例题说明 . . . 30

1.4.5 题目 . . . 31

1.5 贪心算法思想 . . . 31

1.5.1 基本概念 . . . 31

1.5.2 基本思想和策略 . . . 32

1.5.3 适用的情况 . . . 32

1.5.4 求解的基本步骤 . . . 32

1.5.5 算法实现的说明 . . . 32

1.5.6 与动态规划的区别 . . . 32

1.6 参考 . . . 32

第二章 剑指 **33**

2.1 基础知识 . . . 33

2.1.1 C++ . . . 33

2.1.2 数据结构 . . . 34
2.1.3 解题分析 . . . 35
2.2 高质量代码 . . . 35
2.2.1 代码的质量 . . . 36
2.2.2 代码的规范性 . . . 36
2.2.3 代码的完整性 . . . 36
2.2.4 代码的鲁棒性 . . . 37
2.2.5 小结 . . . 37
2.3 解题思路 . . . 38
2.3.1 画图让抽象形象化 . . . 38
2.3.2 举例让抽象具体化 . . . 38
2.3.3 分解让复杂问题简化 . . . 38
2.4 优化时间和空间效率 . . . 38
2.4.1 时间效率 . . . 38
2.4.2 时间效率与空间效率的平衡 . . . 40
2.5 能力体现篇 . . . 41
2.5.1 知识迁移能力 . . . 41
2.5.2 抽象建模能力 . . . 43
2.5.3 思维发散能力 . . . 43
2.6 英文新增篇 . . . 43
2.6.1 数组 . . . 43
2.6.2 字符串 . . . 44
2.6.3 链表 . . . 44
2.6.4 树 . . . 44
2.6.5 回溯法 . . . 45

第三章 编程之美 47

3.1 数字中的技巧 . . . 47
3.1.1 求二进制数中 1 的个数 . . . 47
3.1.2 不要被阶乘吓到 . . . 47
3.1.3 寻找超过占比 1/2 的 ID . . . 47
3.1.4 1 的数目 . . . 48
3.1.5 寻找最大的 k 个数 . . . 49
3.1.6 精确表达浮点数 . . . 49
3.1.7 最大公约数问题 . . . 49
3.1.8 找到符合条件的整数 . . . 49
3.1.9 斐波那契数列 . . . 49
3.1.10 寻找数组中的最大最小值 . . . 49
3.1.11 寻找最近点对 . . . 49

3.1.12 找到符合条件的两个数 . . . 49
3.1.13 子数组的最大乘积 . . . 49
3.1.14 子数组之和的最大值 . . . 49
3.1.15 子数组之和的最大值 (二维) . . . 49
3.1.16 求数组中最长递增子序列 . . . 49
3.1.17 数组循环移位 . . . 50
3.1.18 数组分割 . . . 50
3.1.19 区间重合判断 . . . 50
3.1.20 程序理解和时间分析 . . . 50
3.1.21 只考加法的面试题 . . . 50
3.2 字符串及链表的探索 . . . 50
3.2.1 字符串移位包含的问题 . . . 50
3.2.2 电话号码对应英语单词 . . . 50
3.2.3 计算字符串的相似度 . . . 50
3.2.4 从无头单链表中删除节点 . . . 50
3.2.5 最短摘要的生成 . . . 50
3.2.6 判断两个链表是否交叉 . . . 50
3.2.7 队列中取最大值操作问题 . . . 50
3.2.8 求二叉树中结点的最大距离 . . . 50
3.2.9 重建二叉树 . . . 50
3.2.10 分层遍历二叉树 . . . 53
3.2.11 二分查找 . . . 53
3.3 数学游戏 . . . 54
3.3.1 金刚坐飞机问题 . . . 54
3.3.2 瓷砖覆盖地板 . . . 54
3.3.3 买票找零 . . . 54
3.3.4 点是否在三角形内 . . . 54
3.3.5 磁带文件存放优化 . . . 54
3.3.6 桶中取黑白球 . . . 54
3.3.7 蚂蚁爬杆 . . . 54
3.3.8 三角形测试用例 . . . 54
3.3.9 数独知多少 . . . 54
3.3.10 数字哑谜和回文 . . . 54
3.3.11 扫雷游戏的概率 . . . 54

第四章 Array- 80 **55**

4.1 Two Sum . . . 55
4.1.1 问题 . . . 55
4.1.2 解答 . . . 55

4.1.3 Normal-两层循环 . . . 55
4.1.4 Good-HashTable:HashMap . . . 55
4.2 Median of Two Sorted Arrays . . . 56
4.2.1 分析 . . . 57
4.2.2 解答 . . . 57
4.3 Word Ladder II . . . 57
4.3.1 分析 . . . 57
4.3.2 解答 . . . 57
4.4 4Sum . . . 57
4.4.1 分析 . . . 58
4.4.2 解答 . . . 58
4.5 Rotate Array . . . 58
4.5.1 分析 . . . 58
4.5.2 解答 . . . 58
4.6 Maximum Product Subarray . . . 58
4.6.1 分析 . . . 58
4.6.2 解答 . . . 58
4.7 Spiral Matrix . . . 58
4.7.1 分析 . . . 59
4.7.2 解答 . . . 59
4.8 First Missing Positive . . . 59
4.8.1 分析 . . . 59
4.8.2 解答 . . . 59
4.9 Word Search . . . 59
4.9.1 分析 . . . 60
4.9.2 解答 . . . 60
4.10 Largest Rectangle in Histogram . . . 60
4.10.1 分析 . . . 60
4.10.2 解答 . . . 60
4.11 Jump Game II . . . 60
4.11.1 分析 . . . 61
4.11.2 解答 . . . 61
4.12 Maximal Rectangle . . . 61
4.12.1 分析 . . . 61
4.12.2 解答 . . . 61

第五章 Hash Table- 52 **63**

5.1 Hash . . . 63
5.1.1 hash 函数的设计 . . . 63

5.1.2 hash 表大小的确定 . . . 64
5.1.3 冲突的解决 . . . 64
5.2 Longest Substring Without Repeating Characters . . . 64
5.3 . . . 65
5.3.1 分析 . . . 65
5.3.2 解答 . . . 65
5.4 . . . 65
5.4.1 分析 . . . 66
5.4.2 解答 . . . 66
5.5 . . . 66
5.5.1 分析 . . . 66
5.5.2 解答 . . . 66
5.6 . . . 66
5.6.1 分析 . . . 66
5.6.2 解答 . . . 66
5.7 . . . 66
5.7.1 分析 . . . 66
5.7.2 解答 . . . 66
5.8 . . . 66
5.8.1 分析 . . . 66
5.8.2 解答 . . . 66
5.9 . . . 66
5.9.1 分析 . . . 67
5.9.2 解答 . . . 67
5.10 . . . 67
5.10.1 分析 . . . 67
5.10.2 解答 . . . 67

第六章 Linked List- 27 **69**
6.1 将单链表反转 . . . 69
6.1.1 解答 . . . 69
6.2 查找单链表中的倒数第 K 个结点（k > 0） . . . 69
6.2.1 解答 . . . 69
6.3 查找单链表的中间结点 . . . 69
6.3.1 解答 . . . 69
6.4 从尾到头打印单链表 . . . 69
6.4.1 分析 . . . 69
6.4.2 解答 . . . 69
6.5 已知两个单链表 pHead1 和 pHead2 各自有序，把它们合并成一个链表依然有序 . . 69

6.5.1 分析 . . . 70
6.5.2 解答 . . . 70
6.6 判断一个单链表中是否有环 . . . 70
6.6.1 分析 . . . 70
6.6.2 解答 . . . 70
6.7 判断两个单链表是否相交 . . . 70
6.7.1 分析 . . . 70
6.7.2 解答 . . . 70
6.8 求两个单链表相交的第一个节点 . . . 70
6.8.1 分析 . . . 70
6.8.2 解答 . . . 70
6.9 已知一个单链表中存在环，求进入环中的第一个节点 . . . 70
6.9.1 分析 . . . 71
6.9.2 解答 . . . 71
6.10 给出一单链表头指针 pHead 和一节点指针 pToBeDeleted，O(1) 时间复杂度删除节点 pToBeDeleted . . . 71
6.10.1 分析 . . . 71
6.10.2 解答 . . . 71

第七章 Math- 58 **73**

7.1 . . . 73
7.1.1 分析 . . . 73
7.1.2 解答 . . . 73
7.2 . . . 73
7.2.1 分析 . . . 73
7.2.2 解答 . . . 73
7.3 . . . 73
7.3.1 分析 . . . 73
7.3.2 解答 . . . 73
7.4 . . . 73
7.4.1 分析 . . . 74
7.4.2 解答 . . . 74
7.5 . . . 74
7.5.1 分析 . . . 74
7.5.2 解答 . . . 74
7.6 . . . 74
7.6.1 分析 . . . 74
7.6.2 解答 . . . 74
7.7 . . . 74

7.7.1 分析 . . . 74
7.7.2 解答 . . . 74
7.8 . . . 74
7.8.1 分析 . . . 74
7.8.2 解答 . . . 74
7.9 . . . 74
7.9.1 分析 . . . 75
7.9.2 解答 . . . 75
7.10 . . . 75
7.10.1 分析 . . . 75
7.10.2 解答 . . . 75

第八章 Two Pointers- 33 **77**
8.1 . . . 77
8.1.1 分析 . . . 77
8.1.2 解答 . . . 77
8.2 . . . 77
8.2.1 分析 . . . 77
8.2.2 解答 . . . 77
8.3 . . . 77
8.3.1 分析 . . . 77
8.3.2 解答 . . . 77
8.4 . . . 77
8.4.1 分析 . . . 78
8.4.2 解答 . . . 78
8.5 . . . 78
8.5.1 分析 . . . 78
8.5.2 解答 . . . 78
8.6 . . . 78
8.6.1 分析 . . . 78
8.6.2 解答 . . . 78
8.7 . . . 78
8.7.1 分析 . . . 78
8.7.2 解答 . . . 78
8.8 . . . 78
8.8.1 分析 . . . 78
8.8.2 解答 . . . 78
8.9 . . . 78
8.9.1 分析 . . . 79

8.9.2 解答 . . . 79
8.10 . . . 79
8.10.1 分析 . . . 79
8.10.2 解答 . . . 79

第九章 String- 55 **81**
9.1 . . . 81
9.1.1 分析 . . . 81
9.1.2 解答 . . . 81
9.2 . . . 81
9.2.1 分析 . . . 81
9.2.2 解答 . . . 81
9.3 . . . 81
9.3.1 分析 . . . 81
9.3.2 解答 . . . 81
9.4 . . . 81
9.4.1 分析 . . . 82
9.4.2 解答 . . . 82
9.5 . . . 82
9.5.1 分析 . . . 82
9.5.2 解答 . . . 82
9.6 . . . 82
9.6.1 分析 . . . 82
9.6.2 解答 . . . 82
9.7 . . . 82
9.7.1 分析 . . . 82
9.7.2 解答 . . . 82
9.8 . . . 82
9.8.1 分析 . . . 82
9.8.2 解答 . . . 82
9.9 . . . 82
9.9.1 分析 . . . 83
9.9.2 解答 . . . 83
9.10 . . . 83
9.10.1 分析 . . . 83
9.10.2 解答 . . . 83
9.11 . . . 83
9.11.1 分析 . . . 83
9.11.2 解答 . . . 83

9.12 83
9.12.1 分析 83
9.12.2 解答 83
9.13 83
9.13.1 分析 83
9.13.2 解答 83
9.14 83
9.14.1 分析 84
9.14.2 解答 84
9.15 84
9.15.1 分析 84
9.15.2 解答 84
9.16 Divide and Conquer- 12 84
9.17 84
9.17.1 分析 84
9.17.2 解答 84
9.18 84
9.18.1 分析 84
9.18.2 解答 84
9.19 84
9.19.1 分析 84
9.19.2 解答 84
9.20 84
9.20.1 分析 85
9.20.2 解答 85
9.21 85
9.21.1 分析 85
9.21.2 解答 85
9.22 85
9.22.1 分析 85
9.22.2 解答 85
9.23 85
9.23.1 分析 85
9.23.2 解答 85
9.24 85
9.24.1 分析 85
9.24.2 解答 85
9.25 85
9.25.1 分析 86

9.25.2 解答 . . . 86
9.26 . . . 86
9.26.1 分析 . . . 86
9.26.2 解答 . . . 86

第十章 Binary Search- 38 **87**
10.1 . . . 87
10.1.1 分析 . . . 87
10.1.2 解答 . . . 87
10.2 . . . 87
10.2.1 分析 . . . 87
10.2.2 解答 . . . 87
10.3 . . . 87
10.3.1 分析 . . . 87
10.3.2 解答 . . . 87
10.4 . . . 87
10.4.1 分析 . . . 88
10.4.2 解答 . . . 88
10.5 . . . 88
10.5.1 分析 . . . 88
10.5.2 解答 . . . 88
10.6 . . . 88
10.6.1 分析 . . . 88
10.6.2 解答 . . . 88
10.7 . . . 88
10.7.1 分析 . . . 88
10.7.2 解答 . . . 88
10.8 . . . 88
10.8.1 分析 . . . 88
10.8.2 解答 . . . 88
10.9 . . . 88
10.9.1 分析 . . . 89
10.9.2 解答 . . . 89
10.10 . . . 89
10.10.1 分析 . . . 89
10.10.2 解答 . . . 89

第十一章 Dynamic Programming- 66 **91**
11.1 Range Sum Query - Immutable . . . 91
11.1.1 分析 . . . 91

11.1.2 解答 . . . 91
11.2 Maximum Subarray . . . 92
11.2.1 分析 . . . 92
11.2.2 解答 . . . 92
11.3 Triangle . . . 93
11.3.1 分析 . . . 94
11.3.2 解答 . . . 94
11.4 2 Keys Keyboard . . . 94
11.4.1 分析 . . . 95
11.4.2 解答 . . . 95
11.5 Palindromic Substrings . . . 96
11.5.1 分析 . . . 96
11.5.2 解答 . . . 97
11.6 Longest Increasing Subsequence . . . 97
11.6.1 分析 . . . 97
11.6.2 解答 . . . 98
11.7 0-1 背包 . . . 98
11.7.1 分析 . . . 98
11.7.2 解答 . . . 99
11.8 完全背包 . . . 100
11.8.1 分析 . . . 100
11.8.2 解答 . . . 100
11.9 石子合并 . . . 100
11.9.1 分析 . . . 100
11.9.2 实现 . . . 101
11.10 Maximum Length of Pair Chain . . . 101
11.10.1 分析 . . . 102
11.10.2 解答 . . . 102
11.11 合唱团-2016 网易编程 . . . 103
11.11.1 分析 . . . 103
11.11.2 解答 . . . 103
11.12 Palindrome Partitioning . . . 105
11.13 Palindrome Partitioning II . . . 105
11.13.1 分析 . . . 105
11.13.2 解答 . . . 105
11.14 Maximal Rectangle . . . 105
11.14.1 分析 . . . 105
11.14.2 解答 . . . 105
11.15 Interleaving String . . . 105

11.15.1 分析 . . . 106
11.15.2 解答 . . . 106
11.16 Scramble String(混杂字符串) . . . 106
11.16.1 分析 . . . 106
11.16.2 解答 . . . 106
11.17 Minimum Path Sum . . . 106
11.17.1 分析 . . . 106
11.17.2 解答 . . . 106
11.18 Edit Distance . . . 106
11.18.1 分析 . . . 107
11.18.2 解答 . . . 107
11.19 Decode Ways . . . 107
11.19.1 分析 . . . 107
11.19.2 解答 . . . 107
11.20 Distinct Subsequences . . . 107
11.21 Word Break . . . 107

第十二章 Backtracking- 34 **109**

12.1 . . . 109
12.1.1 分析 . . . 109
12.1.2 解答 . . . 109
12.2 . . . 109
12.2.1 分析 . . . 109
12.2.2 解答 . . . 109
12.3 . . . 109
12.3.1 分析 . . . 109
12.3.2 解答 . . . 109
12.4 . . . 109
12.4.1 分析 . . . 110
12.4.2 解答 . . . 110
12.5 . . . 110
12.5.1 分析 . . . 110
12.5.2 解答 . . . 110
12.6 . . . 110
12.6.1 分析 . . . 110
12.6.2 解答 . . . 110
12.7 . . . 110
12.7.1 分析 . . . 110
12.7.2 解答 . . . 110

第十三章 Stack- 25 **111**
13.1 . . . 111
13.1.1 分析 . . . 111
13.1.2 解答 . . . 111
13.2 . . . 111
13.2.1 分析 . . . 111
13.2.2 解答 . . . 111
13.3 . . . 111
13.3.1 分析 . . . 111
13.3.2 解答 . . . 111
13.4 . . . 111
13.4.1 分析 . . . 112
13.4.2 解答 . . . 112
13.5 . . . 112
13.5.1 分析 . . . 112
13.5.2 解答 . . . 112

第十四章 Heap- 15 **113**
14.1 . . . 113
14.1.1 分析 . . . 113
14.1.2 解答 . . . 113
14.2 . . . 113
14.2.1 分析 . . . 113
14.2.2 解答 . . . 113
14.3 . . . 113
14.3.1 分析 . . . 113
14.3.2 解答 . . . 113
14.4 . . . 113
14.4.1 分析 . . . 114
14.4.2 解答 . . . 114
14.5 . . . 114
14.5.1 分析 . . . 114
14.5.2 解答 . . . 114
14.6 . . . 114
14.6.1 分析 . . . 114
14.6.2 解答 . . . 114
14.7 . . . 114
14.7.1 分析 . . . 114
14.7.2 解答 . . . 114

14.8 . . . 114
14.8.1 分析 . . . 114
14.8.2 解答 . . . 114
14.9 . . . 114
14.9.1 分析 . . . 115
14.9.2 解答 . . . 115
14.10 . . . 115
14.10.1 分析 . . . 115
14.10.2 解答 . . . 115

第十五章 Greedy- 19 **117**
15.1 . . . 117
15.1.1 分析 . . . 117
15.1.2 解答 . . . 117
15.2 . . . 117
15.2.1 分析 . . . 117
15.2.2 解答 . . . 117
15.3 . . . 117
15.3.1 分析 . . . 117
15.3.2 解答 . . . 117
15.4 . . . 117
15.4.1 分析 . . . 118
15.4.2 解答 . . . 118
15.5 . . . 118
15.5.1 分析 . . . 118
15.5.2 解答 . . . 118
15.6 . . . 118
15.6.1 分析 . . . 118
15.6.2 解答 . . . 118
15.7 . . . 118
15.7.1 分析 . . . 118
15.7.2 解答 . . . 118
15.8 . . . 118
15.8.1 分析 . . . 118
15.8.2 解答 . . . 118
15.9 . . . 118
15.9.1 分析 . . . 119
15.9.2 解答 . . . 119
15.10 . . . 119

15.10.1 分析 . . . 119
15.10.2 解答 . . . 119

第十六章 Sort- 16 **121**

16.1 冒泡 . . . 121
16.1.1 解答 . . . 121
16.2 选择排序 . . . 122
16.2.1 解答 . . . 122
16.3 快速排序 . . . 123
16.3.1 核心功能分责 . . . 123
16.3.2 解答 . . . 123
16.4 插入排序 . . . 124
16.4.1 解答 . . . 125
16.5 归并排序 . . . 125
16.5.1 如何合并 . . . 125
16.5.2 如何分解 . . . 126
16.5.3 非递归版本 . . . 126
16.6 堆 . . . 129
16.6.1 解答 . . . 129
16.7 . . . 129
16.7.1 分析 . . . 130
16.7.2 解答 . . . 130
16.8 . . . 130
16.8.1 分析 . . . 130
16.8.2 解答 . . . 130
16.9 . . . 130
16.9.1 分析 . . . 130
16.9.2 解答 . . . 130
16.10 . . . 130
16.10.1 分析 . . . 130
16.10.2 解答 . . . 130

第十七章 Bit Manipulation- 26 **131**

17.1 . . . 131
17.1.1 分析 . . . 131
17.1.2 解答 . . . 131
17.2 . . . 131
17.2.1 分析 . . . 131
17.2.2 解答 . . . 131
17.3 . . . 131

17.3.1 分析 . . . 131
17.3.2 解答 . . . 131
17.4 . . . 131
17.4.1 分析 . . . 132
17.4.2 解答 . . . 132
17.5 . . . 132
17.5.1 分析 . . . 132
17.5.2 解答 . . . 132
17.6 . . . 132
17.6.1 分析 . . . 132
17.6.2 解答 . . . 132
17.7 . . . 132
17.7.1 分析 . . . 132
17.7.2 解答 . . . 132
17.8 . . . 132
17.8.1 分析 . . . 132
17.8.2 解答 . . . 132
17.9 . . . 132
17.9.1 分析 . . . 133
17.9.2 解答 . . . 133
17.10 . . . 133
17.10.1 分析 . . . 133
17.10.2 解答 . . . 133

第十八章 Tree- 48 **135**

18.1 Binary Tree Maximum Path Sum . . . 135
18.1.1 分析 . . . 135
18.1.2 解答 . . . 135
18.2 Validate Binary Search Tree . . . 136
18.2.1 分析 . . . 137
18.2.2 解答 . . . 137
18.3 求二叉树中的节点个数 . . . 137
18.3.1 分析 . . . 137
18.3.2 解答 . . . 137
18.4 求二叉树的深度 . . . 137
18.4.1 分析 . . . 137
18.4.2 解答 . . . 137
18.5 前序遍历，中序遍历，后序遍历 . . . 137
18.5.1 已知前序、中序遍历，求后序遍历 . . . 137

18.5.2 已知中序和后序遍历，求前序遍历 . . . 137
18.5.3 总结 . . . 138
18.5.4 代码实现思路 . . . 140
18.6 分层遍历二叉树（按层次从上往下，从左往右） . . . 140
18.6.1 分析 . . . 140
18.6.2 解答 . . . 140
18.7 将二叉查找树变为有序的双向链表 . . . 140
18.7.1 分析 . . . 140
18.7.2 解答 . . . 140
18.8 求二叉树第 K 层的节点个数 . . . 140
18.8.1 分析 . . . 140
18.8.2 解答 . . . 140
18.9 求二叉树中叶子节点的个数 . . . 140
18.9.1 分析 . . . 140
18.9.2 解答 . . . 140
18.10 判断两棵二叉树是否结构相同 . . . 140
18.10.1 分析 . . . 140
18.10.2 解答 . . . 140
18.11 判断二叉树是不是平衡二叉树 . . . 140
18.11.1 分析 . . . 140
18.11.2 解答 . . . 140
18.12 求二叉树的镜像 . . . 140
18.12.1 分析 . . . 140
18.12.2 解答 . . . 140
18.13 求二叉树中两个节点的最低公共祖先节点 . . . 140
18.13.1 分析 . . . 140
18.13.2 解答 . . . 140
18.14 求二叉树中节点的最大距离 . . . 140
18.14.1 分析 . . . 140
18.14.2 解答 . . . 140
18.15 由前序遍历序列和中序遍历序列重建二叉树 . . . 140
18.15.1 分析 . . . 140
18.15.2 解答 . . . 140
18.16 判断二叉树是不是完全二叉树 . . . 140
18.16.1 分析 . . . 140
18.16.2 解答 . . . 140

第十九章 Depth-first Search- 41 **141**
19.1 部分和问题 . . . 141

19.1.1 分析 . . . 141
19.1.2 解答 . . . 141
19.2 . . . 142
19.2.1 分析 . . . 142
19.2.2 解答 . . . 142
19.3 . . . 142
19.3.1 分析 . . . 142
19.3.2 解答 . . . 142
19.4 . . . 142
19.4.1 分析 . . . 142
19.4.2 解答 . . . 142
19.5 . . . 142
19.5.1 分析 . . . 142
19.5.2 解答 . . . 142
19.6 . . . 142
19.6.1 分析 . . . 143
19.6.2 解答 . . . 143
19.7 . . . 143
19.7.1 分析 . . . 143
19.7.2 解答 . . . 143
19.8 . . . 143
19.8.1 分析 . . . 143
19.8.2 解答 . . . 143
19.9 . . . 143
19.9.1 分析 . . . 143
19.9.2 解答 . . . 143
19.10 . . . 143
19.10.1 分析 . . . 143
19.10.2 解答 . . . 143

第二十章 Breadth-first Search- 21 **145**

20.1 . . . 145
20.1.1 分析 . . . 145
20.1.2 解答 . . . 145
20.2 . . . 145
20.2.1 分析 . . . 145
20.2.2 解答 . . . 145
20.3 . . . 145
20.3.1 分析 . . . 145

20.3.2 解答 . . . 145
20.4 . . . 145
20.4.1 分析 . . . 146
20.4.2 解答 . . . 146
20.5 . . . 146
20.5.1 分析 . . . 146
20.5.2 解答 . . . 146
20.6 . . . 146
20.6.1 分析 . . . 146
20.6.2 解答 . . . 146
20.7 . . . 146
20.7.1 分析 . . . 146
20.7.2 解答 . . . 146
20.8 . . . 146
20.8.1 分析 . . . 146
20.8.2 解答 . . . 146
20.9 . . . 146
20.9.1 分析 . . . 147
20.9.2 解答 . . . 147
20.10 . . . 147
20.10.1 分析 . . . 147
20.10.2 解答 . . . 147

第二十一章 Union Find- 6 **149**
21.1 . . . 149
21.1.1 分析 . . . 149
21.1.2 解答 . . . 149
21.2 . . . 149
21.2.1 分析 . . . 149
21.2.2 解答 . . . 149
21.3 . . . 149
21.3.1 分析 . . . 149
21.3.2 解答 . . . 149
21.4 . . . 149
21.4.1 分析 . . . 149
21.4.2 解答 . . . 149

第二十二章 Graph- 10 **151**
22.1 . . . 151
22.1.1 分析 . . . 151

22.1.2 解答 . . . 151
22.2 . . . 151
22.2.1 分析 . . . 151
22.2.2 解答 . . . 151
22.3 . . . 151
22.3.1 分析 . . . 151
22.3.2 解答 . . . 151
22.4 . . . 151
22.4.1 分析 . . . 152
22.4.2 解答 . . . 152
22.5 . . . 152
22.5.1 分析 . . . 152
22.5.2 解答 . . . 152
22.6 . . . 152
22.6.1 分析 . . . 152
22.6.2 解答 . . . 152
22.7 . . . 152
22.7.1 分析 . . . 152
22.7.2 解答 . . . 152
22.8 . . . 152
22.8.1 分析 . . . 152
22.8.2 解答 . . . 152
22.9 . . . 152
22.9.1 分析 . . . 153
22.9.2 解答 . . . 153
22.10 . . . 153
22.10.1 分析 . . . 153
22.10.2 解答 . . . 153

第二十三章 Design- 27 **155**

23.1 . . . 155
23.1.1 分析 . . . 155
23.1.2 解答 . . . 155
23.2 . . . 155
23.2.1 分析 . . . 155
23.2.2 解答 . . . 155
23.3 . . . 155
23.3.1 分析 . . . 155
23.3.2 解答 . . . 155

23.4 . . . 155
23.4.1 分析 . . . 156
23.4.2 解答 . . . 156
23.5 . . . 156
23.5.1 分析 . . . 156
23.5.2 解答 . . . 156
23.6 . . . 156
23.6.1 分析 . . . 156
23.6.2 解答 . . . 156
23.7 . . . 156
23.7.1 分析 . . . 156
23.7.2 解答 . . . 156
23.8 . . . 156
23.8.1 分析 . . . 156
23.8.2 解答 . . . 156
23.9 . . . 156
23.9.1 分析 . . . 157
23.9.2 解答 . . . 157
23.10 . . . 157
23.10.1 分析 . . . 157
23.10.2 解答 . . . 157

第二十四章 Topological Sort[拓扑排序]- 5 **159**
24.1 . . . 159
24.1.1 分析 . . . 159
24.1.2 解答 . . . 159
24.2 . . . 159
24.2.1 分析 . . . 159
24.2.2 解答 . . . 159
24.3 . . . 159
24.3.1 分析 . . . 159
24.3.2 解答 . . . 159

第二十五章 Trie[前缀-字典树]- 7 **161**
25.1 . . . 161
25.1.1 分析 . . . 161
25.1.2 解答 . . . 161
25.2 . . . 161
25.2.1 分析 . . . 161
25.2.2 解答 . . . 161

25.3 . . . 161
25.3.1 分析 . . . 161
25.3.2 解答 . . . 161

第二十六章 Bianry Indexed Tree- 4 **163**
26.1 . . . 163
26.1.1 分析 . . . 163
26.1.2 解答 . . . 163
26.2 . . . 163
26.2.1 分析 . . . 163
26.2.2 解答 . . . 163
26.3 . . . 163
26.3.1 分析 . . . 163
26.3.2 解答 . . . 163

第二十七章 Segment Tree- 4 **165**
27.1 . . . 165
27.1.1 分析 . . . 165
27.1.2 解答 . . . 165
27.2 . . . 165
27.2.1 分析 . . . 165
27.2.2 解答 . . . 165
27.3 . . . 165
27.3.1 分析 . . . 165
27.3.2 解答 . . . 165

第二十八章 Binary Search Tree- 4 **167**
28.1 . . . 167
28.1.1 分析 . . . 167
28.1.2 解答 . . . 167
28.2 . . . 167
28.2.1 分析 . . . 167
28.2.2 解答 . . . 167
28.3 . . . 167
28.3.1 分析 . . . 167
28.3.2 解答 . . . 167

第二十九章 Other Aspects **169**
29.1 Recursion[递归]- 2 . . . 169
29.1.1 . . . 169
29.1.2 分析 . . . 169

29.1.3 解答 . . . 169
29.2 Brainteaser[谜题]- 2 . . . 169
29.2.1 . . . 169
29.2.2 分析 . . . 169
29.2.3 解答 . . . 169
29.3 Memoization[以避免递归重复计算，如 Fibonacci(斐波那契) 问题]- 1 . . . 169
29.3.1 . . . 169
29.3.2 分析 . . . 170
29.3.3 解答 . . . 170
29.4 Queue- 3 . . . 170
29.4.1 . . . 170
29.4.2 . . . 170
29.5 Reservoir Sampling[随机抽样]- 2 . . . 170
29.5.1 . . . 170
29.6 Minimax[极小化极大算法]- 3 . . . 170
29.6.1 . . . 170
29.6.2 . . . 170
29.7 Other Interview . . . 171
29.7.1 迷宫字符串 . . . 171

第一章　算法基础

1.1　时间复杂度

n	log_2^n	n×log_2^n	n^2	n^3	2^n	$n!$
4	2	8	16	64	16	24
8	3	24	64	512	256	80320
10	3.32	33.2	100	1000	1024	3628800
16	4	64	256	4096	65536	$2.1 * 10^{13}$
32	5	160	1024	32768	4.3×10^9	$2.6 * 10^{35}$
128	7	896	16384	2097152	3.4×10^{38}	∞
1024	10	10240	1048576	1.07×10^9	∞	∞
10000	13.29	132877	10^8	10^{12}	∞	∞

表 1.1: 各个函数随 n 的增长函数值的变化

1.2　解题提醒

1.2.1　等价意思.. 简化题目

1.2.2　有无最优子结构

即是否可以用数学公式简化模型。

1.2.3　想通可行再实践

1.3　分治思想

1.3.1　基本概念

在计算机科学中，分治法是一种很重要的算法。字面上的解释是“分而治之”，就是把一个复杂的问题分成两个或更多的相同或相似的子问题，再把子问题分成更小的子问题……直到最后子问题可以简单的直接求解，原问题的解即子问题的解的合并。这个技巧是很多高效算法的基础，如排序算法 (快速排序，归并排序)，傅立叶变换 (快速傅立叶变换)……

任何一个可以用计算机求解的问题所需的计算时间都与其规模有关。问题的规模越小，越容易直接求解，解题所需的计算时间也越少。例如，对于 n 个元素的排序问题，当 n=1 时，不需任何计算。n=2 时，只要作一次比较即可排好序。n=3 时只要作 3 次比较即可，…。而当 n 较大时，问题就不那么容易处理了。要想直接解决一个规模较大的问题，有时是相当困难的。

1.3.2 基本思想及策略

分治法的设计思想是：将一个难以直接解决的大问题，分割成一些规模较小的相同问题，以便各个击破，分而治之。

分治策略是：对于一个规模为 n 的问题，若该问题可以容易地解决（比如说规模 n 较小）则直接解决，否则将其分解为 k 个规模较小的子问题，这些子问题互相独立且与原问题形式相同，递归地解这些子问题，然后将各子问题的解合并得到原问题的解。这种算法设计策略叫做分治法。

如果原问题可分割成 k 个子问题，1<k n，且这些子问题都可解并可利用这些子问题的解求出原问题的解，那么这种分治法就是可行的。由分治法产生的子问题往往是原问题的较小模式，这就为使用递归技术提供了方便。在这种情况下，反复应用分治手段，可以使子问题与原问题类型一致而其规模却不断缩小，最终使子问题缩小到很容易直接求出其解。这自然导致递归过程的产生。分治与递归像一对孪生兄弟，经常同时应用在算法设计之中，并由此产生许多高效算法。

1.3.3 分治法适用的情况

分治法所能解决的问题一般具有以下几个特征：

- 该问题的规模缩小到一定的程度就可以容易地解决: 绝大多数问题都可以满足的，因为问题的计算复杂性一般是随着问题规模的增加而增加；
- 该问题可以分解为若干个规模较小的相同问题，即该问题具有最优子结构性质: 应用分治法的前提它也是大多数问题可以满足的，此特征反映了递归思想的应用；
- 利用该问题分解出的子问题的解可以合并为该问题的解: 是关键，能否利用分治法完全取决于问题是否具有第三条特征，如果具备了第一条和第二条特征，而不具备第三条特征，则可以考虑用贪心法或动态规划法。
- 该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子子问题: 涉及到分治法的效率，如果各子问题是不独立的则分治法要做许多不必要的工作，重复地解公共的子问题，此时虽然可用分治法，但一般用动态规划法较好

1.3.4 基本步骤

分治法在每一层递归上都有三个步骤：

step1 分解：将原问题分解为若干个规模较小，相互独立，与原问题形式相同的子问题；

step2 解决：若子问题规模较小而容易被解决则直接解，否则递归地解各个子问题

step3 合并：将各个子问题的解合并为原问题的解。

1.3.5 可使用分治法求解的一些经典问题

- 二分搜索
- 大整数乘法
- Strassen 矩阵乘法
- 棋盘覆盖
- 合并排序
- 快速排序
- 线性时间选择
- 最接近点对问题
- 循环赛日程表
- 汉诺塔

1.3.6 依据分治法设计程序时的思维过程

1、一定是先找到最小问题规模时的求解方法
2、然后考虑随着问题规模增大时的求解方法
3、找到求解的递归函数式后（各种规模或因子），设计递归程序即可。

1.4 动态规划思想

http://mp.weixin.qq.com/s/3h9iqU4rdH3EIy5m6AzXsg

1.4.1 基本概念

动态规划过程是：每次决策依赖于当前状态，又随即引起状态的转移。一个决策序列就是在变化的状态中产生出来的，所以，这种多阶段最优化决策解决问题的过程就称为动态规划。

1.4.2 基本思想和策略

基本思想与分治法类似，也是将待求解的问题分解为若干个子问题（阶段），按顺序求解子阶段，前一子问题的解，为后一子问题的求解提供了有用的信息。在求解任一子问题时，列出各种可能的局部解，通过决策保留那些有可能达到最优的局部解，丢弃其他局部解。依次解决各子问题，最后一个子问题就是初始问题的解。

由于动态规划解决的问题多数有重叠子问题这个特点，为减少重复计算，对每一个子问题只解一次，将其不同阶段的不同状态保存在一个二维数组中。

与分治法最大的差别是：适合于用动态规划法求解的问题，经分解后得到的子问题往往不是互相独立的（即下一个子阶段的求解是建立在上一个子阶段的解的基础上，进行进一步的求解）。

1.4.3 适用的情况

能采用动态规划求解的问题的一般要具有 3 个性质：

(1) 最优化原理：如果问题的最优解所包含的子问题的解也是最优的，就称该问题具有最优子结构，即满足最优化原理。

(2) 无后效性：即**某阶段状态一旦确定，就不受这个状态以后决策的影响**。也就是说，某状态以后的过程不会影响以前的状态，只与当前状态有关。

(3) 有重叠子问题：即子问题之间是不独立的，一个子问题在下一阶段决策中可能被多次使用到。（该性质并不是动态规划适用的必要条件，但是如果没有这条性质，动态规划算法同其他算法相比就不具备优势）

1.4.4 例题说明

上楼梯

有一座高度是 10 级台阶的楼梯，从下往上走，每跨一步只能向上 1 级或者 2 级台阶。要求用程序来求出一共有多少种走法。

比如，每次走 1 级台阶，一共走 10 步，这是其中一种走法。我们可以简写成 1,1,1,1,1,1,1,1,1,1。

再比如，每次走 2 级台阶，一共走 5 步，这是另一种走法。我们可以简写成 2,2,2,2,2。

当然，除此之外，还有很多很多种走法。

穷举法 二叉树

动态规划法

分析 **设**走到 m 层需要 x 种走法，**这个套路大家肯定非常熟悉，中学数学最喜欢设各种东西**。好，现在要开始求解 x 了，**首先**我们自然会想到先**找找**看这个 x **跟什么有关系**，看**能不能跟谁联立一个方程啥的不就好整了么**？

问题本身会提供有用信息，这是中学数学的套路，我们看到题目说每一次只能跨一步或两步，**也就是说**要想到达 m 层，我们只能从 $m-1$ 层跨一步上去或者从 $m-2$ 层跨两步上去。**假设**到达 $m-1$ 层有 $x1$ 种**走法**，到达 $m-2$ 层有 $x2$ 种走法，那么：

$$x = x1 + x2$$

这块有个思维陷阱，就是很容易想到从 $m-2$ 层到 m 层有两种方法，所以 $x = x1 + 2 * x2$, 这样思考是对的，是包含了一步步上的和 2 步上的，但是忽略了 $x1$ 包含了第一种情况，故 $x2$ 仅剩下 2 步到达 m 的情况了。

要点从上式中可以看出所求到达 m 层的走法 x 是依附于到达 $m-1$ 层的走法 $x1$ 和到达 $m-2$ 层的走法 $x2$。看到没有，**一个问题的解依附于其子问题的解** (*动态规划的精髓所在*)。也就是说只要我们知道了 $m-1$ 层的走法 $x1$ 和 $m-2$ 层的走法 $x2$ 就能知道到达 m 层的走法 x 了。

既然我们知道 m 层的走法能从 $m-1$ 层和 $m-2$ 层的走法求得，**那同样**的道理，$m-1$ 层的走法不也可以从 $(m-1)-1$ 和 $(m-1)-2$ 层的走法求得, 换种简单的描述方法，用 $F(m)$ 表示到达 m 层的方法数，那么就有

$$F(m) = F(m-1) + F(m-2)$$

$$F(m-1) = F(m-1-1) + F(m-1-2)$$

当只有 1 个台阶和 2 个台阶时，显然可以看出分别有 1 种与 2 种走法

$$F(1) = 1$$

$$F(2) = 2$$

理论对应 动态规划包括 3 个重要的概念

- **最优子结构**：我们分析到的 $x = x1 + x2$ 就是最优子结构，**将主问题的解依附于相似的子问题的假设解上**。
- **状态转移公式**：$F(m) = F(m-1) + F(m-2)$ **一个问题的解依附于其子问题的解**
- **边界**：当只有 1 级和 2 级台阶时，我们**可以直接得到结果，无需继续简化**，这时我们称 $F(1)$ 与 $F(2)$ 为问题的边界

求解技巧 存储已经计算过的结果，避免重复计算。

1.4.5 题目

入门http://www.cnblogs.com/zedosu/p/6556000.html

进阶http://www.cnblogs.com/zedosu/p/6556392.html

1.5 贪心算法思想

1.5.1 基本概念

所谓贪心算法是指，在对问题求解时，总是做出在当前看来是最好的选择。也就是说，不从整体最优上加以考虑，他所做出的仅是在某种意义上的局部最优解。

贪心算法没有固定的算法框架，算法设计的关键是贪心策略的选择。必须注意的是，贪心算法不是对所有问题都能得到整体最优解，选择的贪心策略必须具备无后效性，即某个状态以后的过程不会影响以前的状态，只与当前状态有关。

所以对所采用的贪心策略一定要仔细分析其是否满足无后效性。

1.5.2 基本思想和策略

1、建立数学模型来描述问题。
2、把求解的问题**分成若干个子问题**。
3、对每一子问题求解，得到子问题的**局部最优解**。
4、把子问题的解**局部最优解**合成原来解**问题的一个解**。

1.5.3 适用的情况

贪心策略适用的前提是：局部最优策略能导致产生全局最优解。

实际上，贪心算法适用的情况很少。一般，对一个问题分析是否适用于贪心算法，可以先选择该问题下的几个实际数据进行分析，就可做出判断。

1.5.4 求解的基本步骤

从问题的某一初始解出发；

```
while(能朝给定总目标前进一步)
{
    利用可行的决策，求出可行解的一个解元素；
}
```

由所有解元素组合成问题的一个可行解；

1.5.5 算法实现的说明

背包问题

1.5.6 与动态规划的区别

http://blog.csdn.net/jarvischu/article/details/6056387

1.6 参考

算法过来人 + 面试题:http://blog.csdn.net/qingyuanluofeng/article/details/47612589

第二章　剑指

2.1　基础知识

2.1.1　C++

面试题 1：赋值运算符　核心考点：

1. 返回为引用，否则无法连续赋值
2. 参数声明为常引用
3. 忘记分配新内存之前释放原有内存会内存泄露
4. 判断参数与当前实例是否相同

```
class CMyString1
{
public:
    CMyString1(char* pData = NULL);
    CMyString1(const CMyString1& str);
    ~CMyString1(void);

    //通过创建临时对象，再让临时对象与原有对象交换，临时对象指向原来的内存，出括号后释放
    CMyString1& operator = (const CMyString1& str1)
    {
        if(this != &str1)
        {
            CMyString strTemp(str1);//在构造函数中用new分配内存，若有异常，状态有效
            char* tmpP = strTemp.m_pData;
            strTemp.m_pData = m_pData;
            m_pData = tmpP;
        }
        return *this;
    }
private:
    char* m_pData;
};
```

面试题 2 实现 Singleton 模式 核心考点：

1. singleton 模型实现
2. 线程安全如何保证- doubleCheck
3. private 构造
4. static 成员

```
template<typename T>
class Singleton
{
public:
    static T& getInstance()
    {
        if(!value_)
        {
            MutexGuard guard(mutex_);
            if (!value_)
            {
                value_ = new T();
            }
        }
        return *value_;
    }

private:
    Singleton();
    ~Singleton();

    static T* value_;
    static Mutex mutex_;
};

template<typename T>
T* Singleton<T>::value_ = NULL;

template<typename T>
Mutex Singleton<T>::mutex_;
```

面试题 48 不能被继承的类

2.1.2 数据结构

线性表

面试题 5 从尾到头打印链表

面试题 7 用两个栈实现队列 ->

```
void EnQueue(Stack *s1, Stack *s2, int k)
{
    Push(s1, k);
}

int DeQueue(Stack *s1, Stack*s2)
{
    if(IsStackEmpty(s2) == 1)
    {
        while(IsStackEmpty(s1) == 0)
        {
            Push(s2, Pop(s1));
        }
    }
    if(IsStackEmpty(s2) == 1)
    {
        printf("Empty!\n");
    }
    return Pop(s2);
}
```

面试题 8 旋转数组的最小数字

树

面试题 6 二叉树中前序确定后续 参考http://www.cnblogs.com/edisonchou/p/4741099.html

堆

红黑树

B 树

hash table

2.1.3 解题分析

面试题 10 二进制中 1 的个数

查找数组中第 k 大的数字

2.2 高质量代码

http://blog.csdn.net/oMengLiShuiXiang1234/article/details/51785436

2.2.1 代码的质量

下面是几个面试官对代码质量的要求

- **代码的容错能力**，对一些**特别的输入**需要**考虑异常状况**，考虑**资源的回收问题**。
- 一些基本的知识点，如 double 类型的数据比较的问题。`if( d1==d2 )` 上述比较有问题，由于精度原因不能用等号判断两个小数是否相等。
- **不能忽略边界的情况**
- 变量、函数**命名的问题**，而且解决一个具体的问题，需要有个**合适的数据结构**。
- 从程序的**正确性和鲁棒性**两方面检验代码的质量。关注对**输入参数的检查、处理错误和异常的方式、命名方式**等

2.2.2 代码的规范性

面试官是根据应聘者写出的代码来决定是否录用他的。如果应聘者代码写的不够规范，影响面试官阅读代码的兴致，那面试官就会默默地减去几分。

- 规范的**代码书写清晰**

  写代码前形成清晰的思路并能把思路用编程语言清楚地书写出来。

- 规范的**代码布局清晰**

  当循环、判断较多，逻辑较复杂时，缩进的层次可能会比较多。

- 规范的代码**命名合理**

建议：在写代码的时候，用完整的英文单词组合命名变量和函数。

2.2.3 代码的完整性

在面试的过程中，面试官会**非常关注**应聘者**考虑问题是否周全**。面试官通过**检查代码是否完整**来考查应聘者的思维是否全面。通常面试官会检查应聘者的代码是否完成了基本功能、**输入边界值**是否能得到正确的输出、**是否对各种不合规范的非法输入做出了合理的错误处理**。

面试题 11 数值的整数次方

面试题 12 打印 1 到最大的 n 位数

面试题 13 在 O(1) 时间删除链表节点

面试题 14 调整数组顺序使奇数位于偶数掐面

2.2.4 代码的鲁棒性

鲁棒性（Robust），有时也翻译成健壮性。所谓鲁棒性是指程序能够判断输入是否合乎规范要求，并对不合要求的输入予以合理的处理。

容错性是鲁棒性的一个重要体现。不鲁棒性的软件在发生异常事件的时候，比如用户输入错误的用户名、试图打开的文件不存在或者网络不能连接，就会出现不可预见的诡异行为，或者干脆整个软件崩溃。这样的软件对于用户而言，不亚于一场灾难。

由于鲁棒性对软件开发非常重要，面试官在招聘的时候对应聘者写出的代码是否具有鲁棒性也非常关注。提高代码的鲁棒性的有效途径是进行防御性变成。防御性编程是一种编程习惯，是指遇见在什么地方可能会出现问题，并为这些可能出现的问题制定处理方式。比如试图打开文件时发现文件不存在，我们可以提示用户检查文件名和路径；当服务器连接不上时，我们可以试图连接备用服务器等。这样异常发生时，软件的行为也尽在我们的掌握之中，而不至于出现不可预见的事情。

在面试时，最简单也是最使用的防御性编程就是在函数入口添加代码以验证用户输入是否符合要求。通常面试要求写一两个函数，我们需要格外关注这些函数的输入参数。如果输入的是一个指针，那指针时空指针怎么办？如果输入的是一个字符串，那么字符串的内容为空怎么办？如果能把这些问题都提前考虑到，并做相应的处理，那么面试官就会觉得我们有防御性编程的习惯，能够写出鲁棒性的软件。

当然并不是所有与鲁棒性相关的问题都只是检查输入的参数这么简单。我们看到问题的时候，要多问几个“如果不... 那么...”这样的问题。比如面试题 15“链表中倒数第 K 个结点”，这里隐含一个条件就是链表中结点的个数大于 k。我们要问如果链表中的结点的数目不是大于 k 个，那么，代码会出现什么问题？主要的思考方式能够帮助我们发现潜在的问题并提前解决问题。这比让面试官发现问题之后我们再去慌忙分析代码查找问题的根源要好得多。

面试题 15 链表中倒数第 k 个结点

面试题 16 反转链表

面试题 17 合并两个排序的链表

面试题 18 树的子结构

2.2.5 小结

- 规范性：书写清晰、布局清晰、命名合理
- 完整性：**完成基本功能、考虑边界条件、做好错误处理**
- 鲁棒性：采取**防御式编程、处理无效的输入**

2.3 解题思路

2.3.1 画图让抽象形象化

面试题 19 二叉树的镜像

面试题 20 顺时针打印矩阵

2.3.2 举例让抽象具体化

面试题 21 包含 min 函数的栈

面试题 22 栈的压入、弹出序列

面试题 23 从上往下打印二叉树

面试题 24 二叉搜索树的后序遍历序列

面试题 25 二叉树中和为某一值的路径

2.3.3 分解让复杂问题简化

面试题 26 复杂链表的复制

面试题 27 二叉搜索树与双向链表

面试题 28 字符串的排列

2.4 优化时间和空间效率

2.4.1 时间效率

面试题 29 数组中出现次数超过一半的数字

面试题 30 最小的 k 个数

面试题 31 连续子数组的最大和

面试题 32 从 1 到 n 整数中 1 出现的次数

面试题 33 把数组排成最小的数

- 整体可以看成一个排好序的数
- 排序规则

实现 ->

```
#include <iostream>
#include <string>
#include <sstream>
#include <algorithm>
using namespace std;

bool compare(const string& str1, const string &str2)
{
    string s1=str1+str2;
    string s2=str2+str1;
    return s1<s2;
}

void ComArrayMin(int *pArray, int num)
{
    int i;
    string *pStrArray=new string[num];

    for(i=0; i<num; i++)
    {
        stringstream stream;
        stream<<pArray[i];
        stream>>pStrArray[i];
    }

    sort(pStrArray, pStrArray+num, compare);

    for(i=0; i<num; i++)
    cout<<pStrArray[i];

    cout<<endl;

    delete[] pStrArray;

}

void main()
{
    int Num;
    cin>>Num;
```

```
    int *pArray=new int[Num];

    for(int i=0; i<Num; i++)
    cin>>pArray[i];

    ComArrayMin(pArray, Num);

}
```

2.4.2 时间效率与空间效率的平衡

面试题 34 丑数

- 是否能想到只构造丑数
- 构造过程的双指针操作，选择当前最小的数，然后移动..

实现 ->

```
int FindUgly(int n) //
{
    int* ugly = new int[n];
    ugly[0] = 1;
    int index2 = 0;
    int index3 = 0;
    int index5 = 0;
    int index = 1;
    while (index < n)
    {
        int val = Min(ugly[index2]*2, ugly[index3]*3, ugly[index5]*5); //竞争产生下一个
            丑数
        if (val == ugly[index2]*2) //将产生这个丑数的index*向后挪一位;
            ++index2;
        if (val == ugly[index3]*3) //这里不能用elseif，因为可能有两个最小值，这时都要挪
            动;
            ++index3;
        if (val == ugly[index5]*5)
            ++index5;
        ugly[index++] = val;
    }
    /*
    for (int i = 0; i < n; ++i)
    cout << ugly[i] << endl;
    //*/
    int result = ugly[n-1];
    delete[] ugly;
    return result;
```

```
}
```

面试题 35 第一个只出现一次的字符

- 出现次数
- 查找第一次出现为 1 次的

```
#include <unordered_map>
class Solution {
    public:
    int FirstNotRepeatingChar(string str) {
        unordered_map<char, int> all;
        for (int i = 0; i < (int)str.length(); ++i)
        {
            if (all.find(str[i]) != all.end())
            ++all[str[i]];
            else all.insert(make_pair(str[i], 1));
        }

        for (int i = 0; i < (int)str.length(); ++i)
        {
            if (all[str[i]] == 1)
            return i;
        }
        return -1;
    }
};
```

面试题 36 数组中的逆序对 http://blog.csdn.net/imzoer/article/details/8050224

面试题 37 两个链表的第一个公共结点

2.5 能力体现篇

2.5.1 知识迁移能力

面试题 38 数字在排序数组中出现的次数

面试题 39 二叉树的深度

面试题 40 数组中只出现一次的数字

面试题 41 和为 s 的多个数字 https://www.nowcoder.com/practice/c451a3fd84b64cb19485dad75tpId=13&tqId=11194&tPage=3&rp=3&ru=/ta/coding-interviews&qru=/ta/coding-interviews/question-ranking

1. 数组连续问题是否能想到 **双指针解决方法**

```
vector<vector<int> > FindContinuousSequence(int sum) {
    int low = 1, high = 1;
    int _sum = 0;
    vector<vector<int> >all;
    for(int i = 1; i <= sum; ++i)
    {
        if(_sum == sum)
        {
            vector<int> temp;
            for(int j = low; j < high; ++j)
            {
                temp.push_back(j);
            }
            all.push_back(temp);
            temp.clear();
            _sum += high++;
        }
        else if(_sum>sum)
        {
            _sum -= low;
            ++low;
        }
        else
        {
            _sum += high;
            ++high;
        }
    }
    return all;
}
```

面试题 42 反转单词顺序

1. string 类的应用

```
class Solution {
    public:
    string ReverseSentence(string str) {
        string temp;
        int low;
        int high = str.length();
```

```
        for (low = str.length(); low >= 0; --low)
        {
            if (str[low] == '␣')
            {
                if (low + 1 < str.length())
                {
                    temp += string(&str[low+1], &str[high]);
                    temp += '␣';
                    high = low;
                }
            }
            else if (low == 0)
            {
                temp += string(&str[low], &str[high]);
            }
        }
        temp += '\0';
        return temp;
    }
};
```

2.5.2 抽象建模能力

面试题 43 n 个骰子的点数

面试题 44 扑克牌的顺子

面试题 45 圆圈中最后剩下的数字

2.5.3 思维发散能力

面试题 46 求 1+2+...+n

面试题 47 不用加减乘除做加法

面试题 49 把字符串转换成整数

面试题 50 树中两个节点的最低公共祖先

2.6 英文新增篇

2.6.1 数组

面试题 51 数组中重复的数-O(n)

1.

2.

构建乘积数组

2.6.2 字符串

面试题 52 正则表达式匹配 https://www.nowcoder.com/practice/45327ae22b7b413ea21df13ee7tpId=13&tqId=11205&tPage=3&rp=3&ru=/ta/coding-interviews&qru=/ta/coding-interviews/question-ranking

1. 是否能考虑到用递归解决问题：简化问题的能力。
2. 是否能把问题考虑全面

表示数值的字符串

2.6.3 链表

面试题 53 链表中环的入口节点

1. 是否存在环
2. 环的节点个数
3. pFast 先移动环的个数，然后 pSlow 从头节点开始

2.6.4 树

二叉树的下一个节点

对称二叉树

把二叉树打印成多行

按之字形打印二叉树

序列化二叉树

二叉搜索树的第 K 个节点

数据流中的中位数

2.6.5 回溯法

滑动窗口的最大值

矩阵中的路径

机器人的运动范围

第三章　编程之美

http://blog.csdn.net/qingyuanluofeng/article/category/2544589

3.1　数字中的技巧

3.1.1　求二进制数中 1 的个数

对于一个字节的无符号整型变量，求其二进制表示中 1 的个数，要求算法的执行效率尽可能高。

3.1.2　不要被阶乘吓到

3.1.3　寻找超过占比 1/2 的 ID

本质: 寻找出现超过一半次数的人

- 转化为更小的问题-Delete Copied
- 如何避免排序超找重复

```
#include<iostream>
#include<vector>
using namespace std;

int find1(vector<int>&id)
{
    int a;
    int ta=0;

    int len = id.size();

    for (int i = 0; i < len; i++){
        if (ta == 0){
            a = id[i];
            ta = 1;
        }
        else{
```

```
            if (a == id[i])
                ta++;
            else
                ta--;
        }
    }

    return a;
}
```

3.1.4 1 的数目

给定一个十进制正整数 N，写下从 1 开始，到 N 的所有整数然后数一下其中出现的所有 1 的个数　例如:

N=2, 写下 1,2 这样只出现了 1 个 1

N=12, 写下 1,2,3,4,5，6,7，8,9,10,11,12, 这样 1 的个数是 5

问题是:　写一个函数 f(N), 返回 1 到 N 之间出现的 1 的个数，比如 f(12) = 5

核心　规律总结

```
long long count1_divide(int n)
{
    long long lCount = 0;
    long long lHigh,lLow,lCur;
    lHigh = lLow = lCur = 0;
    long long lFactor = 1;
    while(n / lFactor)
    {
        lLow = n - (n/lFactor)*lFactor;//参见123 - (123/10)*10
        lCur = (n / lFactor) % 10;//参见(123/10)%10
        lHigh = n / (lFactor * 10);//参见123 / (10*10)
        switch(lCur)
        {
            case 0: lCount += lHigh * lFactor;//如果当前位为0，那么当前位出现1的次数 = 高
                位*位数
                break;
            case 1: lCount += lHigh * lFactor + lLow + 1;//如果当前位为1，那么当前位出现1
                的次数 = 高位 * 位数 + 低位 + 1
                break;
            default: lCount += (lHigh + 1) * lFactor; //如果当前位>=2，那么当前为出现1的
                次数 = (高位+1)*位数
        }
        lFactor *= 10;
    }
    return lCount;
}
```

3.1.5 寻找最大的 k 个数

3.1.6 精确表达浮点数

在计算机中，使用 float 或 double 来存储小数不能得到精确值。希望得到精确值，最好用分数形式来表示小数。有限小数或者无限循环小数可以转化为分数

例如:

```
0.9 = 9/10
0.333(3) = 1/3(括号中的数字表示是循环节)
//一个小数可以用好几种分数形式来表示。如:
0.333(3) = 1/3 = 3/9

//给定一个有限小数或者无限循环小数，你能否用分母最小的分数形式来返回这个小数呢?如果输入为
    循环小数，循环节用括号标记出来。下面是一些可能的输入数据，
如0.3,0.30,0.3(000),0.3333(3333)、......
```

3.1.7 最大公约数问题

3.1.8 找到符合条件的整数

3.1.9 斐波那契数列

3.1.10 寻找数组中的最大最小值

3.1.11 寻找最近点对

3.1.12 找到符合条件的两个数

3.1.13 子数组的最大乘积

给定一个长度为 N 的整数数组，只允许用乘法，不能用除法，计算任意 (N-1) 个数的组合中乘积最大的一组，并写出算法的时间复杂度

3.1.14 子数组之和的最大值

3.1.15 子数组之和的最大值 (二维)

3.1.16 求数组中最长递增子序列

写一个时间复杂度尽可能低的程序，求一个一维数组 (N 个元素) 中最长递增子序列的长度。

例如，在序列 1,-1,2,-3,4,-5,6,-7 中，最长递增子序列的长度为 4(如 1，2,4,6)

3.1.17 数组循环移位

3.1.18 数组分割

3.1.19 区间重合判断

给定一个源区间 [x,y]（y>=x）和 N 个无序的目标区间 [x1,y1],[x2,y2],[x3,y3],...,[xn,yn]，判断源区间 [x,y] 是不是在目标区间内 (也即 [x,y] 是否属于任意 [xi,yi])

例如：给定源区间 [1,6] 和一组无序的目标区间 [2,3][1,2][3,9]，即可认为区间 [1,6] 在区间 [2,3][1,2][3,9] 内 (因为目标区间实际上是 [1,9])

3.1.20 程序理解和时间分析

3.1.21 只考加法的面试题

3.2 字符串及链表的探索

3.2.1 字符串移位包含的问题

3.2.2 电话号码对应英语单词

3.2.3 计算字符串的相似度

3.2.4 从无头单链表中删除节点

3.2.5 最短摘要的生成

3.2.6 判断两个链表是否交叉

3.2.7 队列中取最大值操作问题

3.2.8 求二叉树中结点的最大距离

3.2.9 重建二叉树

输入某二叉树的前序遍历和中序遍历，请重建出该二叉树。假设输入的前序遍历和中序遍历的结果中都不含有重复的数字。

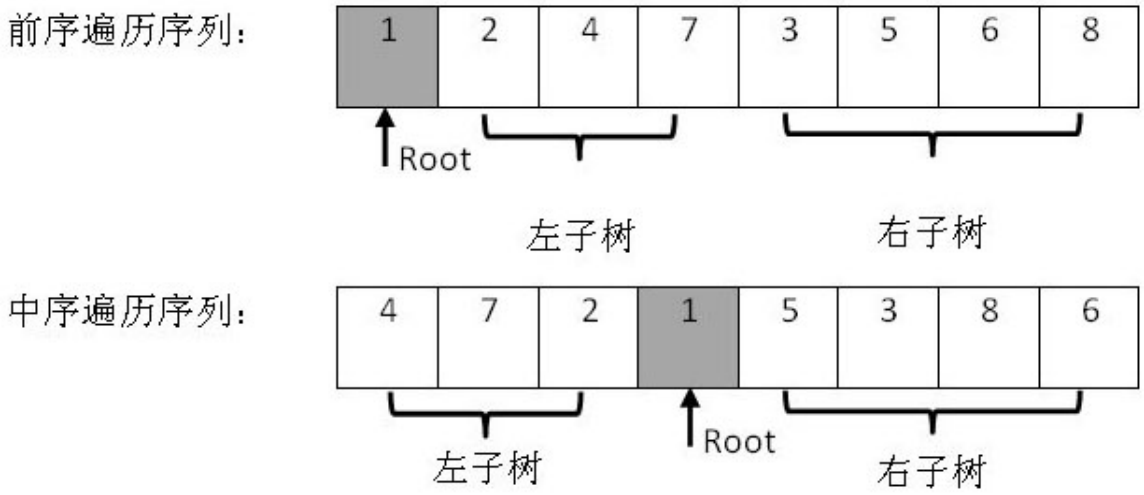
例如，前序遍历序列：1，2，3，7，3，5，6，8，中序遍历序列：4，7，2，1，5，3，8，6

分析

在二叉树的前序遍历序列中，第一个数字总是树的根结点的值。但在中序遍历序列中，根结点的值在序列的中间，左子树的结点的值位于根结点的值的左边，而右子树的结点的值位于根结点的值的右边。因此我们需要扫描中序遍历序列，才能找到根结点的值。

前序遍历序列的第一个数字 1 就是根结点的值。扫描中序遍历序列，就能确定根结点的值的位置。根据中序遍历特点，在根结点的值 1 前面的 3 个数字都是左子树结点的值，位于 1 后面的数字都是右子树结点的值。

在二叉树的前序遍历和中序遍历的序列中确定根结点的值、左子树结点的值和右子树结点的值的步骤如下图所示：



分别找到了左、右子树的前序遍历序列和中序遍历序列，我们就可以用同样的方法分别去构建左右子树。换句话说，这是一个递归的过程。

思路总结 ：先根据前序遍历序列的第一个数字创建根结点，接下来在中序遍历序列中找到根结点的位置，这样就能确定左、右子树结点的数量。在前序遍历和中序遍历的序列中划分了左、右子树结点的值之后，就可以递归地去分别构建它的左右子树。

要点

- 边界条件 `preOrder` 与 `inOrder` 是否为空
- 终止条件 `leftLength > 0` 与 `rightLength > 0` 时才重构子树
- 分解子树

```
public static Node<int> Construct(int[] preOrder, int[] inOrder, int length)
{
    // 空指针判断
    if (preOrder == null || inOrder == null || length <= 0)
    {
        return null;
    }

    return ConstructCore(preOrder, 0, preOrder.Length - 1, inOrder, 0, inOrder.Length
        - 1);
}
```

```
public static Node<int> ConstructCore(int[] preOrder, int startPreOrder, int
    endPreOrder, int[] inOrder, int startInOrder, int endInOrder)
{
    // 前序遍历序列的第一个数字是根结点的值
    int rootValue = preOrder[startPreOrder];
    Node<int> root = new Node<int>();
    root.data = rootValue;
    root.lchild = root.rchild = null;

    if (startPreOrder == endPreOrder)
    {
        if (startInOrder == endInOrder &&
        preOrder[startPreOrder] == inOrder[startInOrder])
        {
            return root;
        }
        else
        {
            throw new Exception("Invalid input!");
        }
    }

    // 在中序遍历中找到根结点的值
    int rootInOrder = startInOrder;
    while (rootInOrder <= endInOrder && inOrder[rootInOrder] != rootValue)
    {
        rootInOrder++;
    }

    // 输入的两个序列不匹配的情况
    if (rootInOrder == endInOrder && inOrder[rootInOrder] != rootValue)
    {
        throw new Exception("Invalid input!");
    }

    int leftLength = rootInOrder - startInOrder;
    int leftPreOrderEnd = startPreOrder + leftLength;
    if (leftLength > 0)
    {
            // 构建左子树
            root.lchild = ConstructCore(preOrder, startPreOrder + 1, leftPreOrderEnd,
                inOrder, startInOrder, rootInOrder - 1);
    }
    if (leftLength < endPreOrder - startPreOrder)
    {
            // 构建右子树
            root.rchild = ConstructCore(preOrder, leftPreOrderEnd + 1, endPreOrder,
```

```
                inOrder, rootInOrder + 1, endInOrder);
        }

        return root;
    }
```

3.2.10 分层遍历二叉树

- 递归
- 如何体现层次化

```
    void printLevelNode_index(Node* pRoot)
    {
        int iCur = 0,iLast = 1;//初始化游标起始位置和当前层结束位置的下一个位置
        vector<Node*> vecNode;
        vecNode.push_back(pRoot);
        while(iCur < vecNode.size())//这里应该是层结束位置等于数组大小的时候结束循环
        {
            iLast = vecNode.size();//牛逼，直接用当前层结束位置的下一位置 等于 向量大小 来解
                决了每次的更新层结束位置的问题
            while(iCur < iLast)
            {
                Node* pNode = vecNode[iCur];
                printf("%d ",pNode->_iVal);//应该先打印出当前节点值
                if(pNode->_pLeft)
                {
                    vecNode.push_back(pNode->_pLeft);
                }
                if(pNode->_pRight)
                {
                    vecNode.push_back(pNode->_pRight);
                }
                iCur++;
            }
            printf("\n");
        }
    }
```

3.2.11 二分查找

核心

- 防止溢出 `mid = minIndex + (maxIndex - minIndex)/2`
- 结束条件 `while( left <= right )`

3.3 数学游戏

3.3.1 金刚坐飞机问题

3.3.2 瓷砖覆盖地板

3.3.3 买票找零

3.3.4 点是否在三角形内

3.3.5 磁带文件存放优化

3.3.6 桶中取黑白球

3.3.7 蚂蚁爬杆

3.3.8 三角形测试用例

3.3.9 数独知多少

3.3.10 数字哑谜和回文

3.3.11 扫雷游戏的概率

第四章　Array- 80

4.1 Two Sum

4.1.1 问题

Given an array of integers, return **indices** of the two numbers such that they add up to a specific target. You may assume that each input would have exactly one solution.

Example Given nums = [2, 7, 11, 15], target = 9,
Because nums[0] + nums[1] = 2 + 7 = 9,
return [0, 1].

4.1.2 解答

4.1.3 Normal-两层循环

4.1.4 Good-HashTable:HashMap

- **Time complexity** : O(n). We traverse the list containing n elements only once. Each look up in the table costs only O(1) time.
- **Space complexity** :O(n).The extra space required depends on the number of items stored in the hash table, which stores at most n elements.

```
//\\\\\\\\\\\\\\\\\\\\\\\\\\Two-pass Hash Table\\\\\\\\\\\\\\\\\\\\\\\\\\\\
#include <unordered_map>
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
    vector<int> result;
    unordered_multimap<int, int> intergers;
    for(int i = 0; i < nums.size(); ++i)
    {
        // map 的插入值操作
        intergers.insert(pair<int,int>(nums[i],i));
    }
```

```
        for(unordered_multimap<int, int>::const_iterator d = intergers.begin(); d !=
            intergers.end(); ++d)
        {
            unordered_multimap<int, int>::const_iterator find_it = intergers.find(target -
                d->first);
            if(find_it != intergers.end() && find_it -> second != d -> second)
            {
                result.push_back(d->second);
                result.push_back(find_it -> second);
                break;
            }
        }
        // sort 函数使用
        sort(result.begin(),result.end());
        return result;
        }
    };

    //\\\\\\\\\\\\\\\\\\\\\\\\\\One-Pass HashTable\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
    Java:
    public int[] twoSum(int[] nums, int target) {
        Map<Integer, Integer> map = new HashMap<>();
        for (int i = 0; i < nums.length; i++) {
            int complement = target - nums[i];
            if (map.containsKey(complement)) {
                return new int[] { map.get(complement), i };
            }
            map.put(nums[i], i);
        }
        throw new IllegalArgumentException("No␣two␣sum␣solution");
    }
```

4.2 Median of Two Sorted Arrays

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)).

Examples nums1 = [1, 3] nums2 = [2]

The median is 2.0

nums1 = [1, 2] nums2 = [3, 4]

The median is (2 + 3)/2 = 2.5

4.2.1 分析

4.2.2 解答

4.3 Word Ladder II

Given two words (beginWord and endWord), and a dictionary's word list, find all shortest transformation sequence(s) from beginWord to endWord, such that:

1-Only one letter can be changed at a time

2-Each transformed word must exist in the word list. Note that beginWord is not a transformed word.

Note:

Return an empty list if there is no such transformation sequence.
All words have the same length.
All words contain only lowercase alphabetic characters.
You may assume no duplicates in the word list.
You may assume beginWord and endWord are non-empty and are not the same.

Examples Given:
beginWord = "hit"
endWord = "cog"
wordList = ["hot","dot","dog","lot","log","cog"]
Return
[
["hit","hot","dot","dog","cog"],
["hit","hot","lot","log","cog"]
]

4.3.1 分析

4.3.2 解答

`http://www.cnblogs.com/ShaneZhang/p/3748494.html`

4.4 4Sum

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: The solution set must not contain duplicate quadruplets.

Examples 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]

]

4.4.1 分析

4.4.2 解答

4.5 Rotate Array

Rotate an array of n elements to the right by k steps.

Note: Try to come up as many solutions as you can, there are at least 3 different ways to solve this problem.

Examples 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].

4.5.1 分析

4.5.2 解答

4.6 Maximum Product Subarray

Find the contiguous subarray within an array (containing at least one number) which has the largest product.

Examples Given the array [2,3,-2,4],

the contiguous subarray [2,3] has the largest product = 6.

4.6.1 分析

4.6.2 解答

4.7 Spiral Matrix

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

Examples 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].

4.7.1 分析

4.7.2 解答

4.8 First Missing Positive

Given an unsorted integer array, find the first missing positive integer.

Examples Given [1,2,0] return 3,

and [3,4,-1,1] return 2.

Your algorithm should run in O(n) time and uses constant space.

4.8.1 分析

4.8.2 解答

4.9 Word Search

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.

Examples Given board =

[

[’A’,’B’,’C’,’E’],

[’S’,’F’,’C’,’S’],

[’A’,’D’,’E’,’E’]

]

word = ”ABCCED”, -> returns true,

word = ”SEE”, -> returns true,

word = ”ABCB”, -> returns false.

4.9.1 分析

4.9.2 解答

4.10 Largest Rectangle in Histogram

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.

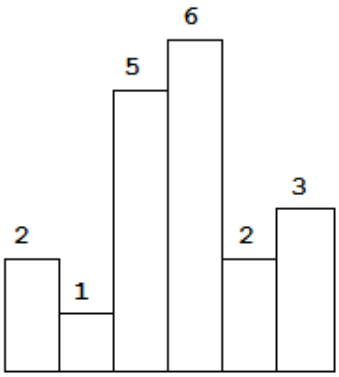


Fig 4.1: Above is a histogram where width of each bar is 1, given height = [2,1,5,6,2,3]

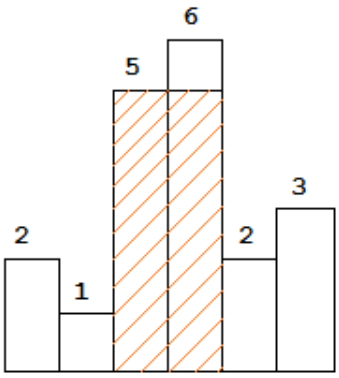


Fig 4.2: The largest rectangle is shown in the shaded area, which has area = 10 unit.

Examples Given heights = [2,1,5,6,2,3],
return 10.

4.10.1 分析

4.10.2 解答

4.11 Jump Game II

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.

Examples 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.)

Note: You can assume that you can always reach the last index.

4.11.1 分析

4.11.2 解答

4.12 Maximal Rectangle

Given a 2D binary matrix filled with 0's and 1's, find the largest rectangle containing only 1's and return its area.

Examples given the following matrix:

1 0 1 0 0

1 0 1 1 1

1 1 1 1 1

1 0 0 1 0

Return 6.

4.12.1 分析

4.12.2 解答

第五章　Hash Table- 52

5.1　Hash

Hash 表也称散列表，也有直接译作哈希表，Hash 表是一种特殊的数据结构，它同数组、链表以及二叉排序树等相比较有很明显的区别，它能够快速定位到想要查找的记录，而不是与表中存在的记录的关键字进行比较来进行查找。

Hash 表采用一个**映射函数** f : `key —> address` 将关键字映射到该记录在表中的存储位置，从而在想要查找该记录时，可以直接根据关键字和映射关系计算出该记录在表中的存储位置，通常情况下，这种映射关系称作为`Hash`函数，而通过 Hash 函数和关键字计算出来的存储位置 (注意这里的存储位置只是表中的存储位置，并不是实际的物理地址) 称作为`Hash`地址。

5.1.1　hash 函数的设计

1. **直接定址法**

   取关键字或者关键字的某个线性函数为 Hash 地址，即`address(key)=a*key+b`; 如知道学生的学号从 2000 开始，最大为 4000，则可以将`address(key)=key-2000` 作为 Hash 地址

2. **平方取中法**

   对关键字进行平方运算，然后取结果的中间几位作为 Hash 地址。假如有以下关键字序列`{421, 423, 436}`，平方之后的结果为`{177241, 178929, 190096}`，那么可以取`{72, 89, 00}` 作为 Hash 地址

3. **折叠法**

   将关键字拆分成几部分，然后将这几部分组合在一起，以特定的方式进行转化形成 Hash 地址。假如知道图书的`ISBN`号为`8903-241-23`，可以将`address(key)=89+03+24+12+3` 作为 Hash 地址。

4. **除留取余法**

   如果知道 Hash 表的最大长度为`m`，可以取不大于 m 的最大质数 p，然后对关键字进行取余运算，`address(key)=key%p`。

   在这里 p 的选取非常关键，p 选择的好的话，能够最大程度地减少冲突，**p 一般取不大于 m 的最大质数**。

5.1.2 hash 表大小的确定

Hash 表大小的确定也非常关键，如果 *Hash* 表的空间远远大于最后实际存储的记录个数，则造成了很大的空间浪费，如果选取小了的话，则容易造成冲突。

->固定空间： 在实际情况中，一般需要根据最终记录**存储个数**和**关键字的分布特点**来确定 Hash 表的大小。

->ReHash： 还有一种情况时可能**事先不知道最终需要存储的记录个数，则需要动态维护 Hash 表的容量**，此时可能需要**重新计算** Hash 地址.

5.1.3 冲突的解决

1. **开放定址法**

   当一个关键字和另一个关键字发生冲突时，使用某种**探测技术在 Hash 表中形成一个探测序列**，然后沿着这个探测序列依次查找下去，当碰到一个空的单元时，则插入其中。

   比较常用的探测方法有线性探测法，比如有一组关键字{12，13，25，23，38，34，6，84，91}，Hash 表长为 14，Hash 函数为address(key)=key%11，当插入12，13，25 时可以直接插入，而当插入23 时，地址 1 被占用了，因此沿着地址 1 依次往下探测 (探测步长可以根据情况而定)，直到探测到地址 4，发现为空，则将 23 插入其中

2. **链地址法**

   采用数组和链表相结合的办法，将 Hash 地址相同的记录存储在一张线性表中，而每张表的表头的序号即为计算得到的 Hash 地址

http://www.cnblogs.com/dolphin0520/archive/2012/09/28/2700000.html

5.2 Longest Substring Without Repeating Characters

问题

Given a string, find the length of the longest substring without repeating characters.

Examples Given "abcabcbb", the answer is "abc", which the length is 3.

Given "bbbbb", the answer is "b", with the length of 1.

Given "pwwkew", the answer is "wke", with the length of 3. Note that the answer must be a substring, "pwke" is a subsequence and not a substring.

分析

如果发现重复：第一个指针指向以查找的序列的最后一个该字符，另一个指向当前的，求取最长的中间长度。

解答

```
class Solution {
public:
int lengthOfLongestSubstring(string s)
{
    int maxLen = 0,repeatCharacter_First_index = -1;
    unordered_multimap<char,int> subString;
    for (int i = 0; i < s.length(); ++i)
    {
        unordered_multimap<char,int>::const_iterator it = subString.find(s[i]);
        if (it != subString.end())
        {
            int max_second = it->second;
            auto range = subString.equal_range(s[i]);
            for_each(range.first,range.second,[&max_second](unordered_multimap<char,
                int>::value_type& x){max_second = max(max_second, x.second); });
            repeatCharacter_First_index = max(repeatCharacter_First_index, max_second);
                 // [...First_max]...Second_not insert
        }
        subString.insert(pair<char, int>(s[i], i));
        maxLen = max(maxLen, i - repeatCharacter_First_index );
    }
    return maxLen;
}
};
```

5.3

Examples

5.3.1 分析

5.3.2 解答

5.4

Examples

5.4.1 分析

5.4.2 解答

5.5

Examples

5.5.1 分析

5.5.2 解答

5.6

Examples

5.6.1 分析

5.6.2 解答

5.7

Examples

5.7.1 分析

5.7.2 解答

5.8

Examples

5.8.1 分析

5.8.2 解答

5.9

Examples

5.9.1 分析

5.9.2 解答

5.10

Examples

5.10.1 分析

5.10.2 解答

第六章　Linked List- 27

http://blog.csdn.net/luckyxiaoqiang/article/details/7393134/

6.1　将单链表反转

6.1.1　解答

6.2　查找单链表中的倒数第 K 个结点（k > 0）

双指针

6.2.1　解答

6.3　查找单链表的中间结点

双指针

6.3.1　解答

6.4　从尾到头打印单链表

Examples

6.4.1　分析

6.4.2　解答

6.5　已知两个单链表 pHead1 和 pHead2 各自有序，把它们合并成一个链表依然有序

Examples

6.5.1 分析

6.5.2 解答

6.6 判断一个单链表中是否有环

Examples

6.6.1 分析

6.6.2 解答

6.7 判断两个单链表是否相交

6.7.1 分析

http://blog.csdn.net/jiqiren007/article/details/6572685

方法 1. 将第一个链表的尾部指向第二个链表，如果相交则必产生环，且环的头结点为第二个链表的头结点，没有相交则不存在环。

方法 2. 如果两个链表相交，则相交后的节点都一样，所以结尾节点也将一样，我们以此遍历第一个第二个链表，取出其尾部地址，判断是否相等即可。

6.7.2 解答

6.8 求两个单链表相交的第一个节点

Examples

6.8.1 分析

6.8.2 解答

6.9 已知一个单链表中存在环，求进入环中的第一个节点

Examples

6.9.1 分析

6.9.2 解答

6.10 给出一单链表头指针 pHead 和一节点指针 pToBeDeleted，O(1) 时间复杂度删除节点 pToBeDeleted

6.10.1 分析

对于删除节点，我们普通的思路就是让该节点的前一个节点指向该节点的下一个节点，这种情况需要遍历找到该节点的前一个节点，**时间复杂度为 O(n)**。对于链表，**链表中的每个节点结构都是一样的，所以我们可以把该节点的下一个节点的数据复制到该节点，然后删除下一个节点即可**。要注意最后一个节点的情况，这个时候只能用常见的方法来操作，先找到前一个节点，但总体的平均时间复杂度还是 O(1)。

6.10.2 解答

第七章　Math- 58

7.1

Examples

7.1.1　分析

7.1.2　解答

7.2

Examples

7.2.1　分析

7.2.2　解答

7.3

Examples

7.3.1　分析

7.3.2　解答

7.4

Examples

7.4.1 分析

7.4.2 解答

7.5

Examples

7.5.1 分析

7.5.2 解答

7.6

Examples

7.6.1 分析

7.6.2 解答

7.7

Examples

7.7.1 分析

7.7.2 解答

7.8

Examples

7.8.1 分析

7.8.2 解答

7.9

Examples

7.9.1 分析

7.9.2 解答

7.10

Examples

7.10.1 分析

7.10.2 解答

第八章　Two Pointers- 33

8.1

Examples

8.1.1　分析

8.1.2　解答

8.2

Examples

8.2.1　分析

8.2.2　解答

8.3

Examples

8.3.1　分析

8.3.2　解答

8.4

Examples

8.4.1 分析

8.4.2 解答

8.5

Examples

8.5.1 分析

8.5.2 解答

8.6

Examples

8.6.1 分析

8.6.2 解答

8.7

Examples

8.7.1 分析

8.7.2 解答

8.8

Examples

8.8.1 分析

8.8.2 解答

8.9

Examples

8.9.1 分析

8.9.2 解答

8.10

Examples

8.10.1 分析

8.10.2 解答

第九章　String- 55

9.1

Examples

9.1.1　分析

9.1.2　解答

9.2

Examples

9.2.1　分析

9.2.2　解答

9.3

Examples

9.3.1　分析

9.3.2　解答

9.4

Examples

9.4.1 分析

9.4.2 解答

9.5

Examples

9.5.1 分析

9.5.2 解答

9.6

Examples

9.6.1 分析

9.6.2 解答

9.7

Examples

9.7.1 分析

9.7.2 解答

9.8

Examples

9.8.1 分析

9.8.2 解答

9.9

Examples

9.9.1 分析

9.9.2 解答

9.10

Examples

9.10.1 分析

9.10.2 解答

9.11

Examples

9.11.1 分析

9.11.2 解答

9.12

Examples

9.12.1 分析

9.12.2 解答

9.13

Examples

9.13.1 分析

9.13.2 解答

9.14

Examples

9.14.1 分析

9.14.2 解答

9.15

Examples

9.15.1 分析

9.15.2 解答

9.16 Divide and Conquer- 12

9.17

Examples

9.17.1 分析

9.17.2 解答

9.18

Examples

9.18.1 分析

9.18.2 解答

9.19

Examples

9.19.1 分析

9.19.2 解答

9.20

Examples

9.20.1 分析

9.20.2 解答

9.21

Examples

9.21.1 分析

9.21.2 解答

9.22

Examples

9.22.1 分析

9.22.2 解答

9.23

Examples

9.23.1 分析

9.23.2 解答

9.24

Examples

9.24.1 分析

9.24.2 解答

9.25

Examples

9.25.1 分析

9.25.2 解答

9.26

Examples

9.26.1 分析

9.26.2 解答

第十章　Binary Search- 38

10.1

Examples

10.1.1　分析

10.1.2　解答

10.2

Examples

10.2.1　分析

10.2.2　解答

10.3

Examples

10.3.1　分析

10.3.2　解答

10.4

Examples

10.4.1 分析

10.4.2 解答

10.5

Examples

10.5.1 分析

10.5.2 解答

10.6

Examples

10.6.1 分析

10.6.2 解答

10.7

Examples

10.7.1 分析

10.7.2 解答

10.8

Examples

10.8.1 分析

10.8.2 解答

10.9

Examples

10.9.1 分析

10.9.2 解答

10.10

Examples

10.10.1 分析

10.10.2 解答

第十一章 Dynamic Programming- 66

http://blog.csdn.net/eagle_or_snail/article/details/50987044
http://m.blog.csdn.net/cqkxboy168/article/details/40465389

11.1 Range Sum Query - Immutable

Given an integer array nums, find the sum of the elements between indices i and j (i ≤ j), inclusive.

https://leetcode.com/problems/range-sum-query-immutable/description/

Examples ->

```
Given nums = [-2, 0, 3, -5, 2, -1]

sumRange(0, 2) -> 1
sumRange(2, 5) -> -1
sumRange(0, 5) -> -3
```

11.1.1 分析

1- for 循环 谁都可以想出来的方法。

2- 真正的考验 这道题的难点就在于**是否能想到来用建立累计直方图的思想来建立一个累计和的数组 dp**

1. 使用一个数组 dp[i], 记录 0 到第 i 个数的和
2. 求 i 到 j 之间的和时，输出 dp[j]-dp[i]+num[i] 即可

11.1.2 解答

```
class NumArray {
public:
    vector<int> dp;
    vector<int> num;
    NumArray(vector<int> nums) {
```

```
        int n=nums.size();
        dp=vector<int>(n,0);
        num=nums;
        for(int i=0;i<n;i++)
        {
            if(i>0)
                dp[i]=dp[i-1]+nums[i];
            else
                dp[0]=nums[0];
        }
    }

    int sumRange(int i, int j) {
        return dp[j]-dp[i]+num[i];
    }
};
```

11.2 Maximum Subarray

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

https://leetcode.com/problems/maximum-subarray/description/

Examples ->

```
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.
```

11.2.1 分析

子数组的和大于 0 则递加，否则令子数组的开始为当前位置，并令当前和为当前值，整个过程中的最大值既是 maxSum;

1. **寻找最优子结构：**如果和要递增，则前面的数不能小于 0，求最大则保存过程中的最大值即可
2. **确定边界值**

11.2.2 解答

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int sum = 0;
```

```
        int lagest = nums[0];
        for(auto d:nums)
        {
            if(sum > 0)
            {
                sum += d;
                if(sum > lagest) lagest = sum;
            }
            else
            {
                sum = d;
                if(sum > lagest) lagest = sum;
            }
        }
        return lagest;
    }
};

// DP 公式法
class Solution {
public:
    int maxSubArray(int A[], int n) {
        int ans=A[0],i,j,sum=0;
        for(i=0;i<n;i++){
            sum+=A[i];
            ans=max(sum,ans);
            sum=max(sum,0);
        }
        return ans;
    }
};
```

11.3 Triangle

Given a triangle, find the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below.

`https://leetcode.com/problems/triangle/description/`

Examples ->

```
For example, 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).
```

11.3.1 分析

如何确定是否用动态规划：**各小部分是否有关联，如果有关联，即存在父问题的解依赖于子问题的解**, 和问题肯定依赖与上或下一个数。

1. **寻找最优子结构：**一般是从最后一步反推，这个的规律就是从后推的，从前推问题规模越来越大，从后推问题规模越来越小。

2. **确定边界值**

11.3.2 解答

```
class Solution {
public:
int minimumTotal(vector<vector<int>>& triangle) {
       for(int i = triangle.size()-2; i >= 0; --i)
       {
           for(int j = 0; j < i+1; ++j)
               triangle[i][j] += std::min(triangle[i+1][j], triangle[i+1][j+1]);
       }
       return triangle[0][0];
   }
};
```

11.4 2 Keys Keyboard

Initially on a notepad only one character 'A' is present. You can perform two operations on this notepad for each step:

Copy All: You can copy all the characters present on the notepad (partial copy is not allowed).

Paste: You can paste the characters which are copied last time.

Given a number n. You have to get exactly n 'A' on the notepad by performing the minimum number of steps permitted. Output the minimum number of steps to get n 'A'.

Examples ->

```
Input: 3
Output: 3
Explanation:
Intitally, we have one character 'A'.
In step 1, we use Copy All operation.
```

```
In step 2, we use Paste operation to get 'AA'.
In step 3, we use Paste operation to get 'AAA'.
```

11.4.1 分析

当然，从正面走的话，规模越来越大，从后推则简化了问题。这个问题当然想到的是分治思路，类似与 pow 的分治。

1. **寻找最优子结构：**刚开始能想到的是奇数和偶数，当测试到 25 时，就会发现因子不止 2、3、5。这个过程就是思维发散的假设过程。
   - `if n % 2 == 0, then f(n) = f(n/2) + 2`
   - `if n % 3 == 0, then f(n) = f(n/3) + 3`
   - `if n % 5 == 0, then f(n) = f(n/5) + 5...`
2. **确定边界值**

11.4.2 解答

```
class Solution {
    public:
    int step = 0;
    int minSteps(int n) {
        if(n == 1) return 0; //return Copy
        if(n % 2== 0)
        {
            step += 2;
            minSteps(n/=2);
        }
        else
        {
            bool test = false;
            for(int j = 3; j < n/2; j+=2)
            {
                if(n%j == 0)
                {
                    step += j;
                    test = true;
                    minSteps(n/=j);
                    break;
                }
            }
            if(!test) step += n;
        }
```

```
            return step;
        }
    };

    // 简化代码
    class Solution {
        public:
        int minSteps(int n) {
            if (n == 1) return 0;
            for (int i = 2; i < n; i++)
                if (n % i == 0) return i + minSteps(n / i);
            return n;
        }
    };
```

11.5 Palindromic Substrings

Given a string, your task is to count how many palindromic substrings in this string.

The substrings with different start indexes or end indexes are counted as different substrings even they consist of same characters.

https://leetcode.com/problems/palindromic-substrings/description/

Examples

```
Input: "abc"
Output: 3
Explanation: Three palindromic strings: "a", "b", "c".

Input: "aaa"
Output: 6
Explanation: Six palindromic strings: "a", "a", "a", "aa", "aa", "aaa".
```

11.5.1 分析

这个题使用 dp 思想时不能局限在一维 dp 数组上，使用二维可能问题就迎刃而解。

$$\texttt{dp[i][j]=1}\begin{cases} i=j \\ \texttt{s[i]==s[j]} \quad j-i<2\texttt{(start+1 >= end-1)} \\ \texttt{s[i]==s[j]} \quad \texttt{dp[i+1][j-1]==1} \end{cases}$$

1. **寻找最优子结构：定义** d[i][j]：若从 i 到 j 的字符串为回文，则为真（1），否则为假（0），**那么 d[i][j] 为真的前提是：**头尾两个字符串相同并且去掉头尾以后的字串也是回文（即 d[i+1][j-1] 为真），这里面要注意特殊情况，即：去掉头尾以后为空串，所以如果`j-i<3`，并且头尾相等，也是回文的。

2. **确定边界值**

11.5.2 解答

```
class Solution {
    public:
    int countSubstrings(string s) {
        int n = s.size(), count = 0;
        vector<vector<int>> dp(n, vector<int> (n));
        for ( int end = 0; end < n; ++end ) {
            dp[end][end] = 1;
            ++count;
            for ( int start = 0; start < end; ++start ) {
                if ( s[start] == s[end] && (start+1 >= end-1 || dp[start+1][end-1])) {
                    dp[start][end] = 1;
                    ++count;
                }
            }
        }
        return count;
    }
};
```

11.6 Longest Increasing Subsequence

Given an unsorted array of integers, find the length of longest increasing subsequence.

https://leetcode.com/problems/longest-increasing-subsequence/description/

Examples ->

```
Given [10, 9, 2, 5, 3, 7, 101, 18],
The longest increasing subsequence is [2, 3, 7, 101], therefore the length is 4. Note
    that there may be more than one LIS combination, it is only necessary for you to
    return the length.

Your algorithm should run in O(n2) complexity.
```

11.6.1 分析

首先，如果想让递增序列增加，那么首先这个值必须大于前面的某一个，但是大于哪个就是决策的关键。如何选择会让当前结果最大是本题的关键。

1. **寻找最优子结构:**dp[当前] = max{自身, dp[之前某一个]+1},**前提是** data[当前] > data[之前某一个]
2. **确定边界值**

11.6.2 解答

```
int lengthOfLIS(vector<int>& nums) {
    const int size = nums.size();
    if (size == 0) { return 0; }
    vector<int> dp(size, 1);
    int res = 1;
    for (int i = 1; i < size; ++i) {
        for (int j = 0; j < i; ++j) {
            if (nums[j] < nums[i]) {
                dp[i] = max(dp[i], dp[j]+1);
            }
        }
        res = max (res, dp[i]);
    }
    return res;
}
```

11.7 0-1 背包

有 N 件物品和一个容量为 V 的背包。第 i 件物品的费用是 c[i]，价值是 w[i]。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且**价值总和最大**。

这个问题的特点是：**每种物品只有一件，可以选择放或者不放**。因为这里的物品一般指花瓶、玉器什么的，要么拿、要么不拿，只有 0 和 1 两种状态，所以也叫 0-1 背包。0-1 背包虽然简单，却很重要，是“万法之源”，是其他几类问题的基础。

http://blog.csdn.net/wumuzi520/article/details/7014559

Examples

11.7.1 分析

初学者有时会认为，0-1 背包可以这样求解：**计算每个物品的 Vi/Wi**，*然后依据 Vi/Wi 的值，对所有的物品从大到小进行排序*。其实这种贪心方法是错误的。如下表，有三件物品，背包的最大负重量是 50，求可以取得的最大价值。

Fig 11.1: 0-1 贪心错误示例

一般会想到的是放与不放的二叉树 $O(2^n)$ 方法。其实，0-1 背包是 DP 的一个经典实例，可以用动态规划求解。

DP 求解过程 ->

最优子结构：对于**前 i 件物品**，背包**剩余容量为 j** 时，所**取得的最大价值**（此时称为状态 3）**只依赖于两个状态**。

1. 前i-1 件物品，背包剩余容量为j。在该状态下，只要**不选第 i 个物品**，就可以转换到状态 3

2. 前i-1 件物品，背包剩余容量为j-w[i]。在该状态下，**选第 i 个物品**，也可以转换到状态 3。

因为，这里**要求最大价值**，所以只要**从状态 1 和状态 2 中选择最大价值较大的一个即可**。

递归定义最优解的值：用c(i,j) 表示前 i 件物品，背包剩余容量为 j 时，所**取得的最大价值**。

$$c[i\ ,\ w] = \begin{cases} 0 & i = 0或w = 0 \\ c[i-1,\ w] & w_i > w \\ max\{c[i-1,\ w-w\_i]+v\_i,\ c[i-1,w]\} & i > 0且w_i <= w \end{cases}$$

11.7.2 解答

http://blog.csdn.net/yeepom/article/details/8712224

```
#include <iostream>
using namespace std;

/***
c[i][w]表示背包容量为w时，i个物品导致的最优解的总价值，大小为(n+1)*(w+1)
v[i]表示第i个物品的价值，大小为n
w[i]表示第i个物品的重量，大小为n
***/

void DP(int n, int W, int c[][18], int *v, int *wei)
{
    memset(*c, 0, (W+1)*sizeof(int));
    for (int i = 1; i <= n; i++)
    {
        c[i][0] = 0;
        for (int w = 1; w <= W; w++)
        {
            if (wei[i-1] > w) //此处比较是关键
            {
                c[i][w] = c[i-1][w];
            }
            else
            {
                int temp = c[i-1][w-wei[i-1]] + v[i-1]; //注意wei和v数组中的第i个应该为
                    wei[i-1]和v[i-1]
                if (c[i-1][w] > temp)
                {
                    c[i][w] = c[i-1][w];
```

```
                }
                else
                c[i][w] = temp;
            }
        }
    }
}

int main()
{
    int n = 5;
    int W = 17;
    int w[] = {3, 4, 7, 8, 9};
    int v[] = {4, 5, 10, 11, 13};
    int c[6][18] = {0};
    DP(n, W, c, v, w);
    cout<<c[5][17]<<endl;
}
```

11.8 完全背包

http://blog.csdn.net/wumuzi520/article/details/7014830

Examples

11.8.1 分析

11.8.2 解答

11.9 石子合并

有 N 堆石子，现要将石子有序的合并成一堆，规定如下：每次只能**移动相邻的 2 堆**石子合并，合并花费为新合成的一堆石子的数量。求将这 N 堆石子合并成一堆的总花费最小（或最大）。

11.9.1 分析

设dp[i][j] 表示第 i 到第 j 堆石子合并的最优值，sum[i][j] 表示第 i 到第 j 堆石子的总数量

$$\texttt{d[i][j]} = \begin{cases} 0 & i = j \\ \texttt{min(dp[i][k], dp[k+1][j]) + sum[i][j]} & i \neq j \end{cases}$$

11.9.2 实现

```
#include <iostream>
using namespace std;
#define M 101
#define INF 1000000000
int n,f[M][M],sum[M][M],stone[M];
int main()
{
    int i,j,k,t;
    cin>>n;
    for(i=1;i<=n;i++)
        scanf("%d",&stone[i]);

    for(i=1;i<=n;i++)
    {
        f[i][i]=0;
        sum[i][i]=stone[i];
        for(j=i+1;j<=n;j++)
            sum[i][j]=sum[i][j-1]+stone[j];
    }

    for(int len=2;len<=n;len++)//归并的石子长度
    {
        for(i=1;i<=n-len+1;i++)//i为起点，j为终点
        {
            j=i+len-1;
            f[i][j]=INF;
            for(k=i;k<=j-1;k++)
            {
                if(f[i][j]>f[i][k]+f[k+1][j]+sum[i][j])
                    f[i][j]=f[i][k]+f[k+1][j]+sum[i][j];
            }
        }
    }
    printf("%d/n",f[1][n]);
    return 0;
}
```

11.10 Maximum Length of Pair Chain

You are given n pairs of numbers. In every pair, the first number is always smaller than the second number.

Now, we define `a pair (c, d)` can follow another `pair (a, b)` if and `only if b < c`. Chain of pairs can be formed in this fashion.

Given a set of pairs, find the length longest chain which can be formed. You needn't use up all the given pairs. You can select pairs in any order.

https://leetcode.com/problems/maximum-length-of-pair-chain/description/

Examples ->

```
Input: [[1,2], [2,3], [3,4]]
Output: 2
Explanation: The longest chain is [1,2] -> [3,4]
```

11.10.1 分析

1. **寻找最优子结构：**dp[当前] = max{自身，dp[之前某一个]+1}，**前提是** data[当前] > data[之前某一个], 区间排序

2. **确定边界值**

11.10.2 解答

```
class Solution {
    public:
    int findLongestChain(vector<vector<int>>& pairs) {
        vector<int> dp(pairs.size(), 1);
        sort(pairs.begin(), pairs.end(), cmp);
        for (int i = 1; i < pairs.size(); ++i)
        {
            for (int j = 0; j < i; ++j)
            {
                if (pairs[i][0] > pairs[j][1])
                {
                    dp[i] = std::max(dp[i], dp[j] + 1);
                }
            }
        }

        int big = 0;
        for (int i = 0; i < dp.size(); ++i)
        {
            if (dp[i] > big)
            big = dp[i];
        }
        return big;
    }

    private:
    static bool cmp(vector<int>& a, vector<int>&b) {
```

```
        return a[1] < b[1] || a[1] == b[1] && a[0] < b[0];
    }
};
```

11.11 合唱团-2016 网易编程

http://blog.csdn.net/hui1140621618/article/details/65437387

有 n 个学生站成一排，每个学生有一个能力值，牛牛想从这 n 个学生中按照顺序选取 k 名学生，要求相邻两个学生的位置编号的差不超过 d，使得这 k 个学生的能力值的乘积最大，你能返回最大的乘积吗？

每个输入包含 1 个测试用例。每个测试数据的第一行包含一个整数 n (1 <= n <= 50)，表示学生的个数，接下来的一行，包含 n 个整数，按顺序表示每个学生的能力值 ai（-50 <= ai <= 50）。接下来的一行包含两个整数，k 和 d (1 <= k <= 10, 1 <= d <= 50)。

Examples

```
// 输入
3
7 4 7
2 50
// 输出
49
```

11.11.1 分析

http://blog.csdn.net/lengxiao1993/article/details/52305420

采用动态规划。设 Maxval[i][j] 表示以第 i 个人为最后一个人，一共选取了 j+1 个人时的最大乘积。

同理，Minval[i][j] 表示同样状态下的最小乘积（由于数据中存在负数，负数乘上某个极大的负数反而会变成正的极大值，因而需要同时记录最小值）。

Maxval[i][j] 很显然与 Maxval[i][j-1] 相关，可以理解为 Maxval[i][j] 由两部分组成，一部分是自身作为待选值，另一部分是 Maxval[i][j-1] 乘上一个人后得到的值，然后取它们的极大值，由此可以得到状态转移方程如下：

$Maxval[i][j] = max(Maxval[i][j], Maxval[i][j-1] * a[i], Minval[i][j-1] * a[i])$

$Minval[i][j] = max(Minval[i][j], Maxval[i][j-1] * a[i], Minval[i][j-1] * a[i])$

11.11.2 解答

```
public class Choir {

    public static void main(String[] args){

        Scanner s = new Scanner(System.in);
```

```
        while(s.hasNextInt()){
            int n = s.nextInt(); //学生人数
            int[] ability = new int[n];
            for(int i = 0; i < n; i++){
                ability[i] = s.nextInt();
            }
            int k = s.nextInt();
            int d = s.nextInt();

            //maxProduct[i][j]表示以第i个人为结尾，合唱团的人数为j+1时，合唱团最大的能力乘
                积
            long[][] maxProduct = new long[n][k];
            //minProduct[i][j]表示以第i个人为结尾，合唱团的人数为j+1时，合唱团最小的能力乘
                积
            long[][] minProduct = new long[n][k];

            //合唱团中只有一个人
            for(int i = 0; i < n; i++){
                maxProduct[i][0] = ability[i];
                minProduct[i][0] = ability[i];
            }

            long max = Long.MIN_VALUE;
            for(int i = 0; i < n; i++){
                for(int j = 1; j < k; j++){
                    for(int p = i-1; p >= Math.max(i-d,0); p--){
                        maxProduct[i][j] = Math.max(maxProduct[i][j],
                        maxProduct[p][j-1]*ability[i]);
                        maxProduct[i][j] = Math.max(maxProduct[i][j],
                        minProduct[p][j-1]*ability[i]);
                        minProduct[i][j] = Math.min(minProduct[i][j],
                        minProduct[p][j-1]*ability[i]);
                        minProduct[i][j] = Math.min(minProduct[i][j],
                        maxProduct[p][j-1]*ability[i]);
                    }
                }
                max = Math.max(max, maxProduct[i][k-1]);
            }

            System.out.println(max);
        }
    }

}// end of class
```

11.12 Palindrome Partitioning

11.13 Palindrome Partitioning II

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.

Examples ->

```
For example, given s = "aab",
Return 1 since the palindrome partitioning ["aa","b"] could be produced using 1 cut.
```

11.13.1 分析

http://blog.csdn.net/yutianzuijin/article/details/16850031

11.13.2 解答

11.14 Maximal Rectangle

Given a 2D binary matrix filled with 0’s and 1’s, find the largest rectangle containing all ones and return its area.

Examples

11.14.1 分析

11.14.2 解答

11.15 Interleaving String

Given s1, s2, s3, find whether s3 is formed by the interleaving of s1 and s2.

Examples ->

```
Given:
s1 = "aabcc",
s2 = "dbbca",
When s3 = "aadbbcbcac", return true.
When s3 = "aadbbbaccc", return false.
```

11.15.1 分析

11.15.2 解答

11.16 Scramble String(混杂字符串)

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”

Examples

11.16.1 分析

11.16.2 解答

11.17 Minimum Path Sum

Given a m 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. Note: You can only move either down or right at any point in time

Examples

11.17.1 分析

11.17.2 解答

11.18 Edit Distance

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:

- Insert a character
- Delete a character
- Replace a character

Examples

11.18.1 分析

11.18.2 解答

11.19 Decode Ways

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. For example, Given encoded message "12", it could be decoded as "AB" (1 2) or "L" (12). The number of ways decoding "12" is 2

Examples

11.19.1 分析

11.19.2 解答

11.20 Distinct Subsequences

11.21 Word Break

第十二章　Backtracking- 34

12.1

Examples

12.1.1　分析

12.1.2　解答

12.2

Examples

12.2.1　分析

12.2.2　解答

12.3

Examples

12.3.1　分析

12.3.2　解答

12.4

Examples

12.4.1 分析

12.4.2 解答

12.5

Examples

12.5.1 分析

12.5.2 解答

12.6

Examples

12.6.1 分析

12.6.2 解答

12.7

Examples

12.7.1 分析

12.7.2 解答

第十三章　Stack- 25

13.1

Examples

13.1.1　分析

13.1.2　解答

13.2

Examples

13.2.1　分析

13.2.2　解答

13.3

Examples

13.3.1　分析

13.3.2　解答

13.4

Examples

13.4.1 分析

13.4.2 解答

13.5

Examples

13.5.1 分析

13.5.2 解答

第十四章　Heap- 15

14.1

Examples

14.1.1　分析

14.1.2　解答

14.2

Examples

14.2.1　分析

14.2.2　解答

14.3

Examples

14.3.1　分析

14.3.2　解答

14.4

Examples

14.4.1 分析

14.4.2 解答

14.5

Examples

14.5.1 分析

14.5.2 解答

14.6

Examples

14.6.1 分析

14.6.2 解答

14.7

Examples

14.7.1 分析

14.7.2 解答

14.8

Examples

14.8.1 分析

14.8.2 解答

14.9

Examples

14.9.1 分析

14.9.2 解答

14.10

Examples

14.10.1 分析

14.10.2 解答

第十五章　Greedy- 19

15.1

Examples

15.1.1　分析

15.1.2　解答

15.2

Examples

15.2.1　分析

15.2.2　解答

15.3

Examples

15.3.1　分析

15.3.2　解答

15.4

Examples

15.4.1 分析

15.4.2 解答

15.5

Examples

15.5.1 分析

15.5.2 解答

15.6

Examples

15.6.1 分析

15.6.2 解答

15.7

Examples

15.7.1 分析

15.7.2 解答

15.8

Examples

15.8.1 分析

15.8.2 解答

15.9

Examples

15.9.1 分析

15.9.2 解答

15.10

Examples

15.10.1 分析

15.10.2 解答

第十六章　Sort- 16

16.1　冒泡

冒泡排序的基本思想是，对**相邻的元素**进行**两两比较**，顺序相反则进行交换，这样，**每一趟会将最小或最大的元素“浮”到顶端**，最终达到完全有序

16.1.1　解答

https://github.com/ctzhenghua/C-NetworkPractice-Code/blob/master/DataStructure/Sort/PopSort.cc

16.2 选择排序

选择排序的思想非常直接，不是要排序么？那好，我就从所有序列中**先找到最小的，然后放到第一个位置。之后再看剩余元素中最小的，放到第二个位置……**以此类推，就可以完成整个的排序工作了。可以很清楚的发现，选择排序是固定位置，找元素。相比于插入排序的固定元素找位置，是两种思维方式。不过条条大路通罗马，两者的目的是一样的。

16.2.1 解答

https://github.com/ctzhenghua/C-NetworkPractice-Code/blob/master/DataStructure/Sort/SelectSort.cc

16.3 快速排序

快速排序是**找出一个元素**（理论上可以随便找一个）**作为基准 (pivot)**, 然后对数组进行分区操作, **使基准左边元素的值都不大于基准值, 基准右边的元素值都不小于基准值**，如此作为基准的元素调整到排序后的正确位置。递归快速排序，将其他 n-1 个元素也调整到排序后的正确位置。最后每个元素都是在排序后的正确位置，排序完成。所以快速排序算法的核心算法是分区操作，即如何调整基准的位置以及调整返回基准的最终位置以便分治递归。

16.3.1 核心功能分责

partition()

- 在给定序列 (low，high)，找切割位置，既pivot 最终位置。
- 填充pivot 位置

sort() 切分**序列**，这是改非递归版本的核心，既将填充功能分出，然后用栈模拟出序列的变化。

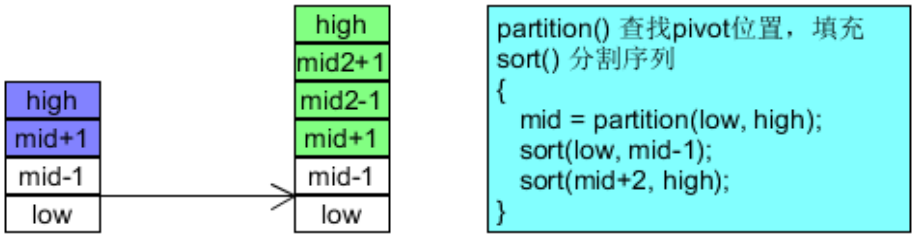


Fig 16.1: 非递归思路剖析

16.3.2 解答

https://github.com/ctzhenghua/C-NetworkPractice-Code/DataStructure/Sort/QuickSort.cc

非递归版本：http://www.cnblogs.com/zhangchaoyang/articles/2234815.html

```
/**把数组分为两部分，轴pivot左边的部分都小于轴右边的部分**/
template <typename Comparable>
int partition(vector<Comparable> &vec,int low,int high){
    Comparable pivot=vec[low]; //任选元素作为轴，这里选首元素
    while(low<high){
        while(low<high && vec[high]>=pivot)
            high--;
        vec[low]=vec[high];
        while(low<high && vec[low]<=pivot)
            low++;
        vec[high]=vec[low];
    }
```

```
    //此时low==high
    vec[low]=pivot;
    return low;
}
//其实就是用栈保存每一个待排序子串的首尾元素下标，下一次while循环时取出这个范围，对这段子序列
    进行partition操作
void quicksort2(vector<Comparable> &vec,int low,int high){
    stack<int> st;
    if(low>=high) return;
    int mid=partition(vec,low,high);
    if(low<mid-1){
        st.push(low);
        st.push(mid-1);
    }
    if(mid+1<high){
        st.push(mid+1);
        st.push(high);
    }
    while(!st.empty()){
        int q=st.top();
        st.pop();
        int p=st.top();
        st.pop();
        mid=partition(vec,p,q);
        if(p<mid-1){
            st.push(p);
            st.push(mid-1);
        }
        if(mid+1<q){
            st.push(mid+1);
            st.push(q);
        }
    }
}
```

16.4 插入排序

插入即表示将一个新的数据插入到一个有序数组中，并继续保持有序。例如有一个长度为 N 的无序数组，进行 N-1 次的插入即能完成排序；第一次，数组第 1 个数认为是有序的数组，将数组第二个元素插入仅有 1 个有序的数组中；第二次，数组前两个元素组成有序的数组，将数组第三个元素插入由两个元素构成的有序数组中..... 第 N-1 次，数组前 N-1 个元素组成有序的数组，将数组的第 N 个元素插入由 N-1 个元素构成的有序数组中，则完成了整个插入排序。

16.4.1 解答

16.5 归并排序

https://www.cnblogs.com/skywang12345/p/3602369.html

归并排序的思想就是把前一段排序，后一段排序，然后再合并两个有序数组。而分到每个数组只有一个元素为止，然后从底开始回溯。

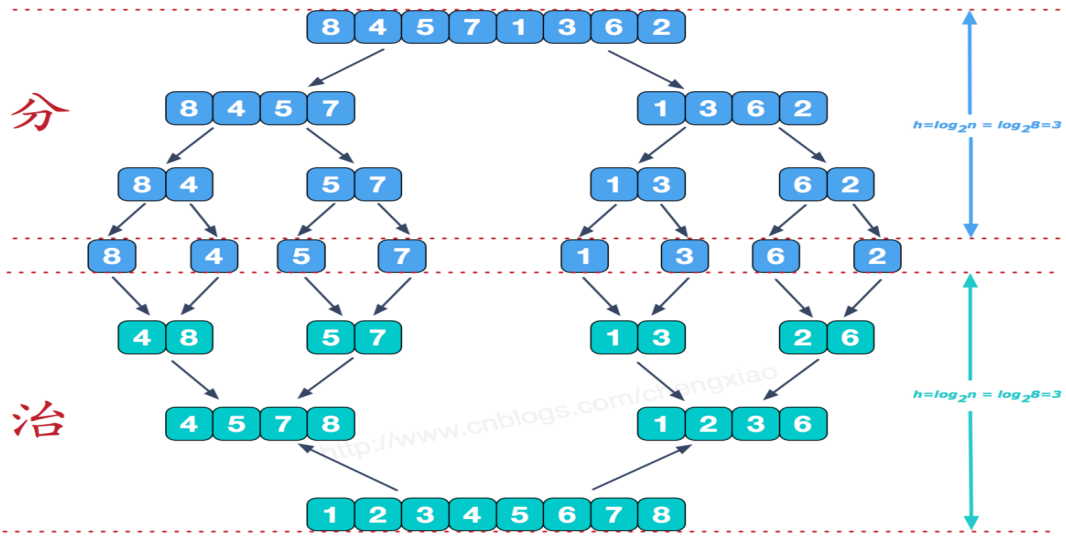


Fig 16.2: 归并排序图例

归并排序其实要做两件事：

1. **“分解”**——将序列每次折半划分。
2. **“合并”**——将划分后的序列段两两合并后排序。

16.5.1 如何合并

在每次合并过程中，都是对两个有序的序列段进行合并 (有序数组合并)。这两个有序序列段分别为 R[low, mid] 和 R[mid+1, high]。先将他们合并到一个局部的暂存数组R2 中，带合并完成后再将R2 复制回R 中。

为了方便描述，我们称 R[low, mid] 第一段，R[mid+1, high] 为第二段。

每次从两个段中取出一个记录进行关键字的比较，将较小者放入 R2 中。最后将各段中余下的部分直接复制到 R2 中。经过这样的过程，R2 已经是一个有序的序列，再将其复制回 R 中，一次合并排序就完成了。

实现 ->

```
void mergearray(int a[], int first, int mid, int last, int temp[])
{
```

```
    int i = first, j = mid + 1;
    int m = mid, n = last;
    int k = 0;

    while (i <= m && j <= n)
    {
        if (a[i] <= a[j])
            temp[k++] = a[i++];
        else
            temp[k++] = a[j++];
    }

    while (i <= m)
        temp[k++] = a[i++];

    while (j <= n)
        temp[k++] = a[j++];

    for (i = 0; i < k; i++)
        a[first + i] = temp[i];
}
```

16.5.2 如何分解

在某趟归并中，折半分解。

实现 ->

```
void mergesort(int a[], int first, int last, int temp[])
{
    if (first < last)
    {
        int mid = (first + last) / 2;
        mergesort(a, first, mid, temp); //左边有序
        mergesort(a, mid + 1, last, temp); //右边有序
        mergearray(a, first, mid, last, temp); //再将二个有序数列合并
    }
}
```

16.5.3 非递归版本

https://www.jianshu.com/p/feeaa4296629

```
#define MAXSIZE 100 //用于要排序数组的最大值

typedef struct //定义一个顺序表结构
```

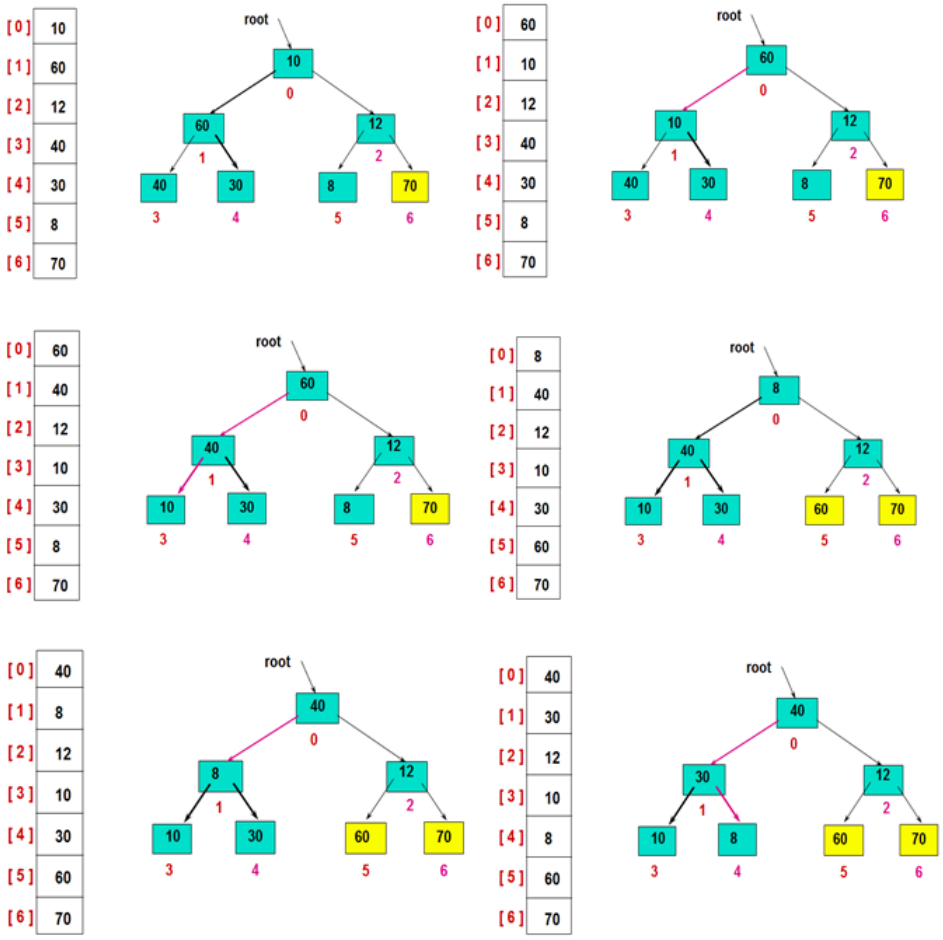
```
{
    int r[MAXSIZE+1]; //用于存储要排序数组，r[0]用作哨兵或者临时变量
    int length; //用于存储顺序表的最大长度
}SqList;

//归并排序最主要实现函数
//将有序的S[low..m]和S[m+1..high]归并到有序的T[low..high]中
void Merge(int *S, int *T, int low, int m, int high)
{//j在[m+1,high]区间递增，k用于目标T的下标递增，l是辅助变量
    int j, k, l;

    for (k = low,j = m+1; low <= m && j <= high; k++) //将S中的元素由小到大归并到T中
    {
        if (S[low] < S[j])
            T[k] = S[low++]; //此处先执行T[k] = S[low],然后low++;
        else
            T[k] = S[j++]; //同理
    }

    //for循环过后，可能有j>high 或者 low>m，故分情况处理
    if (low <= m)
    {
        for (l = 0; l <= m-low; l++)
            T[k+l] = S[low+l]; //将剩余的S[low..m]复制到T中
    }

    if (j <= high)
    {
        for (l = 0; l <= high-j; l++)
            T[k+l] = S[j+l]; //将剩余的S[j..high]复制到T中
    }
}

//将S中相邻长度为s的子系列两两归并到T中
void MergePass(int *S, int *T, int s, int length)
{
    int i = 1; //r[0]用作哨兵或者临时变量
    int j;

    while (i <= length-2*s+1) //length-i+1(未合并元素) >= 2s
    {
        Merge(S, T, i, i+s-1, i+2*s-1);
        i= i+2*s; //自增的下一个元素的起始点
    }
    if (i< length-s+1) //归并最后两个序列
        Merge(S, T, i, i+s-1, length);
    else
```

```
        for (j = i; j <= length; j++)
            T[j] = S[j];
}


void MergeSort(SqList *L)
{
    //申请额外的空间，由于r[0]为哨兵，所以这里长度要+1
    int* TR = (int *)malloc((L->length+1)*sizeof(int));
    int k = 1;/*k用来表示每次k个元素归并*/

    while (k < L->length)
    {
        //k每次乘以2是遵循1->2->4->8->16...的二路归并元素个数的原则
        MergePass(L->r, TR, k, L->length);/*先借助辅助空间归并*/
        k *= 2;
        MergePass(TR, L->r, k, L->length);/*再从辅助空间将排过序的序列归并回原数组*/
        k *= 2;
    }
    free(TR);//释放内存
    TR=NULL;
}


int main()
{
    int i;
    int array[] = {39,80,76,41,13,29,50,78,30,11,100,7,41,86};
    SqList L;
    L.length = sizeof(array)/sizeof(array[0]); //获取数组长度

    for(i=0;i<L.length;i++)
        L.r[i+1]=array[i]; //把数组存入顺序表结构

    MergeSort(&L);

    for(i=0;i<L.length;i++) //输出排序后的数组
        printf("%d␣",L.r[i+1]);

    return 0;
}
```

16.6 堆

http://www.cnblogs.com/mengdd/archive/2012/11/30/2796845.html

- 数组最后总是不需要再调整的
- 调整好后，根与当前剩下未确定的最后一个进行位置交换



16.6.1 解答

http://blog.csdn.net/xiaoxiaoxuewen/article/details/7570621/

16.7 希尔排序

Examples

16.7.1 分析

16.7.2 解答

16.8 基数排序 (桶)

Examples

16.8.1 分析

16.8.2 解答

第十七章　Bit Manipulation- 26

17.1

Examples

17.1.1　分析

17.1.2　解答

17.2

Examples

17.2.1　分析

17.2.2　解答

17.3

Examples

17.3.1　分析

17.3.2　解答

17.4

Examples

17.4.1 分析

17.4.2 解答

17.5

Examples

17.5.1 分析

17.5.2 解答

17.6

Examples

17.6.1 分析

17.6.2 解答

17.7

Examples

17.7.1 分析

17.7.2 解答

17.8

Examples

17.8.1 分析

17.8.2 解答

17.9

Examples

17.9.1 分析

17.9.2 解答

17.10

Examples

17.10.1 分析

17.10.2 解答

第十八章　Tree- 48

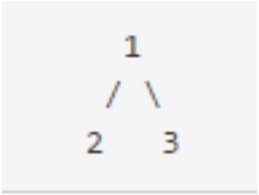
http://blog.csdn.net/luckyxiaoqiang/article/details/7518888

18.1　Binary Tree Maximum Path Sum

Given a binary tree, find the maximum path sum.

For this problem, a path is defined as any sequence of nodes from some starting node to any node in the tree along the parent-child connections. The path must contain at least one node and does not need to go through the root.

Examples　Given the below binary tree,



Return 6.

18.1.1　分析

18.1.2　解答

```
/**
* Definition for a binary tree node.
* struct TreeNode {
* int val;
* TreeNode *left;
* TreeNode *right;
* TreeNode(int x) : val(x), left(NULL), right(NULL) {}
* };
*/
class Solution {
public:
int maxPathSum(TreeNode* root) {
```

```
    }
};
```

18.2 Validate Binary Search Tree

Given a binary tree, determine if it is a valid binary search tree (BST).

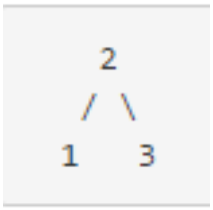
Assume a BST is defined as follows:

1-The left subtree of a node contains only nodes with keys less than the node's key.

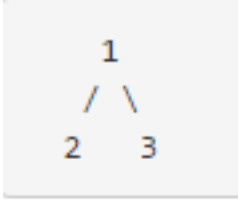
2-The right subtree of a node contains only nodes with keys greater than the node's key.

3-Both the left and right subtrees must also be binary search trees.

Examples Given the below binary tree,



Binary tree [2,1,3], return true.



Binary tree [1,2,3], return false.

18.2.1 分析

18.2.2 解答

18.3 求二叉树中的节点个数

18.3.1 分析

18.3.2 解答

18.4 求二叉树的深度

18.4.1 分析

18.4.2 解答

18.5 前序遍历，中序遍历，后序遍历

18.5.1 已知前序、中序遍历，求后序遍历

前序遍历: GDAFEMHZ
中序遍历: ADEFGHMZ

1. 根据前序遍历的特点，我们知道根结点为 G 根据前序遍历的特点，我们知道根结点为 G

2. 观察中序遍历 ADEFGHMZ。其中 root 节点 G 左侧的 ADEF 必然是 root 的左子树，G 右侧的 HMZ 必然是 root 的右子树

3. 观察左子树 ADEF，左子树的中的根节点必然是大树的 root 的 leftchild。在前序遍历中，大树的 root 的 leftchild 位于 root 之后，所以左子树的根节点为 D

4. 同样的道理，root 的右子树节点 HMZ 中的根节点也可以通过前序遍历求得。在前序遍历中，一定是先把 root 和 root 的所有左子树节点遍历完之后才会遍历右子树，并且遍历的左子树的第一个节点就是左子树的根节点。同理，遍历的右子树的第一个节点就是右子树的根节点。

5. 观察发现，上面的过程是递归的

18.5.2 已知中序和后序遍历，求前序遍历

1. 根据后序遍历的特点，我们知道后序遍历最后一个结点即为根结点，即根结点为 G。

2. 观察中序遍历 ADEFGHMZ。其中 root 节点 G 左侧的 ADEF 必然是 root 的左子树，G 右侧的 HMZ 必然是 root 的右子树。

3. 观察左子树 ADEF，左子树的中的根节点必然是大树的 root 的 leftchild。在前序遍历中，大树的 root 的 leftchild 位于 root 之后，所以左子树的根节点为 D。

4. 同样的道理，root 的右子树节点 HMZ 中的根节点也可以通过前序遍历求得。在前后序遍历中，一定是先把 root 和 root 的所有左子树节点遍历完之后才会遍历右子树，并且遍历的左子树的第一个节点就是左子树的根节点。同理，遍历的右子树的第一个节点就是右子树的根节点。

5. 观察发现，上面的过程是递归的

18.5.3 总结

1. 确定根, 确定左子树，确定右子树。

2. 在左子树中递归。

3. 在右子树中递归。

4. 打印当前根。

结果如图18.1所示：

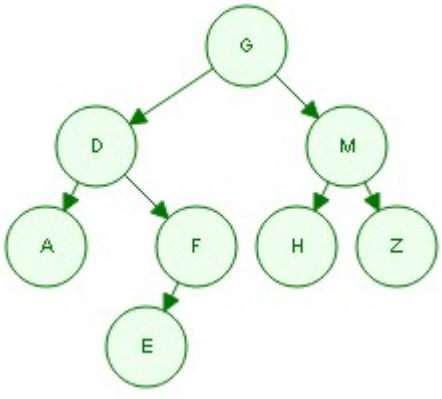


Fig 18.1: 根据先序和中序确定二叉树

18.5.4 代码实现思路

18.6 分层遍历二叉树（按层次从上往下，从左往右）

18.6.1 分析

18.6.2 解答

18.7 将二叉查找树变为有序的双向链表

18.7.1 分析

18.7.2 解答

18.8 求二叉树第 K 层的节点个数

18.8.1 分析

18.8.2 解答

18.9 求二叉树中叶子节点的个数

18.9.1 分析

18.9.2 解答

18.10 判断两棵二叉树是否结构相同

18.10.1 分析

18.10.2 解答

18.11 判断二叉树是不是平衡二叉树

18.11.1 分析

18.11.2 解答

18.12 求二叉树的镜像

18.12.1 分析

18.12.2 解答

18.13 求二叉树中两个节点的最低公共祖先节点

第十九章　Depth-first Search- 41

19.1　部分和问题

给定整数 $a_1, a_2, ..., a_n$, 判断是否可以从中选出若干数，使他们的和恰好为 k

Examples

```
// Test Case 1
    输入
    n = 4
    a = {1, 2, 4, 7}
    k = 13

    输出
    Yes{13 = 2 + 4 + 7}
// Test Case 2
    输入
    n = 4
    a = {1, 2, 4, 7}
    k = 15

    输出
    No
```

19.1.1　分析

当 $n < 20$ 时，运算次数为 100 0000 次，可以接受

19.1.2　解答

```
int a[MAX_N];
int n, k;

bool dfs(int i, sum)
{
    if(i == n) return sum == k;
    if(dfs(i+1, sum)) return true;
    if(dfs(i+1, sum+a[i])) return true;
```

```
        return false;
    }
```

19.2

Examples

19.2.1 分析

19.2.2 解答

19.3

Examples

19.3.1 分析

19.3.2 解答

19.4

Examples

19.4.1 分析

19.4.2 解答

19.5

Examples

19.5.1 分析

19.5.2 解答

19.6

Examples

19.6.1 分析

19.6.2 解答

19.7

Examples

19.7.1 分析

19.7.2 解答

19.8

Examples

19.8.1 分析

19.8.2 解答

19.9

Examples

19.9.1 分析

19.9.2 解答

19.10

Examples

19.10.1 分析

19.10.2 解答

第二十章　Breadth-first Search- 21

20.1

Examples

20.1.1　分析

20.1.2　解答

20.2

Examples

20.2.1　分析

20.2.2　解答

20.3

Examples

20.3.1　分析

20.3.2　解答

20.4

Examples

20.4.1 分析

20.4.2 解答

20.5

Examples

20.5.1 分析

20.5.2 解答

20.6

Examples

20.6.1 分析

20.6.2 解答

20.7

Examples

20.7.1 分析

20.7.2 解答

20.8

Examples

20.8.1 分析

20.8.2 解答

20.9

Examples

20.9.1 分析

20.9.2 解答

20.10

Examples

20.10.1 分析

20.10.2 解答

第二十一章　Union Find- 6

21.1

Examples

21.1.1　分析

21.1.2　解答

21.2

Examples

21.2.1　分析

21.2.2　解答

21.3

Examples

21.3.1　分析

21.3.2　解答

21.4

Examples

21.4.1　分析

21.4.2　解答

第二十二章　Graph- 10

22.1

Examples

22.1.1　分析

22.1.2　解答

22.2

Examples

22.2.1　分析

22.2.2　解答

22.3

Examples

22.3.1　分析

22.3.2　解答

22.4

Examples

22.4.1 分析

22.4.2 解答

22.5

Examples

22.5.1 分析

22.5.2 解答

22.6

Examples

22.6.1 分析

22.6.2 解答

22.7

Examples

22.7.1 分析

22.7.2 解答

22.8

Examples

22.8.1 分析

22.8.2 解答

22.9

Examples

22.9.1 分析

22.9.2 解答

22.10

Examples

22.10.1 分析

22.10.2 解答

第二十三章　Design- 27

23.1

Examples

23.1.1　分析

23.1.2　解答

23.2

Examples

23.2.1　分析

23.2.2　解答

23.3

Examples

23.3.1　分析

23.3.2　解答

23.4

Examples

23.4.1 分析

23.4.2 解答

23.5

Examples

23.5.1 分析

23.5.2 解答

23.6

Examples

23.6.1 分析

23.6.2 解答

23.7

Examples

23.7.1 分析

23.7.2 解答

23.8

Examples

23.8.1 分析

23.8.2 解答

23.9

Examples

23.9.1 分析

23.9.2 解答

23.10

Examples

23.10.1 分析

23.10.2 解答

第二十四章　Topological Sort[拓扑排序]- 5

24.1

Examples

24.1.1　分析

24.1.2　解答

24.2

Examples

24.2.1　分析

24.2.2　解答

24.3

Examples

24.3.1　分析

24.3.2　解答

第二十五章　Trie[前缀-字典树]- 7

25.1

Examples

25.1.1　分析

25.1.2　解答

25.2

Examples

25.2.1　分析

25.2.2　解答

25.3

Examples

25.3.1　分析

25.3.2　解答

第二十六章　Bianry Indexed Tree- 4

26.1

Examples

26.1.1　分析

26.1.2　解答

26.2

Examples

26.2.1　分析

26.2.2　解答

26.3

Examples

26.3.1　分析

26.3.2　解答

第二十七章　Segment Tree- 4

27.1

Examples

27.1.1　分析

27.1.2　解答

27.2

Examples

27.2.1　分析

27.2.2　解答

27.3

Examples

27.3.1　分析

27.3.2　解答

第二十八章　Binary Search Tree- 4

28.1

Examples

28.1.1　分析

28.1.2　解答

28.2

Examples

28.2.1　分析

28.2.2　解答

28.3

Examples

28.3.1　分析

28.3.2　解答

第二十九章　Other Aspects

29.1　Recursion[递归]- 2

29.1.1

Examples

29.1.2　分析

29.1.3　解答

29.2　Brainteaser[谜题]- 2

29.2.1

Examples

29.2.2　分析

29.2.3　解答

29.3　Memoization[以避免递归重复计算，如 Fibonacci(斐波那契) 问题]- 1

29.3.1

Examples

29.3.2 分析

29.3.3 解答

29.4 Queue- 3

29.4.1

Examples

分析

解答

29.4.2

Examples

分析

解答

29.5 Reservoir Sampling[随机抽样]- 2

29.5.1

Examples

分析

解答

29.6 Minimax[极小化极大算法]- 3

29.6.1

Examples

分析

解答

29.6.2

Examples

分析

解答

29.7 Other Interview

29.7.1 迷宫字符串

问题

已知：一个二维的字母矩阵和一个由字母组装的单词。输出：该单词是否可以从二维字母矩阵中拼接出来，拼接规则为从矩阵的某一行的某个字母开始，持续向临近的字母扩展（向上、向下、向左或向右），直至拼接出该单词。若最终可以拼接出，则输出 true；若最终不可以拼接出，则输出 false

Example Given 字母矩阵为
A B C D E
E F G H U
A B S D F
输入：ABCGH 输出 true
输入：FGHDSB 输出 true
输入：EFGU 输出 false

解答

队列方法

```
#include <iostream>
#include <list>
using namespace std;

struct Roud{
    int i; //rows
    int j; //columns
    int now; //testCase[now];
    list<char> roud;
};

// queue
list<Roud> all;
list<char> result;

// arrayCharacters
char array[3][5] = { { 'A', 'B', 'C', 'D', 'E' }, { 'E', 'F', 'G', 'H', 'U' }, { 'A',
    'B', 'S', 'D', 'F' } };
```

```
void createReslut(char * testCase)
{
    for (int i = 0; i < strlen(testCase); ++i)
    {
        result.push_back(testCase[i]);
    }
}
void solve(char * testCase)
{
    // Checking The First Elements
    int i = 0;
    for (int _i = 0; _i < 3; ++_i)
    {
        for (int j = 0; j < 5; ++j)
        {
            if (array[_i][j] == testCase[0])
            {
                Roud temp;
                temp.roud.push_back(testCase[0]);
                temp.i = _i;
                temp.j = j;
                temp.now = 0;
                all.push_back(temp);

                if (i == 0)
                    ++i;
            }
        }
    }

    // Queue Checking Roud
    while (testCase[i] != '\0' && all.size() > 0)
    {
        Roud temp = all.front();
        Roud tempN;
        if (result == temp.roud)
            break;

        i = temp.now;
        ++i;
        // Checking 4 direction
            // Up
            if ((temp.i - 1 >= 0) && (testCase[i] == array[temp.i - 1][temp.j]))
            {
                tempN.i = temp.i - 1;
                tempN.j = temp.j;
                tempN.now = temp.now + 1;
```

```
                tempN.roud = temp.roud;
                tempN.roud.push_back(testCase[i]);
                all.push_back(tempN);
            }
            // Down
            if ((temp.i +1 <= 2) && (testCase[i] == array[temp.i + 1][temp.j]))
            {
                tempN.i = temp.i + 1;
                tempN.j = temp.j;
                tempN.now = temp.now + 1;
                tempN.roud = temp.roud;
                tempN.roud.push_back(testCase[i]);
                all.push_back(tempN);
            }
            // Left
            if ((temp.j -1 >= 0) && (testCase[i] == array[temp.i][temp.j-1]))
            {
                tempN.i = temp.i;
                tempN.j = temp.j-1;
                tempN.now = temp.now + 1;
                tempN.roud = temp.roud;
                tempN.roud.push_back(testCase[i]);
                all.push_back(tempN);
            }
            // Right
            if ((temp.j+1 <= 4) && (testCase[i] == array[temp.i][temp.j+1]))
            {
                tempN.i = temp.i;
                tempN.j = temp.j + 1;
                tempN.now = temp.now + 1;
                tempN.roud = temp.roud;
                tempN.roud.push_back(testCase[i]);
                all.push_back(tempN);
            }
            all.pop_front();
    }

    if (all.size()>0)
        cout << "true" << endl;
    else
        cout << "false" << endl;
}

int main()
{
    char test[256];
    cin >> test;
```

```
    createReslut(test);
    solve(test);

    system("PAUSE");
    return 0;
}
```