

Solutions to Algorithm Design and Analysis

yunzinan

2023 年 10 月 28 日

目录

1 1 抽象的算法设计与分析	14
1.1 1.1 三个数排序	14
1.1.1	14
1.1.2	14
1.2 1.2 三个数的中位数	15
1.2.1	15
1.2.2	15
1.2.3	15
1.3 1.3 最小集合覆盖问题	15
1.3.1	15
1.3.2	16
1.3.3	17
1.4 1.4 换硬币问题	17
1.4.1	17
1.5 1.5	18
1.6 1.6	18
1.7 1.7 多项式计算	18
1.8 1.8 整数相乘	19
1.8.1	19
1.8.2	19
1.9 1.9	20
1.10 1.10	20
1.10.1	20
1.10.2 恰有两个元素相等的输入	20
1.10.3 所有元素等概率输入	20

2	2 从算法的视角重新审视数学的概念	21
2.1	2.1	21
2.2	2.2	21
2.3	2.3	21
	2.3.1	21
	2.3.2	22
2.4	2.4	22
2.5	2.5	22
	2.5.1	22
	2.5.2	22
2.6	2.6	22
2.7	2.7 漸进增长率的基本性质	24
2.8	2.8	24
	2.8.1	24
	2.8.2	24
2.9	2.9 斐波那契数列	24
	2.9.1	25
	2.9.2	25
2.10	2.10	25
2.11	2.11	26
	2.11.1 法一	26
	2.11.2 法二: Stirling 公式	26
2.12	2.12	26
2.13	2.13	27
	2.13.1 $\lceil \log n \rceil!$ 是否多项式有界?	27
	2.13.2 $\lceil \log \log n \rceil!$ 是否多项式有界?	27
2.14	2.14	28
2.15	2.15	28
2.16	2.16 解递归式	28
	2.16.1 $T(n) = 2T(n/3) + 1$	28
	2.16.2 $T(n) = T(n/2) + c \log n$	28
	2.16.3 $T(n) = T(n/2) + cn$	29
	2.16.4 $T(n) = 2T(n/2) + cn$	29
	2.16.5 $T(n) = 2T(n/2) + cn \log n$	29
	2.16.6 $T(n) = 3T(n/3) + n \log^3 n$	29
	2.16.7 $T(n) = 2T(n/2) + cn^2$	29
	2.16.8 $T(n) = 49T(n/25) + n^{3/2} \log n$	29

2.16.9	$T(n) = T(n - 1) + 2$	29
2.16.10	$T(n) = T(n - 1) + n^c$, 常量 $c \geq 1$	30
2.16.11	$T(n) = T(n - 1) + c^n$, 常量 $c > 1$	30
2.16.12	$T(n) = T(n - 2) + 2n^3 - 3n^2 + 3n - 1$, $T(1) = 1$, $T(2) = 9$	30
2.16.13	$T(n) = T(n/2) + T(n/4) + T(n/8) + n$	30
2.17	2.17	31
2.18	2.18	32
2.19	2.19	33
2.20	2.20	33
2.20.1	$T(n) = 5T(n/2) + \Theta(n)$	33
2.20.2	$T(n) = 2T(n - 1) + \Theta(1)$	33
2.20.3	$T(n) = 9T(n/3) + O(n^2)$	33
2.21	2.21 带约束的汉诺塔问题	34
2.22	2.22	34
2.22.1		34
2.22.2		34
2.23	2.23	34
2.23.1	算法正确性证明	34
2.23.2	时间复杂度分析	35
2.24	2.24	35
2.24.1	MYSTERY(n)	35
2.24.2	PERSKY(n)	35
2.24.3	PRESTIFEROUS(n)	35
2.24.4	CONUNDRUM(n)	35
3	3 蛮力算法设计	36
3.1	3.1 插入排序的正确性	36
3.2	3.2 冒泡排序	37
3.2.1	正确性证明	37
3.2.2	时间复杂度分析	37
3.2.3	改进	37
3.3	3.3 摊煎饼	37
3.3.1	基本原理	37
3.3.2	时间复杂度分析	38
3.4	3.4 蛇形数组	38
3.5	3.5 PREVIOUS-LARGER 法一 (单调栈)	39
3.5.1	基本原理	39

3.5.2	时间复杂度分析	39
3.6	3.5 PREVIOUS-LARGER 法二 (标记法)	39
3.6.1	基本原理	39
3.6.2	时间复杂度分析	39
3.7	3.6 对调左右子数组	39
3.7.1	时间 $O(n^2)$, 空间 $O(1)$	39
3.7.2	时间 $O(n)$, 空间 $O(n)$	39
3.7.3	时间 $O(n)$, 空间 $O(1)$	40
3.8	3.7 颠倒所有单词的顺序	40
3.9	3.8 微博名人问题	40
3.9.1		40
3.9.2		40
3.10	3.9 最大和连续子序列	41
3.10.1	$O(n^3)$ 蛮力算法	41
3.10.2	$O(n^2)$ 优化遍历	41
3.10.3	$O(n \log n)$ 分治	41
3.10.4	$O(n)$ 优化遍历	41
3.10.5	$O(n)$ DP	41
4	4 分治排序	41
4.1	4.1	41
4.2	4.4	42
4.2.1		42
4.2.2		42
4.3	4.7	43
4.4	4.8 k-sorted 问题	43
4.5	4.9 螺钉螺母配对问题	44
4.5.1	大致思路	44
4.5.2	伪代码	45
4.5.3	复杂度分析	45
4.6	4.11	46
4.6.1		46
4.6.2		47
4.7	4.14 易位词	47
4.7.1	第一种算法	47
4.7.2	第二种算法	48
4.8	4.15 空闲时间	48

5 5 线性时间选择	50
5.1 5.2 6 次找出 5 个元素的中位数	50
5.2 5.4	51
5.3 5.5	52
5.4 5.6 最大的 k 个数	52
5.4.1 $O(n \log n)$	52
5.4.2 $O(n + k \log n)$	52
5.4.3 $O(n + k^2)$ 法一: 堆	52
5.4.4 $O(n + k^2)$ 法二: 排序	53
5.4.5 $O(n + k \log k)$ 法一: 堆	53
5.4.6 $O(n + k \log k)$ 法二: 排序	54
5.5 5.7 最接近中位数的 k 个数	55
5.5.1 $O(n \log n + k)$	55
5.5.2 $O(n + k \log k)$	56
5.5.3 $O(n + k \log k)$ 法二 (课上介绍) $SEL \oplus PAR$	56
5.6 5.9 多个数组的第 k 大元素	56
5.6.1 两个有序数组	56
5.6.2 三个有序数组	57
5.6.3 m 个有序数组	58
5.7 5.10	59
5.7.1	59
5.7.2	59
5.7.3	59
6 6 对数时间查找	61
6.1 6.2	61
6.2 6.3 证明 RBT 的两种定义等价	61
6.2.1 直接定义 \Rightarrow 递归定义	61
6.2.2 递归定义 \Rightarrow 直接定义	61
6.3 6.4 证明 RBT 的平衡性	62
6.3.1 性质 1: 内部黑色节点	62
6.3.2 性质 2: 内部节点	63
6.3.3 性质 3: 高度	63
6.4 6.5	64
6.5 6.8	64
6.5.1	64
6.5.2	64

6.5.3	65
6.5.4	65
6.6 6.14	66
6.6.1	66
6.6.2 $O(\log n)$ 时间找到一个极小值	66
7 7 分治算法设计要素	67
7.1 7.1	67
7.1.1 BF 算法	67
7.1.2 法三: (离散化) 动态树状数组统计逆序对	69
7.2 7.2 芯片检测算法的细节	69
7.2.1 当芯片的数目为奇数时	69
7.2.2 最后的情况	69
7.3 7.4	70
7.3.1	70
7.3.2	70
7.3.3 算法时间复杂度分析	72
7.4 7.5	72
7.4.1	72
7.4.2	72
7.5 7.7 常见元素	73
7.5.1	73
7.5.2	73
7.5.3	74
7.5.4	75
7.5.5 不会写	75
7.6 7.8 maxima	75
7.6.1	75
7.6.2	75
7.7 7.9 多数政党	76
7.7.1	76
7.7.2	76
7.8 7.12 寻找缺失的比特串	76
7.8.1	76
7.8.2	77

8	8 图的深度优先遍历	77
8.1	8.1	77
8.1.1	77
8.1.2	77
8.1.3	78
8.2	8.2 DFS 算法基于栈的非递归实现	78
8.3	8.3 定理 8.1 证明	79
8.3.1	79
8.3.2	79
8.3.3	79
8.4	8.4	80
8.5	8.5 割点的等价表述	80
8.6	8.6	80
8.7	8.7 有向图的收缩图无环	80
8.8	8.8	81
8.9	8.9	81
8.10	8.10	81
8.11	8.11 Tarjan 找桥算法的正确性	82
8.11.1	82
8.11.2	82
8.12	8.12 无向图和有向图的连通性	82
8.12.1	无向连通图中至少有两个 non-cut vertex	82
8.12.2	83
8.12.3	83
8.13	8.13	83
8.14	8.14	84
8.15	8.15	85
8.15.1	85
8.15.2	85
8.16	8.16	85
8.17	8.17	86
8.18	8.18	86
8.19	8.19	86
8.20	8.20 顶点间的”one-to-all” 可达性问题	86
8.20.1	86
8.20.2	87
8.21	8.21	87

8.22	8.22 影响力值	87
8.22.1	8.22.1	88
8.22.2	8.22.2	88
8.23	8.23 线性时间找奇圈	89
8.24	8.24	89
8.25	8.25	89
8.25.1	8.25.1	89
8.25.2	8.25.2	89
8.26	8.26 小孩排队问题	90
8.26.1	8.26.1	90
8.26.2	8.26.2	90
8.27	8.27 神秘文字的字母排序问题	90
8.28	8.28 2-SAT 问题	91
8.28.1	8.28.1	91
8.28.2	8.28.2	91
8.28.3	8.28.3	91
8.28.4	8.28.4	92
8.28.5	8.28.5	92
8.28.6	8.28.6	93
9	9 图的广度优先遍历	95
9.1	9.1	95
9.2	9.2	95
9.3	9.3 DFS 能否判断二分图	96
9.4	9.4	96
9.4.1	9.4.1 DFS	96
9.4.2	9.4.2 BFS	96
9.5	9.5	97
9.6	9.6 离线处理祖先后继关系	97
9.7	9.7	97
9.7.1	9.7.1 无向图的情况	97
9.7.2	9.7.2 有向图的情况	97
9.8	9.8	98
9.8.1	9.8.1 边权为 1 的 MST	98
9.8.2	9.8.2 m = n + 10	98
9.8.3	9.8.3 边权为 1 或 2	98
9.9	9.9	99

9.9.1	99
9.9.2	99
9.10 9.10	99
9.11 9.11 会员制舞会	100
10 10 图优化问题的贪心求解	100
10.1 10.3	100
10.1.1	100
10.1.2	101
10.1.3	101
10.2 10.6 比较 Prim 和 Kruskal 算法的好坏	102
10.2.1 Prim 算法	102
10.2.2 Kruskal 算法	102
10.2.3 比较	103
10.3 10.10	103
10.3.1 $e \notin E' \wedge \hat{w}(e) > w(e)$	103
10.3.2 $e \notin E' \wedge \hat{w}(e) < w(e)$	103
10.3.3 $e \in E' \wedge \hat{w}(e) < w(e)$	103
10.3.4 $e \in E' \wedge \hat{w}(e) > w(e)$	103
10.4 10.13	104
10.4.1 法一: Kruskal 算法	104
10.5 10.14	104
10.6 10.15	104
10.6.1	104
10.6.2	104
10.6.3	105
10.6.4	105
10.6.5	105
10.6.6	105
10.6.7	106
10.7 10.16	106
10.8 10.17	107
10.9 10.21	107
10.10 10.23	107
10.10.1	107
10.10.2	108
10.11 10.25	108

10.12	10.27	109
10.13	10.31	109
	10.13.1	109
	10.13.2	109
10.14	10.33 推广的最短路径问题	110
10.15	10.34	110
10.16	10.36	111
10.17	10.38	112
	10.17.1	112
	10.17.2	112
11	11 贪心算法设计要素	112
11.1	11.1	112
12	12 图优化问题的动态规划求解	113
12.1	12.1	113
	12.1.1	113
	12.1.2	114
12.2	12.2 最大吞吐率	114
	12.2.1	114
	12.2.2	115
12.3	12.4	115
12.4	12.7	116
13	13 动态规划算法设计要素	116
13.1	13.1	116
	13.1.1 算法设计	116
	13.1.2 算法分析	116
13.2	13.2	116
13.3	13.3	117
13.4	13.4	117
13.5	13.5	118
	13.5.1 法一 DP	118
	13.5.2 法二 二分 + 贪心	118
13.6	13.6	118
	13.6.1 均为正数的情况	118
	13.6.2 有正有负的情况	119
13.7	13.7 打家劫舍	119

13.8	13.8 公共子序列问题	119
13.8.1	LCS	119
13.8.2	LCS 变式 1	120
13.8.3	LCS 变式 2	120
13.9	13.9 不重叠的最长回文串	121
13.10	13.10 公共超序列	121
13.11	13.11 LCS 变式	122
13.11.1	122
13.11.2	122
13.11.3	123
13.12	13.12	123
13.12.1	123
13.12.2	124
13.13	13.13 最长回文子序列	124
13.13.1	法一: DP	124
13.13.2	法二: 转化为 LCS	124
13.13.3	回文串分解	124
13.14	13.14	125
13.14.1	区间划分 DP	125
13.14.2	贪心?	125
13.15	13.15 零钱兑换变式	125
13.15.1	125
13.15.2	126
13.15.3	126
13.16	13.16 Vertex Cover	127
13.17	13.17 带权任务调度问题	127
13.17.1	贪心失败	127
13.17.2	$O(n^2)$ 解法	128
13.17.3	动态规划解法	128
13.18	13.18	128
13.19	13.19	129
13.20	13.23	129
14	14.14 堆与偏序关系	130
14.1	14.1	130
14.2	14.2	131
14.3	14.3	132

14.3.1	132
14.3.2	132
14.4 14.4	133
14.4.1 法一 (失败)	133
14.4.2 法二 (上课介绍)	134
14.4.3 法三 (上课介绍)	135
14.5 14.5 k 个链表的合并	136
14.5.1 法一	136
14.5.2 法二	136
14.6 14.6	136
15 15 并查集与动态等价关系	138
15.1 15.1 并查集的蛮力实现	138
15.1.1 基于矩阵	138
15.1.2 基于数组	139
15.2 15.3	139
16 16 哈希表与查找	140
16.1 16.1	140
16.2 16.4 不同哈希表的存储效率	141
16.2.1	141
16.2.2	141
17 18 平摊分析	141
17.1 18.1 湖景房	141
17.1.1 算法正确性分析	141
17.1.2	142
17.2 18.3	143
17.3 18.5	143
18 19 对手论证	145
18.1 19.1	145
18.2 19.3	145
18.3 19.4 好坏芯片的对手论证	146
18.4 19.5	147
18.4.1 找元素最大值	147
18.4.2 找第二大元素	148
18.5 19.6 唯一的特殊元素	149

18.5.1	149
18.5.2 平均情况下的比较次数	150
18.5.3 最坏情况的下界	151
19 20 问题与归约	151
19.1 20.1	151
19.1.1 CLIQUE	151
19.1.2 KNAPSACK	152
19.1.3 INDEPENDENT-SET	153
19.1.4 VERTEX-COVER	154
19.2 20.2 多项式时间规约关系是传递关系	155
19.3 20.3	155
19.4 20.4	156
19.5 20.5	156
20 21 NP 完全性理论初步	156
20.1 21.1	156
20.1.1	156
20.1.2	156
20.2 21.3 伪最大团问题	157
20.2.1	157
20.2.2	157
20.3 21.4	157
20.3.1	157
20.3.2	157
20.4 21.5	158
20.4.1	158
20.4.2	158
20.4.3	158
20.5 21.6 支配集和集合覆盖问题	159

Chapt. 1 抽象的算法设计与分析

P 1.1 三个数排序

1)

Algorithm 1: SORT-THREE-ELEMENTS(a, b, c)

output: 3 sorted elements

```
1 if  $a \geq b$  then swap ( $a, b$ );
2 if  $b < c$  then
3   return;
4 else
5   if  $a < c$  then
6     swap ( $b, c$ );
7     return ;
8   else
9     swap ( $a, c$ );
10    swap ( $a, b$ );
11    return;
```

2)

最坏情况

3 次.

平均情况

3 个元素一共有 $3! = 6$ 种排序结果, 通过画决策树发现, 其中有 2 种只需要比较 2 次, 其余 4 中需要比较 3 次, 因此平均比较次数 $= \frac{2 \times 2 + 3 \times 4}{6} = \frac{8}{3}$ 次.

最坏情况下, 3 个不同的整数至少需要比较 3 次, 证明如下:

证明. 即证明比较次数不可能小于等于 2 次.

考虑排序过程构成的决策树, 满足所有的排序结果均出现在叶子节点上, 且每个叶子节点上至多对应 1 个排序结果, 因此决策树的叶子节点数 $\geq 3! = 6$. 又因为决策树的高度等于最坏情况下的比较次数, 因此当决策树高度小于等于 2 时, 可以证明至多只有

4 个叶子节点, 矛盾, 因此决策树的高度不可能小于 3, 即比较次数至少为 3 次. 又因为第 (2) 问中已经给出了一种最坏情况下比较次数为 3 次的算法, 因此在最坏情况下至少需要进行 3 次比较. \square

P 1.2 三个数的中位数

1)

注：捷径

显然, 可以直接调用 1.1 中的排序算法, 并返回其第二个元素 (b).

2)

同上, 因此最坏 3 次, 平均 $\frac{8}{3}$ 次.

3)

至少需要 3 次, 证明如下:

证明. 可以通过证明无论使用何种比较方案, 都能构造出需要比较 3 次的 case, 且至多只需要 3 次.

不失一般性, 假设第一次将 a,b 比较, 结果同样不是一般性, 假设 $a < b$ (否则, 交换 a 和 b). 对第二次比较的情况进行讨论:

1. c, a 比较: 如果 $c < a$, 则可以比较 2 次结束; 如果 $c > a$, 则必须要第 3 次比较 b, c, 其中的较小者为中位数.
2. c, b 比较: 如果 $c > b$, 则可以比较两次结束; 如果 $c < b$, 则必须要第 3 次比较 a, c, 其中的较大者为中位数.

因此至少需要 3 次, 且采用上述比较方案, 3 次一定能比较出结果.

\square

P 1.3 最小集合覆盖问题

1)

反例: $U = \{1, 2, 3, 4, 5\}, S = \{S_1, S_2, S_3\}, S_1 = \{1, 2, 3\}, S_2 = \{2, 3, 4\}, S_3 = \{4, 5\}$, 按照题述算法, 得到的 $T = \{S_1, S_2, S_3\}$, 而最小覆盖为 $\{S_1, S_3\}$.

2)

Algorithm 2: MINIMUM-COVERAGE

input : U, S
output: the minimum coverage T of the given set family S

1 **while** U 非空 **do**
2 统计 S 中每个集合中包含 U 中元素的个数;
3 选择包含元素个数最多的集合 S_i ;
4 $U := U - U \cap S_i$;
5 $S := S - S_i$;
6 $T := T + S_i$;
7 **return** T ;

算法正确性证明

算法可终止性

反证法, 假设算法不可终止. 则必然存在 S 为空集, 而 T 非空的情况 (否则算法还可以继续进行). 由于题目假设存在覆盖, 因此对于 U 中的每一个元素, 都至少属于一个集合. 由于 S 为空集, 则 S 的所有元素都已经被选过, 则 T 已经覆盖了所有的元素, 因此 U 为空集, 与 U 非空矛盾. 因此算法是可终止的.

算法正确性

首先证明算法终止时, T 为覆盖. 反证法, 若不然, 则存在一个元素违背覆盖, 则 U 非空, 这与算法的终止条件矛盾, 因此算法终止时 T 为覆盖. 下面证明, 该算法能够找到最小覆盖.

证明. 反证法, 假设存在 T' , $|T'| < |T|$, 且 T' 同样覆盖 U . 记 T 中包含的元素为 S_{Ti} , T' 中包含的元素为 $S'_{T'i}$.

考虑算法 MINIMUM-COVERAGE 的执行过程. 不妨对运行过程中的 S_i 进行转换, 令 $S'_i = S_i \cap U'$, 即去除 S_i 中包含的已经被覆盖的元素, 显然, 根据算法, 这个转换不会影响接下来集合的选取. 因为 $|T'| < |T|$, 因此必然存在某一次选择, $|S'_{Ti}| < |S'_{T'i}|$. 否则, T' 不可能先于 T 完成覆盖 (如果 T' 每次选取的集合含有的未被覆盖的元素都不超过 T 选择的, 那么最后覆盖到的元素个数也不超过 T). 而如果存在 $|S'_{Ti}| < |S'_{T'i}|$, 则根据算法规则, 不会选择 S_{Ti} , 矛盾.

因此不存在这样更小的覆盖, 即算法能够找到最小覆盖. □

注：上述证明错误，该算法不正确

3) 最小集合覆盖是NP问题，因此上述四种算法只有Alg4能保证总是得出最小覆盖，特别地，

针对Alg3，给出如下反例，说明其不能保证总是得到最优解：

$$U = \{1, 2, 3, 4, 5, 6\}, S_1 = \{1, 2, 3\}, S_2 = \{1, 4\}, S_3 = \{2, 5\}, S_4 = \{3, 6\} \Rightarrow$$

输出： $\{S_1, S_2, S_3, S_4\}$ ，正确答案为： $\{S_2, S_3, S_4\}$

问题出在：不是总选择包含未覆盖元素最多的集合，结果就是最优的。可能上来先看起来亏一点，但是后面更赚，例如 $2 + 3 + 3 = 3 + 2 + 2 + 1$ 。而且因为不同的选法是具有后效性的，因此贪心策略不正确，如上图给出的一个反例说明。

3)

第(2)问已经证明。

P 1.4 换硬币问题

1)

对算法的描述比较简单，略过。

找反例

1) SEQ = 9, 2, 2, 2, 2, 2. T = 10.

2) SEQ = 1, 2. T = 2.

3) SEQ = 9, 2, 2, 2, 2, 2. T = 10.

P 1.5

Algorithm 3: COINS(N)

```
1 if N mod 3 = 0 then
2   k = N / 3;
3   print(" 需要 k 个 3 分硬币");
4 else if N mod 3 = 1 then
5   k = (N - 10) / 3;
6   print(" 需要 k 个 3 分硬币, 2 个 5 分硬币");
7 else
8   k = (N - 5) / 3;
9   print(" 需要 k 个 3 分硬币, 1 个 5 分硬币");
```

算法正确性证明

分类讨论较为容易. 略过.

P 1.6

证明. 利用数学归纳法进行证明, 对 n 归纳, 记 $P(k)$: $\text{NEXT}(k) = k+1$.

Basis. $\text{NEXT}(0) = 1$, 成立.

I.H. 假设对于 $k \geq 0$, $\forall i, 0 \leq i \leq k$, $P(i)$ 成立, 即 $\text{NEXT}(i) = i+1$.

I.S. 对于 $n = k+1$,

- $n \bmod 2 = 1$. $\text{NEXT}(k+1) = 2 \times \text{NEXT}(\lfloor \frac{n}{2} \rfloor) = 2 \times \text{NEXT}(k / 2) = 2 \times (k/2+1) = k + 2 = (k + 1) + 1$. 成立.
- $n \bmod 2 = 0$. $\text{NEXT}(k+1) = (k+1)+1$. 成立.

综上所述, 对于任意非负整数 k , $P(k)$ 成立, 即原命题成立. \square

P 1.7 多项式计算

注

关于循环不变量: <https://www.win.tue.nl/~kbuchin/teaching/JBP030/notebooks/loop-invariants.html>

证明. **L.I.**(short for Loop Invariant) 在 $i = k$ 的这次循环开始前, 有 $p = \sum_{j=k+1}^n A[j]x^{j-(k+1)}$.

Initialization. 在 $i = n-1$ 的循环开始前, $p = A[n] = \sum_{j=n}^n A[j]x^{j-n}$. 成立.

Maintenance. 假设在 $i = k$ ($0 \leq k \leq n-1$) 的循环开始前, 循环不变量成立, 即 $p = \sum_{j=k+1}^n A[j]x^{j-(k+1)}$, 则在这次循环中, $p = p \cdot x + A[k] = \sum_{j=k+1}^n A[j]x^{j+1-(k+1)} + A[k] = \sum_{j=k+1}^n A[j]x^{j-(k-1+1)} + A[k]x^{k-k} = \sum_{j=(k-1)+1}^n A[j]x^{j-(k-1+1)}$, 而这就是下一次循环, 即 $i = k-1$ 开始前需要满足的循环不变量.

Termination. 循环终止时, 由循环不变量, 即, 在 $i = -1$ 的循环开始前, 有 $p = \sum_{j=-1+1}^n A[j]x^{j-(-1+1)} = \sum_{j=0}^n A[j]x^j$, 而这就是算法要求的正确输出. \square

P 1.8 整数相乘

1)

注意到第 2 问是第 1 问的推广, 因此直接证明下一问. 略.

2)

证明. 利用数学归纳法, 对 z 归纳, 设 $P(k): \forall y \geq 0, \text{INT-MULT}(y, k) = yk$.

Basis. $P(0)$: 根据算法, $\forall y, \text{INT}(y, 0) = 0$. 成立.

I.H. 假设对于 $k (k \geq 0), \forall 0 \leq i \leq k, P(i)$ 成立.

I.S. 则对于 $n = k+1$ 时, $\forall y, \text{INT-MULT}(y, k+1) = \text{INT-MULT}(c \cdot y, \lfloor \frac{k+1}{c} \rfloor) + y \cdot (z \bmod c)$.

1. $k+1 \bmod c = 0$. $\text{INT-MULT}(y, k+1) = cy \cdot \frac{k+1}{c} + 0 = y(k+1)$. 成立.

2. $k+1 \bmod c > 0$, 不妨记 $k+1 = p \cdot c + q$. $\text{INT-MUIT}(y, k+1) = cyp + yq = y(pc+q) = y(k+1)$. 成立.

综上所述, 对于任意的非负整数 k , $P(k)$ 成立, 即原命题成立. \square

注 : 简化

$$\begin{aligned} c \cdot \lfloor \frac{z}{c} \rfloor \\ = z - z \bmod c \end{aligned}$$

不需要分类讨论.

P 1.9

$$T(n) = \sum_{I \in D_n} \Pr(I)f(I) = \frac{1}{4} \times 10 + \frac{1}{2} \times 20 + \frac{1}{8} \times 30 + \frac{1}{8} \times n = \frac{n}{8} + 16.25$$

P 1.10

UNIQUE 算法用于判断数组中是否元素两两互不相同. 是则返回 TRUE, 否则返回 FALSE.

1)

最坏情况下, 当输入的数组中不存在相同的元素时, 算法需要完整遍历所有的 i, j , 最后才返回 TRUE. 因此容易得出时间复杂度为 $T(n) = \Theta(n^2)$.

2) 恰有两个元素相等的输入

假设数组中仅有两个元素相等, 其余任意两个元素均不等, 在这一前提下, 所有可能的输入等概率出现. 求算法的平均时间复杂度.

平均时间复杂度 = 所有的可能的输入的时间复杂度之和 / 所有的可能输入的个数.
于是

$$T(n) = \frac{1+2+\dots+\frac{n(n-1)}{2}}{\frac{n(n-1)}{2}} = \frac{n^2-n+2}{4}$$

3) 所有元素等概率输入

假设完全等概率, 求算法的平均时间复杂度.

- 任意两个位置 i, j 上元素相等的概率为: $\Pr(A[i] = A[j]) = k \times (\frac{1}{k})^2 = \frac{1}{k}$
- 任意两个位置 i, j 上元素不相等的概率为 $1 - \frac{1}{k}$
- 若循环代价为 i , 则意味着前 $i-1$ 的比较元素对不等, 而第 i 次相等. 因此循环代价近似服从几何分布 $G(\frac{1}{k})$, 直接通过公式得到期望代价为 $\frac{1}{p} = \frac{1}{1/k} = k = O(k)$.

注 : 为什么是“近似”满足几何分布?

- 几何分布要求样本空间是无限的, 而这个问题的样本空间受到元素个数限制
- 在实际算法执行过程中, 前面的结论产生的同时会改变后面的概率, 因此这里的 $\frac{1}{k}$ 实际只是近似的.

Chapt. 2 从算法的视角重新审视数学的概念

P 2.1

$$\begin{aligned}1 < x < 2 &\Rightarrow \lfloor x \rfloor = 1 \\&\Rightarrow \lfloor x^2 \rfloor = 1 \\&\Rightarrow 1 \leq x^2 < 2 \Rightarrow 1 \leq x < \sqrt{2} \\&\Rightarrow 1 < x < \sqrt{2}\end{aligned}$$

P 2.2

证明. 将大于 0 的整数划分为 $[2^0, 2^1 - 1], [2^1, 2^2 - 1], \dots, [2^k, 2^{k+1} - 1], \dots$, 显然, $\forall n \geq 1, \exists k \geq 0, s.t. 2^k \leq n \leq 2^{k+1} - 1$.

此时, $2^k < n + 1 \leq 2^{k+1} \Rightarrow \lceil \log n + 1 \rceil = k + 1, 2^k \leq n < 2^{k+1} \Rightarrow \lfloor \log n \rfloor = k$, 于是 $\lceil \log n + 1 \rceil = \lfloor \log n \rfloor + 1$. \square

P 2.3

1)

证明. 首先证明引理.

引理 : F_n 的奇偶性

$$\forall k \geq 0, F_n = \begin{cases} \text{奇数}, & n = 3k + 1 \text{ 或 } 3k + 2; \\ \text{偶数}, & n = 3k. \end{cases}$$

证明. 通过数学归纳法证明, 对 n 归纳. 记 $P(k)$: $n = 3k$ 时, F_n 为偶数, $n = 3k+1, 3k+2$ 时, F_n 为奇数.

Basis. $P(0)$: 当 $n=0$ 时, $F_0 = 0$ 为偶数. $F_1 = 1, F_2 = 1$ 为奇数. 成立.

I.H. 假设对于 $k \geq 0, P(k)$ 成立.

I.S. 则对于 $n = k+1, F_{3(k+1)} = F_{3k+1} + F_{3k+2} = \text{奇数} + \text{奇数} = \text{偶数}. F_{3(k+1)+1} = F_{3k+2} + F_{3k+3} = \text{奇数} + \text{偶数} = \text{奇数}, F_{3k+5} = F_{3k+3} + F_{3k+4} = \text{偶数} + \text{奇数} = \text{奇数}. 成立.$

综上所述, 对于任意 $k \geq 0, P(k)$ 成立, 即原命题成立. \square

由引理容易得到, 当且仅当 n 被 3 整除时, F_n 为偶数. \square

2)

证明. 用数学归纳法证明, 对 n 归纳, 记 $P(k)(k \geq 1)$: $F_k^2 - F_{k+1}F_{k-1} = (-1)^{k+1}$.

Basis. $P(1)$: $F_1^2 - F_0F_2 = 1 - 0 = 1 = (-1)^{1+1}$. 成立.

I.H. 假设对于 $k \geq 0$, $P(k)$ 成立, 即 $F_k^2 - F_{k+1}F_{k-1} = (-1)^{k+1}$.

I.S. 对于 $P(k+1)$, $F_{k+1}^2 - F_{k+2}F_k = F_{k+1}^2 - (F_{k+1} + F_k)F_k = F_{k+1}(F_{k+1} - F_k) - F_k^2 = F_{k+1}F_{k-1} - F_k^2 = -(-1)^{k+1} = (-1)^{k+1+1}$. 成立.

综上所述, 对于任意正整数 k , $P(k)$ 成立, 即原命题成立. \square

P 2.4

注

方法与问题 2.3(1) 的证明方法非常相似, 故略.

P 2.5

1)

由于第二问是第一问的推广, 故直接证明第二问.

2)

证明. 首先对于二叉树, 有边数 $|E| = n_1 + 2n_2$, 又因为对于树, 满足 $|V| = |E| + 1$, 于是有 $n_0 + n_1 + n_2 = n_1 + 2n_2 + 1 \Rightarrow n_0 = n_2 + 1$. \square

P 2.6

证明. 用数学归纳法证明, 对 n 归纳, 记 $P(k)(k \geq 1)$: $T(k) = k \lceil \log k \rceil - 2^{\lceil \log k \rceil} + 1$.

Basis. $P(1)$: $T(1) = 0 - 1 + 1 = 0$, 符合.

I.H. 假设对于 $k \geq 0$, $\forall 0 \leq i \leq k$, $P(i)$ 成立.

I.S. 对于 $P(k+1)$:

(1) $k + 1 \equiv 0 \pmod{2}$.

$$\begin{aligned} T(k+1) &= 2 \cdot \frac{k+1}{2} \left\lceil \log \frac{k+1}{2} \right\rceil - 2^{\lceil \log \frac{k+1}{2} \rceil + 1} + 2 + (k+1) - 1 \\ &= (k+1) \left\lceil \log \frac{k+1}{2} + 1 \right\rceil - 2^{\lceil \log \frac{k+1}{2} \rceil + 1} + 1 \\ &= (k+1) \lceil \log k + 1 \rceil - 2^{\lceil \log k + 1 \rceil} + 1 \end{aligned}$$

(2) $k + 1 \equiv 1 \pmod{2}$. 不妨设 $k + 1 = 2m + 1$, $\lfloor \frac{k+1}{2} \rfloor = m$, $\lceil \frac{k+1}{2} \rceil = m + 1$.

先讨论主要的情况, 即 $2^p < m < 2^{p+1}$. (只有一个例外: $m = 2^p$). 于是有 $\lceil \log m \rceil = \lceil \log m + 1 \rceil$.

$$\begin{aligned} T\left(\left\lfloor \frac{k+1}{2} \right\rfloor\right) &= T(m) = m\lceil \log m \rceil - 2^{\lceil \log m \rceil} + 1 \\ T\left(\left\lceil \frac{k+1}{2} \right\rceil\right) &= T(m+1) = (m+1)\lceil \log m + 1 \rceil - 2^{\lceil \log m + 1 \rceil} + 1 \\ T(k+1) &= (2m+1)\lceil \log m \rceil - 2^{\lceil \log m \rceil+1} + 2 + (2m+1) - 1 \\ &= (2m+1)(\lceil \log m \rceil + 1) - 2^{\lceil \log m \rceil+1} + 1 \end{aligned}$$

$$\begin{aligned} \lceil \log m \rceil + 1 &= \lceil \log 2m \rceil \\ 2^p < m < 2^{p+1} \Rightarrow 2^{p+1} &< 2m < 2^{p+2} \\ \Rightarrow 2^{p+1} + 1 &< 2m + 1 < 2^{p+2} + 1 \\ \Rightarrow \lceil \log 2m \rceil &= \lceil \log 2m + 1 \rceil = \lceil \log k + 1 \rceil \\ \Rightarrow T(k+1) &= (k+1)\lceil \log k + 1 \rceil - 2^{\lceil \log k + 1 \rceil} + 1 \end{aligned}$$

最后考虑 $m = 2^p$ 的情况, $\lceil \log m \rceil = p$, $\lceil \log m + 1 \rceil = p + 1$.

$$\begin{aligned} T(m) &= mp - 2^p + 1 \\ T(m+1) &= (m+1)(p+1) - 2^{p+1} + 1 \\ T(2m+1) &= 2mp + m + p + 1 - 2^p - 2^{p+1} + 2 + (2m+1) - 1 \\ &= (2m+1)(p+2-1) + (m-2^p) - 2^{p+1} + 2 \\ &= (2m+1)(p+2) - 2m - 1 - 2^{p+1} + 2 \\ &= k(p+2) - 2^{p+1} - 2p + 1 + 1 \\ &= k(p+2) - 2^{p+2} + 1 \\ &= k\lceil \log k + 1 \rceil - 2^{\lceil \log k + 1 \rceil} + 1 \end{aligned}$$

综上所述, 原命题成立.

□

注：比较繁琐的一题

涉及取整函数和取对数函数, 因此导致证明非常繁琐. 这也是为什么在算法分析中使用“平滑函数”去除取整符号的意义吧.

P 2.7 渐进增长率的基本性质

注

比较基础的概念题，按部就班按照定义说明即可。

P 2.8

1)

$$\log n < n < n \log n < n^2 = n^2 + \log n < n^3 < n - n^3 + n^5 < 2^n$$

2)

$$\begin{aligned}\log \log n &< \log n = \ln n < (\log n)^2 \\ &< \sqrt{n} < n < n \log n < n^{1+\varepsilon} \\ &< n^2 = n^2 + \log n < n^3 < n - n^3 + n^5 \\ &< 2^n = 2^{n-1} < e^n < n!\end{aligned}$$

注意：个人感觉比较迷惑的是 $(\log n)^k < n^\varepsilon$, 这里 k, ε 都是任意常数。

P 2.9 斐波那契数列

注：斐波那契数列通项公式求法

数论解法 [编辑]

实际上，如果将斐波那契数列的通项公式写成 $a_n - a_{n-1} - a_{n-2} = 0$ ，即可利用解二阶线性齐次递归关系式的方法，写出其特征多项式 $\lambda^2 - \lambda - 1 = 0$

(该式和表达斐波那契数列的矩阵的特征多项式一致)，然后解出 $\lambda_1 = \frac{1}{2}(1 + \sqrt{5})$, $\lambda_2 = \frac{1}{2}(1 - \sqrt{5})$ ，即有 $a_n = c_1 \lambda_1^n + c_2 \lambda_2^n$ ，其中 c_1, c_2 为常数。

我们知道 $a_0 = 0, a_1 = 1$ ，因此 $\begin{cases} c_1 + c_2 = 0 \\ \frac{c_1(1+\sqrt{5})}{2} + \frac{c_2(1-\sqrt{5})}{2} = 1 \end{cases}$ ，解得 $c_1 = \frac{1}{\sqrt{5}}, c_2 = -\frac{1}{\sqrt{5}}$ 。

图 1：图源：<https://zh.wikipedia.org/zh-cn/斐波那契数>

1)

法一：求通项公式

通过证明存在斐波那契数列存在极限为黃金比例 > 1.5 , 即存在一个 $N, \forall n > N, F(n+1)/F(n) > 1.6$, 因此 $\exists C, F_n > A_n = C \times 1.6^n$, 将右式同样记为一个数列 B_n , 计算 $C_n = \frac{A_n}{B_n}$, 容易, 显然存在 $k, \forall n > k, C_n > 1$, 因此 $\forall n > k, F_n > B_n$.

The screenshot shows a Quora post by user Justin Rising. The post discusses the ratio F_{n+1}/F_n and its convergence to the golden ratio ϕ . It mentions that if F_{n+1}/F_n converges to x as $n \rightarrow \infty$, then F_{n-1}/F_n converges to $1/x$. The post also notes that there are two solutions to the equation $x = 1 + 1/x$, one positive and one negative, but the limit must be the positive solution, which is the golden ratio ϕ . The post concludes by stating that showing the limit exists is trickier and involves a closed form solution to the recurrence relation.

图 2: 图源: <https://qr.ae/prBw6G>

法二：举反例

错误. 反例: 当 $n = 100$ 时, $\text{Fib}(n) \approx 3.54 \times 10^{20} > 100(\frac{3}{2})^{100} \approx 4.07 \times 10^{19}$.

2)

证明. 用数学归纳法证明如下: 记 $P(k): F(k) \geq 0.01 \left(\frac{3}{2}\right)^k$

Basis. $F(1) = 1$ 成立. $F(2) = 1$ 成立.

I.H. 假设对于 $k \geq 3, \forall x \leq k, P(x)$ 成立.

I.S. 对于 $n = k+1, F(k+1) = F(k) + F(k-1) \geq 0.01 \left[\left(\frac{3}{2}\right)^{k-1} + \left(\frac{3}{2}\right)^k \right] \geq 0.01 \left[\frac{1}{2} \times \left(\frac{3}{2}\right)^k + \left(\frac{3}{2}\right)^k \right] = 0.01 \left(\frac{3}{2}\right)^{k+1}$.

□

P 2.10

即要求 $f(n) < g(n) \Rightarrow n^2 + 4n < 29n + 3$, 解得当 $0 \leq x \leq 25$ 时不等式成立.

P 2.11

1) 法一

证明. 因为

$$\begin{aligned}\log(n!) &= \log 1 + \log 2 + \cdots + \log(n-1) + \log n \\ &\geq \log \frac{n}{2} + \log \frac{n+1}{2} + \cdots + \log \frac{2n}{2} \\ &\geq \frac{n}{2} \log \frac{n}{2} \\ &= \frac{1}{2}(n \log n - n) \\ &\geq \frac{1}{4}n \log n\end{aligned}$$

对于上式, 若要求成立, 则 $\log n \geq 2$, 因此只要令 $N = 4, \forall n \geq N, \log n! \geq \frac{1}{4}n \log n = \Omega(n \log n)$.

又因为

$$\begin{aligned}\log(n!) &= \log 1 + \log 2 + \cdots + \log(n-1) + \log n \\ &\leq n \log n = O(n \log n)\end{aligned}$$

因此 $\log(n!) = \Theta(n \log n)$.

□

2) 法二: Stirling 公式

注意到:

$$\sqrt{2\pi n} \left(\frac{n}{e}\right)^n e^{\frac{1}{12n+1}} < n! < \sqrt{2\pi n} \left(\frac{n}{e}\right)^n e^{\frac{1}{12n}}$$

于是有 $n! = \Theta(n^n) \Rightarrow \log(n!) = \Theta(n \log n)$.

P 2.12

证明. 首先证明引理: $\log \log n = o(\log n)$.

引理

$$\lim_{n \rightarrow \infty} \frac{\log \log n}{\log n} = \lim_{n \rightarrow \infty} \frac{\frac{1}{\log n \ln 2} \cdot \frac{1}{n \ln 2}}{\frac{1}{n \ln 2}} = \lim_{n \rightarrow \infty} \frac{1}{n \ln 2} = 0$$

因为 $k \log k = \Theta(n)$, 因此 $\exists 0 < C < +\infty, \lim_{n \rightarrow \infty} \frac{k \log k}{n} = C$.

于是,

$$\lim_{n \rightarrow \infty} \frac{k}{n / \log n} = \lim \frac{k \log k}{n / \log n \cdot \log k} = \lim \frac{C}{\log k / \log n} = C \lim \frac{\log n}{\log k} = C \lim \frac{\log k + \log \log k}{\log k} = C.$$

根据定义, $k = \Theta(n / \log n)$.

□

P 2.13

1) $\lceil \log n \rceil!$ 是否多项式有界?

$\lceil \log n \rceil!$ 不是多项式有界的, 证明如下:

证明. 只要证明, $\forall k \in \mathbb{N}^+, f(k) = c_k n^k + c_{k-1} n^{k-1} + \dots + c_0 n^0, \lim \frac{\lceil \log n \rceil!}{f(k)} = +\infty$.

$$\begin{aligned} \lim \frac{\lceil \log n \rceil!}{f(k)} &\geq \lim \frac{(\log n)!}{f(k)} \\ &= \lim \frac{(\log n)!}{n^k} \\ &\geq \lim \frac{\log \frac{n}{2} \cdot \log \frac{n}{2} \cdots \log \frac{n}{2}}{n^k} \\ &= \lim \frac{(\log n/2)^{n/2}}{n^k} \\ &\stackrel{m=n/2}{=} \lim \frac{(\log m)^m}{2^k \times m^k} \\ &\geq \lim \frac{2^m}{2^k \times m^k} \\ &= +\infty \end{aligned}$$

因此, $\lceil \log n \rceil!$ 不是多项式有界的.

□

2) $\lceil \log \log n \rceil!$ 是否多项式有界?

$\lceil \log \log n \rceil!$ 不是多项式有界的, 证明类似第 (1) 问.

证明.

$$\begin{aligned}
\lim \frac{[\log \log n]!}{f(k)} &\geq \lim \frac{(\log \log n)!}{f(k)} \\
&= \lim \frac{(\log \log n)!}{n^k} \\
&\geq \lim \frac{\log \log \frac{n}{2} \cdot \log \log \frac{n}{2} \cdots \log \log \frac{n}{2}}{n^k} \\
&= \lim \frac{(\log \log n/2)^{n/2}}{n^k} \\
&\stackrel{m=n/2}{=} \lim \frac{(\log \log m)^m}{2^k \times m^k} \\
&\geq \lim \frac{2^m}{2^k \times m^k} \\
&= +\infty
\end{aligned}$$

□

P 2.14

$$\log f(n) = \log n \log n$$

$$\log g(n) = n \log \log n$$

显然, $\log^2 n = o(n) = o(n \log \log n)$, 即 $f(n)$ 的增长率小于 $g(n)$ 的增长率.

P 2.15

$T(n) = 2^{n-2}(1+k)$ ($n \geq 2$), $T(1) = 1$, 用数学归纳法证明如下, 对 n 归纳:

Basis. $n = 2$. $T(2) = 1 + k$ 成立.

I.H. 假设对于 $k \geq 2$, $T(k) = 2^{k-2}(1+k)$.

I.S. 则对于 $n = k+1$, $T(k+1) = \sum_{i=1}^{k-1} + k + T(k) = 2T(k) = 2^{(k+1)-2}(1+k)$.

综上所述, 对于任意 $n \geq 2$, $T(n) = 2^{n-2}(1+k)$ ($n \geq 2$). $T(1) = 1$.

P 2.16 解递归式

1) $T(n) = 2T(n/3) + 1$

$E = \log_3 2$, $f(n) = O(n^{E-E})$, 符合主定理 case1, 于是有 $T(n) = \Theta(n^E) = \Theta(n^{\log_3 2})$

2) $T(n) = T(n/2) + c \log n$

$E = \log_2 1 = 0$, $f(n) = c \log n = \Theta(n^E \log^1 n)$, 符合主定理 case2, 于是有 $T(n) = \Theta(\log^2 n)$.

$$3) \quad T(n) = T(n/2) + cn$$

$E = \log_2 1 = 0, f(n) = cn = \Omega(n^{E+1})$, 又因为 $f(n/2) = \frac{cn}{2} \leq \frac{1}{2}cn$, 符合主定理 case3, 于是有 $T(n) = \Theta(f(n)) = \Theta(n)$.

$$4) \quad T(n) = 2T(n/2) + cn$$

$E = \log_2 2 = 1, f(n), f(n) = \Theta(n)$, 符合主定理 case2, 于是有 $T(n) = \Theta(n \log n)$.

$$5) \quad T(n) = 2T(n/2) + cn \log n$$

$E = \log_2 2 = 1, f(n) = \Theta(n^E \log n)$, 符合主定理 case2, 于是 $T(n) = \Theta(n \log^2 n)$.

$$6) \quad T(n) = 3T(n/3) + n \log^3 n$$

$E = \log_3 3 = 1, f(n) = \Theta(n^E \log^3 n)$, 符合主定理 case2, 于是 $T(n) = \Theta(n \log^4 n)$.

$$7) \quad T(n) = 2T(n/2) + cn^2$$

$E = \log_2 2 = 1, f(n) = cn^2 = \Omega(n^{E+1})$, 且 $2f(n/2) = \frac{1}{2}cn^2$, 符合主定理 case3, 于是 $T(n) = \Theta(f(n)) = \Theta(n^2)$.

$$8) \quad T(n) = 49T(n/25) + n^{3/2} \log n$$

$E = \log_{25} 49 = \log_5 7 < \frac{3}{2}, f(n) = \Omega(n^{E+(3/2-E)})$, 且 $49f(n/25) \leq \frac{49}{125}n^{3/2} \log n$, 符合主定理 case3, 于是 $T(n) = \Theta(f(n)) = \Theta(n^{3/2} \log n)$.

$$9) \quad T(n) = T(n-1) + 2$$

$$\begin{aligned} T(n) &= T(n-1) + 2 \\ &= T(n-2) + 2 * 2 \\ &= T(n-3) + 3 * 2 \\ &= \dots \\ &= T(n-k) + k * 2 \\ &= T(1) + (n-1) + 2 \\ &= 2n - 1 = \Theta(n). \end{aligned}$$

10) $T(n) = T(n-1) + n^c$, 常量 $c \geq 1$

$$\begin{aligned}
T(n) &= n^c + T(n-1) \\
&= n^c + (n-1)^c + T(n-2) \\
&= \sum_{i=k+1}^n i^c + T(n-k) \\
&= \sum_{i=2}^n i^c + 1 \\
T(n) &\geq \frac{n}{2} \left(\frac{n}{2}\right)^c = \frac{1}{2^{c+1}} n^{c+1} = \Omega(n^{c+1}).
\end{aligned}$$

$$T(n) \leq n \cdot n^c = n^{c+1} = O(n^{c+1}).$$

于是 $T(n) = \Theta(n^{c+1})$.

11) $T(n) = T(n-1) + c^n$, 常量 $c > 1$

$$T(n) = \sum_{i=2}^n c^i + 1 = \Theta(c^n)$$

12) $T(n) = T(n-2) + 2n^3 - 3n^2 + 3n - 1$, $T(1) = 1$, $T(2) = 9$

容易证明, $f(n) = \Theta(n^3)$.

12) 由于 $f(n) = 2n^3 - 3n^2 + 3n - 1 = \Theta(n^3)$

对 n 的奇偶性讨论如下:

① 若 $n = 2k$, 则: $T(2k) = T(2k-2) + \Theta(k^3) = T(2(k-1)) + \Theta(k^3)$, 求和易得:

$$T(2k) = \Theta(k^4)$$

② 若 $n = 2k-1$, 则: $T(2k-1) = T(2k-3) + \Theta(k^3) = T(2(k-1)-1) + \Theta(k^3)$, 求和易得:

$$T(2k-1) = \Theta(k^4)$$

综合①②: $T(n) = \Theta(n^4)$

13) $T(n) = T(n/2) + T(n/4) + T(n/8) + n$

proof by substitution. 猜测 $T(n) = \Theta(n)$, 即 $\exists N, \exists c_1, c_2, \forall n > N, s.t. c_1 n \leq T(n) \leq c_2 n$. 下面给出证明.

$$T(n) = T(n/2) + T(n/4) + T(n/8) + n \leq \left(\frac{7}{8}c_2 + 1\right)n \leq c_2 n$$

只需取 $c_2 \geq 8$, 上式即成立. 不妨令 $c_2 = 8$, 于是 $T(1) = 1 \leq c_2 n$, 对于基础情况也成立. 于是 $T(n) = O(n)$.

$$T(n) = T(n/2) + T(n/4) + T(n/8) + n \geq (\frac{7}{8}c_1 + 1)n \geq c_1 n$$

只需取 $c_1 \leq 8$, 上式即成立. 不妨令 $c_1 = \frac{1}{2}$, 于是 $T(1) = 1 \geq c_1 n$, 对于基础情况也成立. 于是 $T(n) = \Omega(n)$.

综上所述, $T(n) = \Theta(n)$.

注 : 一个简单而不严谨的观察

如果认为 $T(n) = \Theta(n)$, 那么本式和 $T'(n) = T'(7n/8) + n$ 可以认为是相等的, 而对于后者, 用主定理也能推出 $T'(n) = \Theta(n)$.

注 : 补充另一种思路

设 $T(n) = \Theta(n^x)$, 即 $T(n) = cn^x$.

$$\begin{aligned} (cn)^x &= (c\frac{n}{2})^x + (c\frac{n}{4})^x + (c\frac{n}{8})^x \\ &\Rightarrow c^x(1 - \frac{1}{2^x} - \frac{1}{4^x} - \frac{1}{8^x})n^x = n \\ &\Rightarrow Cn^x = n \\ &\Rightarrow x = 1 \end{aligned}$$

于是 $T(n) = \Theta(n)$.

P 2.17

$T(n) = nc$, 用数学归纳法, 对 n 归纳, 记 $P(k)$: $T(k) = kc$:

Basis. $P(0)$: 由定义, $T(0) = 0$, 成立.

I.H. 假设对于 $k \geq 0$, $P(k)$ 成立.

I.S. 对于 $P(k+1)$,

$$\begin{aligned} T(k+1) &= \frac{2}{k+1} \left(\sum_{i=0}^k i \cdot C \right) + C \\ &= \frac{2}{k+1} \cdot \frac{k(k+1)}{2} C + C \\ &= (k+1)C \end{aligned}$$

综上所述, 对于任意 $k \geq 0$, $P(k)$ 成立. 即 $T(n) = cn = \Theta(n)$.

P 2.18

猜测 $T(n) = O(n \log n)$, 证明如下:

设 $\exists C, N, s.t. \forall n > N, T(n) \leq Cn \log n$, 对于 $f(n) = O(n)$ 项, 记 $\exists N_1, C_1, \forall n > N_1, f(n) \leq C_1 n$

$$\begin{aligned} T(n) &= \sqrt{n} \cdot T(\sqrt{n}) + O(n) \\ &\leq \frac{1}{2} Cn \log n + C_1 n \\ &\leq Cn \log n \end{aligned} \tag{1}$$

要想让上式成立, 只需要令 $\frac{1}{2} Cn \log n \geq C_1 n$ 即可. 取 $N = \max\{N_1, 10\}, C = 2C_1$, 则上式成立. 即 $T(n) = O(n \log n)$.

注 : 上述证明不够紧

无法用Master定理, 直接递归树逐层求和:

$$\begin{aligned} T(n) &= \sqrt{n}T(\sqrt{n}) + n = n^{\frac{3}{4}}T(n^{\frac{1}{4}}) + 2n = \dots = n^{1-\frac{1}{2k}}T(n^{\frac{1}{2^k}}) + kn, \\ \text{令 } n^{\frac{1}{2^k}} &= 2, \text{ 得到 : } k = \log \log n \Rightarrow T(n) = \Theta(n \log \log n) \end{aligned}$$

以下是详细证明:

$$\begin{aligned} T(n) &= n^{\frac{1}{2}}T\left(n^{\frac{1}{2}}\right) + cn \\ &= n^{\frac{1}{2}}\left(n^{\frac{1}{2^2}}T\left(n^{\frac{1}{2^2}}\right) + cn^{\frac{1}{2}}\right) + cn \\ &= n^{\frac{1}{2}+\frac{1}{4}}T\left(n^{\frac{1}{4}}\right) + 2cn \\ &= n^{\frac{1}{2}+\frac{1}{4}+\frac{1}{8}}T\left(n^{\frac{1}{8}}\right) + 3cn \\ &= \dots \\ &= n^{\sum_{i=1}^k \frac{1}{2^i}}T\left(n^{\frac{1}{2^k}}\right) + kn \end{aligned}$$

于是 $n^{\frac{1}{2^k}} = C \Rightarrow \frac{1}{2^k} \log n = \log C = C' \Rightarrow 2^k = C^* \log n$, 于是 $k = \log(C^* \log n) = \Theta(\log \log n)$.

于是 $T(n) = n^{1-\varepsilon} + kn\Theta(\log \log n) = \Theta(n \log \log n)$.

P 2.19

注 : Advanced Master Theorem

Advanced Master Theorem

对于如下形式的递归式,

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

令 $E = \log_b a$ 为

1. 如果存在某个常数 $\varepsilon > 0$, 使得 $f(n) = O(n^{E-\varepsilon})$, 则 $T(n) = \Theta(n^E)$.

2. 如果存在某个常数 $\varepsilon \geq 0$, 使得 $f(n) = \Theta(n^E \log^\varepsilon n)$, 则

$$T(n) = \Theta(f(n) \log n) = \Theta(n^E \log^{\varepsilon+1} n)$$

3. 如果存在某个常数 $\varepsilon > 0$, 使得 $f(n) = \Omega(n^{E+\varepsilon})$, 且存在某个常数 $c < 1$, 使得对于所有充分大的 n , $af\left(\frac{n}{b}\right) \leq cf(n)$, 则 $T(n) = \Theta(f(n))$.

注意, 对于 case 2, 本定理对课上介绍的情形进行了推广.

反例: $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\log n}$.

对于该递归式, 对于三种 case 均不满足.

P 2.20

1) $T(n) = 5T(n/2) + \Theta(n)$

$E = \log_2 5$, $f(n) = O(n^E - 1)$, 由主定理 case 1, $T(n) = \Theta(n^E) = \Theta(n^{\log_2 5})$.

2) $T(n) = 2T(n - 1) + \Theta(1)$

记 $T(n) = 2T(n - 1) + C$, 令 $S(n) = T(n) + C' \Rightarrow S(n) = 2S(n - 1) - C' + C$, 令 $C' = C$, 于是 $S(n) = 2S(n - 1) = 2^n \Rightarrow T(n) = S(n) - C = 2^n - C = \Theta(2^n)$.

3) $T(n) = 9T(n/3) + O(n^2)$

$E = \log_3 9 = 2$, $f(n) = O(n^2)$, 由主定理, $T(n) = O(n^2 \log n)$.

因此我会选择算法 C.

P 2.21 带约束的汉诺塔问题

基本原理

记 $T(X_1 \rightarrow X_2, k)$ 为将上面的 k 个盘子从 X_1 柱移动到 X_2 柱. 算法的转移思路可以用如下的式子来表示. 通过观察, 容易发现两个性质:

1. $T(X_1 \rightarrow X_2) = T(X_2 \rightarrow X_1)$. 因为第一个转移过程的逆过程就是第二个转移过程
2. $T(A \rightarrow C) = T(B \rightarrow C) \neq T(A \rightarrow B)$.

结合上述两个性质, 利用数学归纳法, 容易证明该算法的正确性.

$$\begin{aligned}T(A \rightarrow C, n) &= T(A \rightarrow C, n - 1) + T(C \rightarrow B, n - 1) + T(A \rightarrow C, 1) + T(B \rightarrow C, n - 1) \\T(A \rightarrow C, n - 1) &= T(C \rightarrow B, n - 1) = T(B \rightarrow C, n - 1) \equiv T(n - 1) \\&\Rightarrow T(n) = 3T(n - 1) + 1\end{aligned}$$

时间复杂度分析

$T(n) = 3T(n - 1) + 1$, 利用与 P2.20B 相同的方法, 可得 $T(n) = \Theta(3^n)$.

P 2.22

1)

ALG1 返回数组中的最小元素.

ALG2 返回数组中的最小元素.

2)

ALG1 的时间复杂度为 $\Theta(n)$.

ALG2 有递归式 $T(n) = 2T(n/2) + O(1)$, 由主定理 case1, $T(n) = \Theta(n)$.

P 2.23

1) 算法正确性证明

证明. 对于一个数组 $A[1..n]$, 记算法过程中调用 $\text{MAXIMUM}(x,y)$, 对 $\text{len} = y - x$ 归纳, 记 $P(k)$: $\text{len} = k$ 时, MAXIMUM 返回正确的答案.

Basis.

$\text{len} = 0$, 说明只有 1 个元素, 算法直接返回该元素, 成立.

$\text{len} = 1$, 说明有 2 个元素, 算法返回较大者, 成立.

I.H. 假设对于 $k \geq 2, \forall \text{len} = i < k, P(i)$ 均成立.

I.S. 对于 $P(k)$, $\text{MAXIMUM}(x,y)$ 调用 $\text{MAXIMUM}(x, \lfloor \frac{x+y}{2} \rfloor)$ 和 $\text{MAXIMUM}(\lfloor \frac{x+y}{2} \rfloor + 1, y)$. 显然, $\lfloor \frac{x+y}{2} \rfloor - x < y - x = k$, $y - (\lfloor \frac{x+y}{2} \rfloor + 1) < y - x = k$, 根据归纳假设, 对 MAXIMUM 的递归调用均能返回左右两段数组的最大值, 算法计算较大者即为全局的最大值, 因此算法能够正确返回. $P(k)$ 成立.

综上所述, 对于任意的 $k \geq 0, P(k)$ 成立. 考虑 A 数组的任意性, 该算法对于任意数组总是能够返回最大值. \square

2) 时间复杂度分析

$W(1) = W(2) = 1, W(n) = 2W(n/2) + O(1)$, 由主定理 case1, $W(n) = \Theta(n)$.

P 2.24

1) MYSTERY(n)

$$T(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^k \sum_{k=1}^j k = O(n^3).$$

2) PERSKY(n)

$$O(n^3).$$

3) PRESTIFEROUS(n)

$$T(n) = \sum_{i=1}^n i^3 = O(n^4).$$

4) CONUNDRUM(n)

注意到当满足 $i + j - 1 \leq n$ 时, 内存循环才能运行, 于是当 $i > \frac{n}{2}$ 时, 循环终止.

$$T(n) = O(n^3).$$

Chapt. 3 蛮力算法设计

P 3.1 插入排序的正确性

首先证明引理：对于内层 while 循环，循环终止时，假设 $i = k$ ，则有 $A[k] \leq temp < A[k + 2]$ 。

证明。对于 $temp \geq A[k]$ ，考虑 while 循环终止前的最后一次判断，有 $temp \geq A[k]$ 或者 $k = 0$ ，因此，由循环终止条件上式成立。

反证法，假设 $temp \geq A[k + 2]$ ，则在 $i = k + 2$ 时，就已经不满足 while 循环条件，于是算法终止时， $i = k + 2 \neq k$ ，矛盾，故假设不成立。□

下面对外层的 for 循环进行证明，循环不变式为：在 $j = k$ 的迭代开始前， $A[1..j-1]$ 已经升序排列。

初始化 $j = 2$ 的循环开始前， $A[1..1]$ 显然有序，满足。

维持假设 $j = k$ 的循环开始前， $A[1..k-1]$ 升序排列。根据上述对 while 循环的证明，假设 while 循环终止时 $i = m$ ， $A[m] \leq temp < A[m+2]$ 。由归纳假设 $\forall 1 \leq x < m$, $A[x] \leq A[x+1]$ ，由于容易证明 while 循环的操作不断将元素后移，因此 $\forall m+2 \leq x < k$, $A[x] \leq A[x+1]$ ，于是 $A[1] \leq A[2] \leq \dots \leq A[m] \leq temp < A[m+2] \leq A[m+3] \leq \dots \leq A[k]$ 。即在 $j = k+1$ 的循环开始前， $A[1..k]$ 升序排列。

终止 于是当 $j = n$ 的循环结束时，即 $j = n+1$ 的循环开始前， $A[1..n]$ 已经升序排列。

证明插入排序的正确性

爱扑bug的熊

• 循环不变式：

在 **for** 循环每次迭代开始之前，子数组 $A[1..j - 1]$ 由原来在 $A[1..j - 1]$ 中的元素组成，且已经排好序。

• 证明：

初始化：第一次迭代开始前 $j = 2$ ， $A[1..j - 1] = A[1..1]$ 天然有序且是原来元素。

保持：若迭代开始前， $A[1..j - 1]$ 有序且由原本元素组成；迭代过程中将 $A[j - 1], A[j - 2], A[j - 3]$ 等右移，直到空出合适的位置放 $A[j]$ ；所以下次迭代开始前， $A[1..j] = A[1..j - 1]$ 有序且由原本元素组成。

终止：循环终止的时候 $j = n + 1$ ，循环不变式告诉我们 $A[1..n + 1 - 1] = A[1..n]$ 有序且由原本元素组成。



P 3.2 冒泡排序

1) 正确性证明

首先对内层循环做证明. 循环不变量: $j = k$ 的循环开始前, $A[1..j]$ 部分的最大值是 $A[j]$.

初始化: 对于 $j = 1$ 的循环开始前, 显然, 数组 $A[1..1]$ 部分的最大值就是 $A[1]$.

维持: 假设 $j = k$ 的循环开始前, 数组 $A[1..k]$ 部分的最大值为 $A[k]$. 在该次迭代中, 如果 $A[j] > A[j+1]$, 则会交换这两个元素, 此时 $A[j+1] = \max\{A[1..j], A[j+1]\} = \max\{A[1..j+1]\}$. 否则, $A[j+1] \geq A[j] \geq A[1..j]$, 同样满足 $A[j+1]$ 是 $A[1..j+1]$ 部分数组的最大值.

终止: 于是当 $j = i-1$ 的迭代结束, 即 $j = i$ 的迭代 (实际不存在) 开始前, 数组 $A[1..i]$ 中的最大元素就是 $A[i]$.

接下来证明外层循环: 循环不变量: 在 $i = k$ 的循环开始前, 数组 $A[k+1..n]$ 部分是数组 $A[1..n]$ 升序排列后的对应部分, 即 $\forall k+1 \leq j \leq n, A[j]$ 是数组中第 j 小的元素.

初始化: 在 $i = n$ 的循环开始前, $A[n+1..n]$ 为空, 满足.

维持: 假设在 $i = k$ 的循环开始前, 不变式满足, 即 $A[k+1..n]$ 部分已经排好序. 在该次迭代中, 由上述结论, 内层循环会选出 $A[1..k]$ 部分的最大值置于 $A[k]$ 处, 于是 $A[k..n]$ 部分就排好序了.

终止: 在 $i = 2$ 的循环结束后, 即 $i = 1$ 的循环开始前, 数组 $A[2..n]$ 部分已经排好序, 而显然 $A[1]$ 就是第 1 小的元素, 因此 $A[1..n]$ 均已排好序.

2) 时间复杂度分析

对于任意输入, 算法不会提前终止, 因此最坏和平均时间复杂度 $T(n) = (n - 1) + (n - 2) + \dots + 1 = \frac{n(n-1)}{2}$.

3) 改进

在渐进意义上没有优化, 改进后的平均时间复杂度和最坏时间复杂度为 $O(n^2)$.

对于最坏情况, 时间复杂度没有变化.

对于平均情况, 时间复杂度的系数变为原先的约一半.

P 3.3 摊煎饼

1) 基本原理

循环 n 次, 对于每一次操作, 首先找到第 i 大的煎饼, 记其位置为 pos_i , 在该位置进行一次摊煎饼, 这样会将第 i 大的煎饼放在最顶层, 接着对 $n-i+1$ 位置进行一次摊煎饼, 这样会使得第 i 大的煎饼恰好位于第 $n-i+1$ 层, 即倒数第 i 层. 显然, 经过两次摊煎饼操

作, 第 i 大的煎饼就转移到倒数第 i 个位置. 同时注意到上述操作并不影响第 $n-i+1$ 层以下的煎饼的位置. 因此循环不变量为第 i 次循环开始前, 数组 $A[n-i+2..n]$ 部分已经排列正确.

初始化: 第 1 次循环开始前, 数组 $A[n+1..n]$ 部分为空集, 显然成立.

维持: 假设第 i 次循环开始前, 不变式成立, 则第 i 次循环时, 将第 i 大的煎饼放在 $n-i+1$ 的位置, 于是 $A[n-i+1..n]$ 已经放置正确. 于是第 $i+1$ 次循环开始前, 不变式成立.

终止: 当第 n 次循环结束, 即第 $n+1$ 次循环开始前, 根据不变式, $A[1..n]$ 已经放置正确, 即当算法终止时, 所有煎饼均已按照顺序排列.

2) 时间复杂度分析

因为算法循环 n 次, 每次需要调用 2 次摊煎饼操作, 因此总的操作次数为 $T(n) = 2n = \Theta(n)$.

P 3.4 蛇形数组

基本原理

对数组使用归并排序从小到大排序, 然后依次从左向右和从右向左选择元素排列到蛇形数组中.

伪代码

```
Algorithm SNAKE-SHAPED-ARRAY
Input: B[1..2n+1]
Output: 蛇形数组A[1..2n+1]
sort the array B using merge sort;
int pos1 ← 1, int pos2 ← 2n + 1, int pos3 ← 1
While pos3 ≤ 2n do:
    A[pos3] ← B[pos1], pos1 ← pos1 + 1;
    pos3 ← pos3 + 1;
    A[pos3] ← B[pos2], pos2 ← pos2 + 1;
    pos3 ← pos3 + 1;
    A[pos3] ← B[pos1];
```

算法的正确性比较显然.

P 3.5 PREVIOUS-LARGER 法一 (单调栈)

1) 基本原理

从右往左遍历每个元素，维护一个下标对应元素值自底向上单调递减的单调栈，当栈空时将当前元素下标入栈。否则将当前元素与栈顶元素对应的元素值进行比较，如果当前元素较小则入栈其下标，否则这个元素就是栈顶元素的左边第一个大于它的元素。当所有元素遍历完后，栈中还有未出栈的元素下标，则这些对应的元素的左边第一个大于它的元素下标为 0(不存在)。

2) 时间复杂度分析

对于每个元素，至多入栈和出栈各一次，因此总的时间复杂度为 $O(n)$ 。

P 3.5 PREVIOUS-LARGER 法二 (标记法)

1) 基本原理

从左往右遍历数组，用栈维护一个候选表。当栈空时说明当前元素没有左边大于它的元素。否则，不断比较栈顶的下标对应的元素，如果比当前元素大，就找到了左边第一个大于当前元素的元素，否则，将栈顶元素出栈，不断出栈，直到栈空或找到一个大于当前元素的元素。最后将当前元素的下标入栈。

2) 时间复杂度分析

每个元素最多入栈和出栈各一次，因此时间复杂度为 $O(n)$ 。

P 3.6 对调左右子数组

1) 时间 $O(n^2)$, 空间 $O(1)$

基本原理：采用“平移元素”的思想，每次将一个元素不断与相邻元素交换位置以实现“平移”。每个元素移动到目标位置的时间复杂度为 $O(n)$ ，故总的时间复杂度为 $O(n^2)$ 。

2) 时间 $O(n)$, 空间 $O(n)$

基本原理：准备一段额外的交换数组，用于临时存放左半边的元素，然后将右半边的元素直接覆盖左半边的元素，最后用临时数组中的左半边元素覆盖右半边元素。(如果原数组中右半边元素个数较多，则将左右数组对调)

3) 时间 $O(n)$, 空间 $O(1)$

基本原理

利用 $T(n) = T(n/2) + O(n) \Rightarrow T(n) = O(n)$, 递归完成交换.

伪代码

待完成.

注：另一种更简单的方法

显然可以设计一个 $\Theta(n)$ 的使得整个数组翻转的算法 REVERSE(A, l, r);
因此只需要调用三次该算法: REVERSE(A, 1, k); REVERSE(A, k+1, n); REVERSE(A, 1, n);

P 3.7 颠倒所有单词的顺序

P 3.8 微博名人问题

1)

答: 至多有 1 个名人. 反证法: 假设存在 2 个名人 A 和 B, 则 A 被 B 关注, B 被 A 关注, 这与 A,B 均不关注任何人矛盾. 于是假设不成立. 而存在 1 个名人和不存在名人的例子容易举出.

2)

基本原理

定义集合 S 为所有可能的名人候选人的集合. 初始时 S 中包含所有人. 当 S 中元素个数大于 1 时, 不断选择 S 中的两个候选人 A 和 B, 询问 A 是否关注 B. 如果回答为 YES, 则 A 不可能是名人, 否则 B 不可能是名人. 因此无论答案如何, 一定能使得 S 的大小-1. 因此在进行 $n-1$ 次查询后, S 的大小为 1. 此时花费 $O(n)$ 次查询判断该候选人是否是名人即可.

伪代码

时间复杂度分析

根据基本原理的分析, 算法的时间复杂度为 $O(n)$.

P 3.9 最大和连续子序列

1) $O(n^3)$ 蛮力算法

基本原理: 枚举所有的起点和终点, 一共有 $\binom{n}{2}$ 中可能, 而每种可能需要花费 $O(n)$ 的代价求和, 因此总的时间复杂度为 $O(n^3)$.

2) $O(n^2)$ 优化遍历

基本原理: 遍历 n 次数组, 每次的起点依次为 $1, 2, \dots, n$, 终点总是 n . 每次遍历中, 维护累加的总和并尝试更新最大值.

第 1 次遍历会计算 $(1, 1), (1, 2), \dots, (1, n)$ 的和, 第 k 次会计算 $(k, k), (k, k+1), \dots, (k, n)$ 的和. 因此算法终止时, 已经计算出了所有的 pair 的和.

3) $O(n \log n)$ 分治

基本原理: 利用递归式 $T(n) = 2T(n/2) + O(n)$. 递归求解左右半边子数组的最大连续子序列, 并以 $O(n)$ 的时间 Fix 跨越左右子数组部分的最大连续子序列.

4) $O(n)$ 优化遍历

5) $O(n)$ DP

$dp[i]$ 状态维护数组 $A[1..i]$ 以 $A[i]$ 结尾的最大和连续子序列.

$$dp[i] = \begin{cases} dp[i-1] + A[i] & dp[i-1] > 0, \\ A[i] & dp[i-1] \leq 0. \end{cases}$$

从左向右遍历并计算 $dp[i]$, 同时尝试更新最大和.

Chapt. 4 分治排序

P 4.1

法一

证明. 注意到 $n_0 + n_1 + n_2 = n \leq 2^{h+1} - 1 \Rightarrow n_0 + n_2 \leq 2^{h+1} - 1$, 而 $n_1 + 2n_2 = n_0 + n_1 + n_2 - 1 \Rightarrow n_2 = n_0 - 1$, 因此 $n_0 + n_2 = 2n_0 - 1 \leq 2^{h+1} - 1 \Rightarrow n_0 \leq 2^h$. \square

法二

首先证明引理: 对于任意高度为 h 的二叉树, 满二叉树的叶子节点最多.

Pf. 反证法, 若不然, 则说明存在一棵高度为 h 且不是满二叉树的树, 拥有更多的节点.

注意到, 对于任意高度为 h 的二叉树都可以视作由满二叉树逐渐删去叶子结点得到. 考察删除叶子节点的父节点, 若其只有一个子节点, 则在删除后, 叶子节点的数目不变; 若其有两个子节点, 则在删除该叶子节点后, 叶子节点的数目减少, 而对满二叉树删除的第一个叶子节点必然使得总叶子节点数-1. 因此对于任意高度为 h 的二叉树, 其叶子节点数总是小于满二叉树的叶子节点.

因此对于高度为 h 的二叉树, 其叶子节点数 $n <$ 高度为 h 的满二叉树的叶子节点 $n' = 2^h$. \square

(高度为 h 的满二叉树的叶子节点数为 2^h , 这个定理是容易通过数学归纳法证明的.)

P 4.4

1)

Algorithm 4: 4-ELEMENT-SORT(a,b,c,d)

output: 4 个元素的降序

```
1 if a < b then swap (a,b);
2 if c < d then swap (c,d);
3 if a < c then swap (a,c);
4 if b ≥ c then
5   return ;
6 else
7   swap (b,c);
8   /* 这时候需要比较 c,d */
9   if c < d then swap (c,d);
  return ;
```

2)

首先给出一个至多需要比较 7 次的算法, 然后证明不存在比较次数少于 7 次的排序算法.

由于本人比较笨, 没能独立思考得到算法, 在 StackOverflow 上找到了一种算法, 引用如下:

注：7 次比较得到全序的算法

Compare A to B and C to D. WLOG, suppose A>B and C>D. Compare A to C.

WLOG, suppose A>C. Sort E into A-C-D. This can be done with two comparisons.

Sort B into E,C,D. This can be done with two comparisons, for a total of seven.

来源: <https://stackoverflow.com/a/1534932/18494159>

下面证明无法通过少于 7 次的比较得到 5 个元素的全序. 首先证明 6 次比较不能得到全序.

Pf. 由于 5 个元素的排列有 $5! = 120$ 种, 而考虑每次比较形成的二叉树, 因为每一次比较可能产生两种结果 ($a < b$ OR $a \geq b$), 因此每一次比较产生两个子节点, 6 次比较后得到一棵高度为 6 的满二叉树, 产生的结果必然在叶节点上, 由 P4.1 证明, 高度为 6 的二叉树的叶节点数不超过 $2^6 = 64$ 个叶子节点, 因此不能区分 120 种排列. 对于比较次数少于 6 次的情况, 显然更加不能满足. \square

P 4.7

P 4.8 *k-sorted* 问题

大致思路

递归地解决这个问题. 由于题设, 可以假设 n 是 k 的倍数, 且 n, k 均为 2 的幂. 每次将数组等分成两部分, 即 $A[1..n/2], A[n/2 + 1..n]$, 且满足左半边的元素都小于右半边的元素. 这个 partition 过程用到了 $O(n)$ 时间选择中位数的算法, 找到中位数的下标后再调用 partition 函数, 以 $O(n)$ 的时间复杂度将数组等分成 2-sorted. 递归调用这个过程, 直到每段元素的个数为 $\frac{n}{k}$, 直接返回即可.

注：更广泛的情况

如果不要求 k 是 2 的幂, 算法稍作改动即可, 即令 k' 为不小于 k 的最小的 2 的幂次, 相应的, 令 $n' = k' \times \frac{n}{k}$, 然后, 在原先的输入中添加 $n' - n$ 个 $+\infty$, 然后运行本算法即可. 因为 $k' = \Theta(k)$, 因此算法的时间复杂度仍然是 $O(n \log k)$.

举例来说, 如果输入的 $n = 9, k = 3$, 则令 $k' = 4$, 并添加 3 个 $+\infty$, 在算法运行时, 第一次划分得到 1..6, 7..12, 第二次划分得到 1..3, 4..6, 7..9, 10..12. 显然, 前 k 组就是正确的划分.

伪代码

Algorithm 5: K-SORTING(A[1..n], k)

input :一个乱序数组 A[1..n], k 是要分成的段数

output: 将原数组均分成 k 段, 每段子数组均满足所有元素小于右边段的元素

```
1 2-PARTITION(A, 1, n, k, n);
2 return ;
3 2-PARTITION(A[1..n], l, r, k, n) begin
4   if r - l + 1 = n / k then
5     return ;
6   m = FIND-MEDIAN (A, l, r)/* 找到这段数组的中位数的位置 */
7   SWAP (A[m], A[r]);
8   /* 为 partition 做准备 */
9   PARTITION (A, l, r);
10  2-PARTITION(A, l, m-1, k, n);
11  2-PARTITION(A, m, r, k, n);
```

复杂度分析

由于可以做到在 $O(n)$ 时间内将数组等分为两段, 且满足左边的元素总是小于右边的元素, 因此可以写出如下的递归式:

$$T(n) = 2T(n/2) + \Theta(n)$$

由主定理, case2, $T(n) = \Theta(n \log k)$ (递归树的高度是 $\log k$).

P 4.9 螺钉螺母配对问题

1) 大致思路

首先将所有的螺钉和螺母分别排成一列. 然后逐个取螺母进行尝试.

第一次, 将所有的螺钉与该螺母进行比较, 将螺钉小于螺母的一堆排在左边, 大于螺母的排在右边, 中间是等于螺母的. 然后再将所有的螺母与中间的螺钉进行比较, 同样分为大于该螺钉的 (和大的螺钉对应) 和小于该螺钉的 (与小的螺钉对应).

于是, 将未配对的螺钉和螺母都分成了两段. 第二次, 分别从小的螺母堆和大的螺母堆中取一个螺母, 重复上述过程, 最后得到 4 堆螺钉和螺母. 重复上述过程, 直到最后每堆只剩下一个螺钉和一个螺母, 则必然是配对的.

2) 伪代码

Algorithm 6: SCREWS-AND-NUTS

input :一堆乱序的大小不同螺钉 $S[1..n]$ 和一堆与之对应的螺母 $N[1..n]$
output: 配对的螺钉和螺母

```
1 SORT(S, N, 1, n);
2 return ;
3 SORT(S, N, l, r) begin
4     if  $l \geq r$  then
5         return; /* 只剩 1 个或 0 个，必然配对 */ 
6         m = SCAN-THE-SCREWS (S, N, l, r); /* 找到匹配 N[1] 的螺钉的下标，并
7             将 S 数组按照 S[m] 进行划分 */ 
8         SCAN-THE-NUTS (S[m], N, l, r); /* 扫描所有剩余的螺母，将 N 数组按照
9             S[m] 进行划分 */ 
10        SORT(S, N, l, m-1);
11        SORT(S, N, m+1, r);
```

3) 复杂度分析

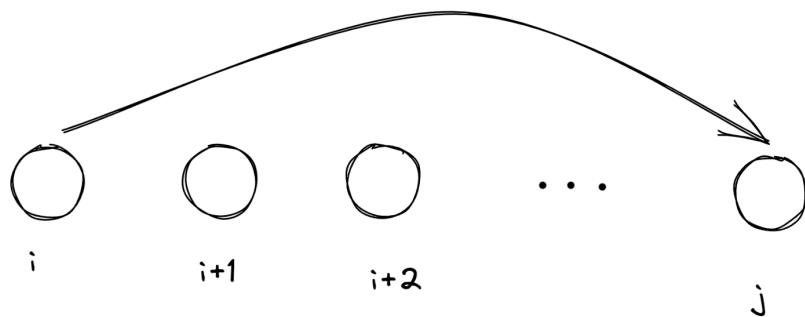
该算法相比 QUICK-SORT 算法, 只是在 partition 部分多了一步 $O(n)$ 时间对螺母数组进行 PARTITION, 因此递归式同样是 $T(n) = T(i) + T(n - i - 1) + O(n)$, 因此利用 QUICK-SORT 分析的结论, 最坏时间复杂度为 $O(n^2)$, 平均时间复杂度为 $O(n \log n)$.

P 4.11

1)

A)

反证法, 即存在逆序对 $(i, j), s.t. j - i \geq 3$, 即 i, j 之间至少还有两个元素 $A[i + 1], A[i + 2]$,



考虑 $A[i+1]$,

- $A[i + 1] \geq A[i] \Rightarrow (i + 1, j)$ 构成一个逆序对
- $A[i + 1] < A[i] \Rightarrow (i, i + 1)$ 构成一个逆序对

同理, 对于 $A[i+2]$, 至少有一个与 $i+2$ 有关的逆序对.

因此至少存在3个逆序对, 这与至多存在2个逆序对矛盾.

故原命题成立.

2)

首先, 对一些性质的观察

- 如果存在 $j - i = 2$, 则两个逆序对必然在 $A[i..j]$ 之间. 且必然在 $A[i..i+1]$ 或者 $A[i+1..j]$ 有存在一个逆序对.

算法设计

1. 遍历数组, 进行 $n-1$ 次相邻元素之间的比较.
 1. 如果找到两个逆序对, 则算法结束
 1. 否则必然找到一个逆序对 $(i, i+1)$
 2. 比较 $(i-1, i+1)$
 1. 如果 $A[i-1] > A[i+1]$ 那么算法直接终止
 2. 否则, 排除了这个可能的逆序对, 同时用掉了多于的 1 次机会
 3. 比较 $(i, i+2)$
 1. 如果 $A[i] > A[i+2]$ 那么算法终止
 2. 否则, 说明 $A[i+2] > A[i] > A[i+1]$, 因此不需要比较 $A[i+1], A[i+2]$ 的关系.(相当于将原本 $(i+1, i+2)$ 的比较换成了 $(i, i+2)$ 的比较) 同时也说明了这个逆序对周围不存在逆序对, 因此接下来如果找到另一个相邻逆序对, 算法直接终止即可.

对于上述算法, 只会在第一次遇到相邻逆序对时多比较一次, 因此比较次数不超过 $n-1 + 1 = n$ 次.

P 4.14 易位词

1) 第一种算法

这种算法是我自己想的.

算法原理

对于每一个单词, 统计其每个字母的出现次数作为一个集合, 这样的一个集合作为一个元素, 用 Hash 表维护这样的映射关系, 即统计相同元素个数, 最后统计 Hash 表中元素出现次数大于等于 2 的元素, 这些元素即为“易位词”.

时间复杂度分析

假设对每个单词的操作为 $O(1)$, 由于 Hash 的存储和查找的平均时间复杂度都是 $O(1)$, 因此可以做到一趟记录完成统计, 最后再遍历整个 hash 表, 统计所有出现次数大于等于 2 的元素个数即可. 因此时间复杂度为 $O(n)$.

2) 第二种算法

这种算法是老师上课讲的基于排序的算法.

算法原理

首先可以做一个预处理: 由于长度不同的单次必然不是“易位词”, 因此首先按照长度对元素进行排序.

然后对每一个元素定义其 ID: 将该单词的字母按照字典序排序后的字符串的编码, 显然, 两个单词构成易位词当且仅当其具有相同的 ID, 因此按照 ID 对元素进行排序, 最后遍历并统计 ID 相同的元素个数即可.

时间复杂度分析

算法基于排序, 因此时间复杂度的下界是 $O(n \log n)$, 这个算法的核心就是排序, 因此时间复杂度是 $O(n \log n)$ 的.

P 4.15 空闲时间

基本原理

将所有的任务按照任务的开始时间升序 (开始时间相同的按照结束时间升序), 然后迭代所有任务, 每次判断是否能够与之前的任务合并, 如果能, 则尝试更新最长非空闲时间, 否则尝试更新最长空闲时间.

伪代码

Algorithm 7: FREETIME

Input : task[1..n], 记录每个任务的开始时间和结束时间

Output: 最长空闲时间 Tf, 最长非空闲时间 Tnf

```
1 初始化:  $Tf := 0$ ,  $Tnf := 0$ ,  $pstart$ ,  $pend$  记录上一个连续工作时间的开始和  
    结束时间;  
2 sort task[1..n] into increasing seq by 1. start 2. end;  
3  $pstart := \text{task}[1].start$ ,  $pend := \text{task}[1].end$ ;  
    /* 尝试更新最长时间 */  
4 if  $pend - pstart > Tnf$  then  $Tnf := pend - pstart$ ;;  
5 for  $i := 2$  to  $n$  do  
6     if  $\text{task}[i].start \leq pend$  then  
        /* 可以和之前的连起来 */  
         $pend := \max(pend, \text{task}[i].end)$ ;  
        if  $pend - pstart > Tnf$  then  $Tnf := pend - pstart$ ;  
9     else  
        /* 否则尝试更新最长空闲时间 */  
10    if  $\text{task}[i].start - pend > Tf$  then  $Tf := \text{task}[i].start - pend$ ;  
        /* 同时更新 pstart, pend */  
11     $pstart := \text{task}[i].start$ ;  
12     $pend := \text{task}[i].end$ ;  
13    if  $pend - pstart > Tnf$  then  $Tnf := pend - pstart$ ;
```

算法正确性证明

即证明, 该算法能够找到最长空闲时间和最长非空闲时间.

证明. 显然, 对所有的任务进行排序不会影响结果. 经过排序之后可以保证, 后面的任务的开始时间一定大于前面的任务, 且相同开始时间的任务的结束时间是升序的. 定义任务串为一串可以合并的任务. 显然, 问题等价于将所有任务合并成一组极大任务串后, 从极大任务串中寻找最大非空闲时间, 从极大任务串的间隙中寻找最大空闲时间.

循环不变量 每次迭代开始前, Tf 为 task[1..i-1] 得出的最长空闲时间, Tnf 为 task[1..i-1] 得出的最长非空闲时间. 且 pstart - pend 记录了包含 task[i-1] 的极大任务串.

初始化 $i = 2$, $Tf = 0$, $Tnf = \text{task}[1].end - \text{task}[1].start$, 显然是最长的. 且此时 $pstart = \text{task}[1].start$, $pend = \text{task}[1].end$, 显然极大任务串.

维持 假设 $i = k$ 前, T_f 为 $\text{task}[1..k-1]$ 得出的最长空闲时间, T_{nf} 为 $\text{task}[1..k-1]$ 得出的最长非空闲时间, 则

- 若 $\text{task}[k].start \leq \text{pend}$, 则说明当前的任务可以和其前面的任务串合并, 因此可以更新当前任务串, 也保证得到了包含 $\text{task}[k]$ 的极大任务串, 如果新的任务串具有更长的非空闲时间, 则更新后的 T_{nf} 是 $\text{task}[1..k]$ 的最长的非空闲时间. 而 $\text{task}[1..k]$ 相比 $\text{task}[1..k-1]$ 并没有增加空闲时间的段数, 因此对 T_f 无影响.
- 若 $\text{task}[k].start > \text{pend}$, 说明当前的任务和其之前的任务串是不连续的, 由于排序的性质, 后面所有的任务的开始时间也一定大于 pend , 即当前的任务串不可能继续延长, 因此包含 $\text{task}[k]$ 的极大任务串只能是其本身, 同时注意到产生了一段新的空闲时间 (因为后面的任务也不可能填补这段时间, 即为极大任务串的间隙), 因此尝试更新最大空闲时间, 保证 T_f 是当前最大的. 也同时更新 T_{nf} , 保证 T_{nf} 是最大的.

终止 因此当 $i = n+1$ 时, T_f 为 $\text{task}[1..n]$ 得出的最长空闲时间, T_{nf} 为 $\text{task}[1..n]$ 得出的最长非空闲时间.

□

注

感觉还是说的比较混乱.

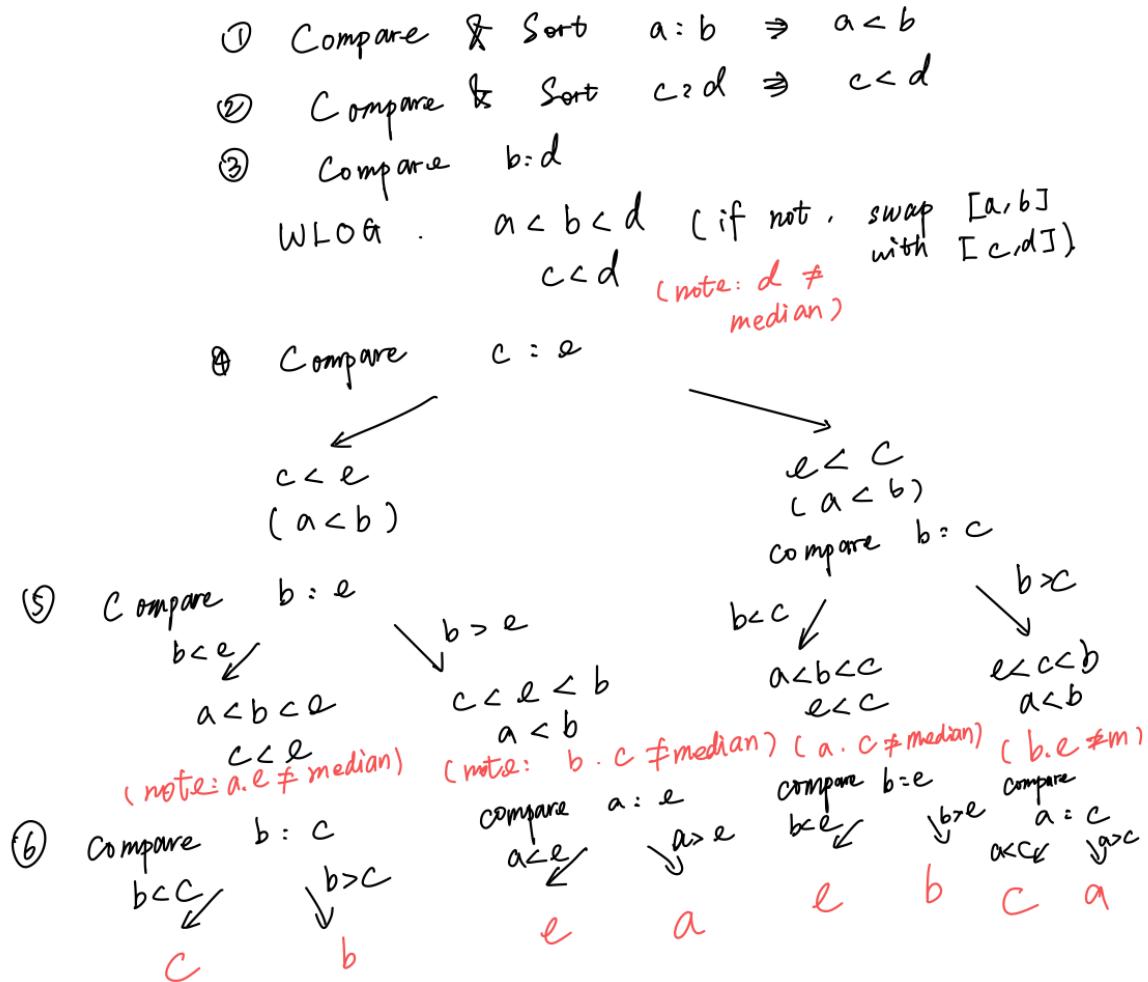
时间复杂度分析

算法分为两个部分: 排序部分如果使用堆排序/归并排序, 时间复杂度为 $O(n \log n)$, 迭代部分容易得出是 $O(n)$, 因此总的时间复杂度为 $O(n \log n)$.

Chapt. 5 线性时间选择

P 5.2 6 次找出 5 个元素的中位数

算法描述和决策树见下图.



注

由于伪代码写起来比较复杂, 且没有画图清晰, 因此就不写伪代码了.

参考资料: <https://cs.stackexchange.com/a/45379>

大致思路就是上来 2 次要充分比较, 然后后面使用排除法.

P 5.4

注

没太懂题目意思.

证明. 该算法能够通过比较找到阶为 i 的元素, 等价于该算法通过比较直接或间接确定该元素有 $i-1$ 个较小的元素和 $n-i$ 个较大的元素, 因此原命题成立. \square

P 5.5

由于题 5.5, 对于一个基于比较的寻找阶为 i 的算法, 其也能够分别找到阶小于 i 和阶大于 i 的所有元素. 因此可以得到基于算法 A 的如下算法 B:

Algorithm 8: B

Input : $A[1..n]$, k
Output: the k th element in ordered array of A

```
1 return B-recursive( $A$ ,  $k$ );  
2 B-recursive( $A[1..n]$ ,  $k$ ) begin  
    /* 找到在  $A[1..n]$  中第  $k$  小的元素 */  
    3   if  $n = 1$  then return  $A[1]$ ;  
    4   if  $k \leq n/2$  then  
        5       Arr = find the elements less than median using  $A$ ;  
        6       return B-recursive(Arr,  $k$ );  
    7   else  
        8       Arr = find the elements larger than median using  $A$ ;  
        9       return B-recursive(Arr,  $k - n/2$ );
```

P 5.6 最大的 k 个数

1) $O(n \log n)$

算法原理: 先使用堆排序对所有元素进行排序, 这个过程的最坏时间复杂度为 $O(n \log n)$, 然后迭代 k 次选出前 k 大.

算法最坏时间复杂度为 $O(n \log n + k) = O(n \log n)$.

2) $O(n + k \log n)$

算法原理: 使用堆排序. 首先以 $O(n)$ 的时间复杂度将数组转化为最大堆. 然后迭代 k 次, 选出第 i 大, 并 Fix-heap, 时间复杂度为 $O(k \times \log n) = O(k \log n)$, 总的时间复杂度为 $O(n + k \log n)$.

3) $O(n + k^2)$ 法一: 堆

算法原理: 使用堆排序, 首先以 $O(n)$ 的时间复杂度将数组转化为最大堆. 注意到一个性质: 只有到根节点的距离小于等于 $k-1$ 的元素才有可能称为前 k 大, 因此只取堆中前 $2^k - 1$ 个元素作为一个新的堆, 后面对堆的维护也只考虑这部分元素, 因此取出前 k 大元素的时间复杂度为 $O(k \times \log(2^k)) = O(k^2)$.

总的最坏时间复杂度为 $O(n + k^2)$.

4) $O(n + k^2)$ 法二：排序

算法原理: $O(n)$ 时间选出第 k 大元素, 最前 k 大元素使用 $O(k^2)$ 的排序 (冒泡, 插排).

5) $O(n + k \log k)$ 法一：堆

注：参见练习 14.2

下面是对 P14.2 的解答的引用.

参考资料:<https://www.geeksforgeeks.org/k-th-greatest-element-in-a-max-heap/>

Algorithm 9: K-LARGEST(Heap h, k)

output: the k th largest element in the heap

```
1 初始化: Heap temp = null; /* 一个空的大顶堆 */  
2 temp.insert(h.top);  
3 for i ← 1 to k - 1 do  
4     int cur = temp.top;  
5     temp.pop;  
6     if cur.left ≠ null then  
7         temp.insert(cur.left);  
8     if cur.right ≠ null then  
9         temp.insert(cur.right);  
10    return temp.top;
```

算法正确性分析

首先证明, 每次 pop 出的元素依次为第 $1, 2, 3, \dots, k-1$ 大的元素.

由于堆能够维护最大的元素总是在 top 的性质, 只要说明, 在第 i 次 pop 前, 第 i 大的元素总是在 temp 堆中即可.

初始化 第一次 pop 前, temp 中只有 $h.\text{top}$, 由 h 堆的性质, $h.\text{top}$ 是第 1 大元素.

维持 对于第 m 次前的每一次 pop , 都弹出第 i 大的元素, 假设第 m 次弹出前, 第 m 大的元素不在 temp 堆中, 即, 第 m 大的元素的父节点不是第 $1, 2, \dots, m-1$ 大的元素, 那么第 m 大的元素的父节点至少是第 m 大的元素, 由堆的偏序性, 该元素至少是第 $m+1$ 大的元素, 这就构成矛盾. 因此第 m 次 pop 前, 第 m 大的元素一定在 temp 堆中.

终止 在第 k 次 pop 前, 第 k 大的元素一定在堆中.

注：说明

这里需要指出，”在堆中”是指至少已经进入过 temp 堆了，那么有没有可能在第 k 次 pop 之前，第 k 大的元素已经被 pop 出去了呢？显然是不可能的，由上述论证， $\forall i$, 第 i 次 pop 前，第 i 大的元素也已经在堆中，由堆的性质，pop 出去的一定是当前堆中的最大元素，因此可以归纳得到每一次 pop 严格弹出第 i 大的元素。

因此，在执行 $k-1$ 次 pop 之后（即第 k 次 pop 之前），temp 堆的 top 元素就是第 k 大的元素。□

时间复杂度分析

每次 Fix-heap 的时间复杂度为 $O(\log k)$ ，一共执行 $k-1$ 次，总的时间复杂度为 $O(k \log k)$ 。

因此，总的最坏时间复杂度为 $O(n + k \log k)$ 。

6) $O(n + k \log k)$ 法二：排序

$O(n)$ 时间找到第 k 大，对前 k 部分进行 SORT。

P 5.7 最接近中位数的 k 个数

1) $O(n \log n + k)$

Algorithm 10: K-NEAREST-MEDIAN($A[1..n]$, k)

Output: k nearest elements of the median

```
1 初始化: ret 是返回的元素的数组, 初始为空 Heap-Sort (A);
2 int m := (1 + n) / 2;
3 int l := m - 1;
4 int r := m + 1;
5 if n is even then
6   r: r + 1;
/* l,r 分别指示左边和右边最接近中位数元素的元素 */
7 int cnt:= 0;
8 while cnt < n do
9   if l > 0 且 r ≤ n then
10    if A[l] 比 A[r] 更接近 A[M] then
11      ret.insert(A[l]);
12      l:= l-1;
13    else
14      ret.insert(A[r]);
15      r: r+1;
16  else if l > 0 then
17    ret.insert(A[l]);
18    l: l-1;
19  else
20    ret.insert(A[r]);
21    r: r+1;
22  cnt := cnt +1;
23 return ret;
```

时间复杂度分析

首先堆排序的最坏时间复杂度是 $O(n \log n)$, 对于 while 循环里面的操作, 移动一次 l, r 指标一次, cnt 就 $+1$, 直到 $cnt=k$ 结束, 因此 while 循环部分的时间复杂度是 $O(k)$ 的, 因此总的时间复杂度为 $O(n \log n + k)$.

2) $O(n + k \log k)$

基本原理

首先以 $O(n)$ 的时间复杂度通过 SELECTION 算法找到数组的中位数值 M , 然后再以 $O(n)$ 的时间复杂度对数组进行预处理: $A[i] = \text{abs}(A[i] - M)$ (可以将中位数的新值设置为 $+\infty$, 防止后面对堆的影响). 得到预处理的数组以 $O(n)$ 的时间复杂度转化为小顶堆, 然后应用练习 5.6(4) 的算法, 以 $O(k \log k)$ 的时间复杂度选出前 k 小的元素, 这些元素即为最接近 M 的 k 个元素.

时间复杂度分析

根据基本原理, 容易得出算法的时间复杂度为 $O(n) + O(n) + O(n) + O(k \log k) = O(n + k \log k)$.

3) $O(n + k \log k)$ 法二 (课上介绍) $SEL \oplus PAR$

基本原理

利用线性时间选择选法, 首先找到 rank 为 $n/2 - k, n/2 + k$ 的元素, 然后利用 PARTITION 算法, 将元素进行划分, 并舍弃左右两半边的不可能的元素, 将剩余元素缩减到 $O(k)$ 的规模.

对于剩余元素使用堆排序即可.

P 5.9 多个数组的第 k 大元素

注：假设

以下问题均假设所有元素均不相同.

1) 两个有序数组

参考资料: <https://leetcode.cn/problems/median-of-two-sorted-arrays/solution/xiang-xi-tong-su-de-si-lu-fen-xi-duo-jie-fa-by-w-2/>

基本原理

用 pos1 , pos2 分别指示 A , B 数组的某个元素, 当 $k \geq 2$ 且数组均不为空时, 每次选择 $\text{pos1} = \min\{A.\text{len}, \lfloor \frac{k}{2} \rfloor\}$, $\text{pos2} = \min\{B.\text{len}, \lfloor \frac{k}{2} \rfloor\}$. 比较 $A[\text{pos1}]$, $B[\text{pos2}]$, 将较小者及其前面所有元素删除, 因为这些元素的阶 $\leq \lfloor \frac{k}{2} \rfloor + \leq \lfloor \frac{k}{2} \rfloor - 1 < k$. 不断重复上述过程,

直到某个数组为空或 $k = 1$. 如果某个数组为空, 则直接返回另一个数组的第 k 个元素; 若 $k = 1$, 则直接比较两个数组的第一个元素返回较小者即可.

伪代码

Algorithm 11: FIND-K-2

Input : $A[1..n], B[1..n]$

Output: the k th element in ordered array of $A \cup B$

```

1 while  $k \geq 2 \wedge A, B$  均非空 do
2    $pos1 = \min\{A.len, \lfloor \frac{k}{2} \rfloor\}, pos2 = \min\{B.len, \lfloor \frac{k}{2} \rfloor\};$ 
3   if  $A[pos1] < B[pos2]$  then
4      $A \leftarrow A - A[1..pos1];$ 
5      $k \leftarrow k - pos1;$ 
6   else
7      $B \leftarrow B - B[1..pos2];$ 
8      $k \leftarrow k - pos2;$ 
9 if  $A$  为空 then
10   return  $B[k];$ 
11 else if  $B$  为空 then
12   return  $A[k];$ 
13 else
14   /*  $k = 1$  */  

    return  $\min(A[1], B[1]);$ 

```

时间复杂度分析

如果提前遇到某个数组为空的情况, 则算法提前终止. 考虑最坏情况, 即两个数组始终不为空直到 $k=1$. 则每次迭代使得 $k \rightarrow k - \lfloor \frac{k}{2} \rfloor$, 因此迭代的次数为 $O(\log k)$. 因此算法的时间复杂度是 $O(\log k)$. 又因为 $1 \leq k \leq 2n$, 所以 $O(\log k) = O(\log n)$.

2) 三个有序数组

基本原理

类似第 (1) 问, 用 $pos1, pos2, pos3$ 来标记 A, B, C 数组的下标. 当 $k \leq 3$ 且数组均不为空时, 每次选择 $pos1 = \min\{A.len, \lfloor \frac{k}{3} \rfloor\}, pos2 = \min\{B.len, \lfloor \frac{k}{3} \rfloor\}, pos3 = \min\{C.len, \lfloor \frac{k}{3} \rfloor\}$. 比较 $A[pos1], B[pos2], C[pos3]$ 中的最小元素, 删除该元素及其左边的

所有元素. 同时更新 k . 当出现 1 个数组为空时, 直接调用第 (1) 问的算法. 当 $k \leq 2$ 时, 可以在 $O(1)$ 时间内比较 A, B, C 的前 2 个元素, 得到结果.

伪代码

Algorithm 12: FIND-K-3

Input : $A[1..n], B[1..n], C[1..n]$

Output: the k th element in ordered array of $A \cup B \cup C$

```

1 while  $k \geq 3 \wedge A, B, C$  均非空 do
2    $pos1 = \min\{A.len, \lfloor \frac{k}{3} \rfloor\}, pos2 = \min\{B.len, \lfloor \frac{k}{3} \rfloor\}, pos3 = \min\{C.len, \lfloor \frac{k}{3} \rfloor\};$ 
3   compare  $A[pos1], B[pos2], C[pos3]$ . WLOG, call  $X[pos]$  is the min num.  $X \leftarrow X - X[1..pos];$ 
4    $k \leftarrow k - pos;$ 
5 if 存在某个数组为空 then
6    $\quad$  return FIND-K-2();
7 else
8   if  $k = 1$  then return  $\min(A[1], B[1], C[1]);$ 
9   else
10     $\quad$  /*  $k = 2$  */  

11     $\quad$  return the second smallest num in  $A, B, C;$ 

```

时间复杂度分析

类似第 (1) 问, 最坏情况下, 每次迭代, $k \rightarrow k - \lfloor \frac{k}{3} \rfloor \approx \frac{2}{3}k$. 因此迭代的次数为 $O(\log k)$. 总的时间复杂度为 $O(\log k) = O(\log n)$.

3) m 个有序数组

基本原理

用 $pos[1..m]$ 来指示 $A[1..m]$ 数组的某个元素, 令 $pos[i] = \min\{A[i].len, \lfloor \frac{k}{m} \rfloor\}$. 当 $k \geq m$ 时, 用 $O(m)$ 的时间找到最小元素, 删掉该元素及其左边的所有元素. 若过程中出现某个数组为空, 则转化为问题规模-1, 即 $A[1..m-1][1..n']$ 的问题. 直至最终只剩一个数组或 $k < m$. 如果只剩一个数组, 则直接返回其第 k 大. 若 $k < m$, 则可以用 $O(mk) = o(m^2)$ 的时间复杂度 (归并排序的前 k 次) 选择第 k 大的元素.

时间复杂度分析

每次迭代中, 计算 pos 及找到最小元素的复杂度为 $O(m)$. 每次迭代将 k 变成原先的 $\frac{m-1}{m}k$, 因此迭代的次数为 $O(\log_{\frac{m}{m-1}} k)$. 迭代完成后, 最坏情况下还需要 $O(m^2)$ 的时间来处理剩余的元素, 所以总的时间复杂度为 $O(m \log_{\frac{m}{m-1}} k + m^2)$.

P 5.10

1)

当 $w_i = \frac{1}{n}$ 时, 加权中位数的定位可以写成 $\sum_{x_i < x_k} 1 < \frac{n}{2}$, $\sum_{x_i > x_k} 1 \leq \frac{n}{2}$, 即小于该元素的元素个数小于所有元素个数的一半, 大于该元素的元素个数不超过所有元素个数的一半, 即中位数的定义.

2)

基本原理

首先使用归并排序将所有元素升序排序, 然后从左到右遍历所有元素, 维护 $\sum_{x_i < x_k} w_i$, 利用 $\sum x_i > x_k w_i = 1 - \sum_{x_i < x_k} w_i - x_k$, 因此可以在 $O(n)$ 的时间内判断每一个元素是否是“加权中位数”.

该算法的伪代码和时间复杂度分析略.

3)

基本原理

首先使用线性时间选择算法找到当前数组的中位数, 然后以中位数为 pivot 使用 PARTITION 算法对数组进行划分, 然后用 $O(n)$ 的时间计算中位数是否是加权中位数,

- $\sum_{x_i < x_k} w_i < \frac{1}{2}, \sum_{x_i > x_k} w_i \leq \frac{1}{2}$. 则说明中位数就是加权中位数, 直接返回;
- $\sum_{x_i < x_k} w_i > \frac{1}{2}, \sum_{x_i > x_k} w_i < \frac{1}{2}$. 则说明加权中位数在中位数的左半边, 递归调用该算法对左半边元素进行查找;
- $\sum_{x_i < x_k} w_i < \frac{1}{2}, \sum_{x_i > x_k} w_i > \frac{1}{2}$. 则说明加权中位数在中位数的右半边, 递归调用该算法对右半边元素进行查找;

伪代码

Algorithm 13: WEIGHTED-MEDIAN(A[1..n], W[1..n])

Output: the weighted median of A

```
1 return WEIGHTED-MEDIAN-R(A, W, 0);
2 WEIGHTED-MEDIAN-R(A[l..r], W[l..r], lsum) begin
3     int ls := lsum;
4     int pivot = MEDIAN (A[l..r]); /* 找到该数组的中位数 */
5     A := PART (A[l..r], pivot);
6     for i := l to pivot do
7         ls := ls + W[i];
8     int rs := 1 - ls - W[pivot];
9     if ls <  $\frac{1}{2}$   $\wedge$  rs  $\leq \frac{1}{2}$  then
10        return pivot;
11    else if ls >  $\frac{1}{2}$  then
12        // 说明在 pivot 的左半边
13        return WEIGHT-MEDIAN-R(A[l..pivot-1], W[l..pivot-1], lsum);
14    else
15        // 说明在右半边，需要更新 lsum
16        return WEIGHT-MEDIAN-R(A[pivot+1..r], W[pivot+1..r], ls);
```

时间复杂度分析

写出递归式 $T(n) = T(n/2) + O(n)$, 由主定理, $T(n) = \Theta(n)$.

Chapt. 6 对数时间查找

P 6.2

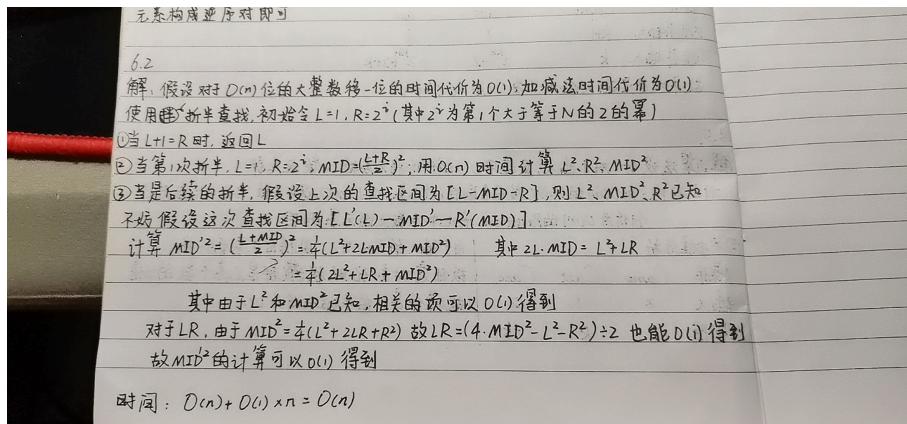


图 3: 陈同学提供的方法, 感谢

P 6.3 证明 RBT 的两种定义等价

1) 直接定义 \Rightarrow 递归定义

注

还不会.

2) 递归定义 \Rightarrow 直接定义

a)

根节点为黑色, 所有外部节点为黑色.

利用数学归纳法, 对黑色高度 h 做归纳. 记 $P(k)$: RB_k 的根节点为黑色, 其所有黑色节点为黑色.

Basis. $P(0)$: 根据递归定义的 basis, 显然满足.

I.H. 假设 $P(k)$ 满足条件.

I.S. 对于 $h = k+1$, 根据递归定义, 其根节点为黑色. 且其左右子树分别为一棵 RB_k 或 ARB_{k+1} . 又由 ARB 的递归定义, ARB_{k+1} 的左右子树均为 RB_k . 由归纳假设, RB_k 的所有外部节点均为黑色, 而 RB_{k+1} 的所有外部节点即 RB_k 的所有外部节点. 因此 RB_{k+1} 的所有外部节点均为黑色.

综上, 对于任意的 $k \geq 0$, $P(k)$ 成立. 原命题成立.

b)

红色节点不能连续出现, 即没有任何一个红色父节点有红色子节点.

用数学归纳法, 对黑色高度进行归纳, 记 $P(k)$: RB_k, ARB_{k+1} 中不存在有红色子节点的父节点.

Basis. $P(0)$: 由 RBT 的归纳奠基, 显然 RB_0 满足. 对于 ARB_1 , 由递归定义, 其根节点为红色, 左右子树中不存在, 且根节点与其左右子节点间也不满足, 因此成立.

I.H. 假设 $P(k)$ 成立.

I.S. 对于 $h = k+1$, 由 RBT 的递归定义, 其根节点为黑色, 且左右子树为 RB_k 或 $ARB_k + 1$, 由归纳假设, 其左右子树中不存在有红色子节点的红色父节点, 且根节点与其左右子节点也不满足, 因此 RB_{k+1} 中也不存在.

综上, 对于任意的 $k \geq 0$, $P(k)$ 成立. 原命题成立.

c)

以任意节点为根的子树中, 所有外部节点的黑色深度相等.

用数学归纳法, 对黑色高度进行归纳, 记 $P(k)$: RB_k, ARB_{k+1} 中所有外部节点的黑色深度相等, 为 k .

Basis. $P(0)$: 由 RBT 的归纳奠基, 显然 RB_0 满足. 对于 ARB_1 , 由递归定义, 其左右子树为 RB_0 , 在各自的子树中, 其外部节点的深度为 0, 又因为当前的根节点不是黑色, 因此 ARB_1 的外部节点的黑色深度也为 0.

I.H. 假设 $P(k)$ 成立.

I.S. 对于 $h = k+1$, 由 RBT 的递归定义, 其根节点为黑色, 且左右子树为 RB_k 或 $ARB_k + 1$, 由归纳假设, 两个子树中的外部节点的黑色深度均为 k , 因此由黑色深度的定义, RB_{k+1} 中的外部节点的黑色深度为其子树中的黑色深度 +1, 因此所有外部节点的黑色深度均为 $k+1$.

综上, 对于任意的 $k \geq 0$, $P(k)$ 成立. 原命题成立.

P 6.4 证明 RBT 的平衡性

1) 性质 1: 内部黑色节点

证明. 用数学归纳法, 对 RB 和 ARB 的黑色高度进行归纳: 记 $P(k)$: ARB_k 有不少于 $2^k - 2$ 个内部黑色节点, RB_{k-1} 有不少于 $2^{k-1} - 1$ 个内部黑色节点.

Basis. $P(1)$: 对于 RB_0 , 有 0 个内部黑色节点, 满足命题. 对于 ARB_1 , 根据 ARB 的递归定义, ARB_k 的内部黑色节点数是 RB_{k-1} 的内部黑色节点数的两倍, 因此 ARB_1 的内部黑色节点数为 0, 满足命题要求.

I.H. 假设 $P(k)$ 成立, $k \geq 1$. 即 ARB_k 有不少于 $2^k - 2$ 个内部黑色节点, RB_{k-1} 有不少于 $2^{k-1} - 1$ 个内部黑色节点.

I.S. 对于 $n = k+1$, 首先考虑 RB_k , 有递归定义, 其内部黑色节点数为左右子树的内部黑色节点数之和 +1(根节点), 而其左右子树可能为 RB_{k-1}, ARB_k , 根据归纳假设, 其内部黑色节点数 $\geq 2 \times (2^{k-1} - 1) + 1 = 2^k - 1$, 满足命题要求. 对于 ARB_{k+1} , 其内部黑色节点数 $\geq 2 \times (2^k - 1) = 2^{k+1} - 2$, 满足命题要求.

综上所述, 对于任意 $k \geq 1$, $P(k)$ 成立, 即原命题成立. \square

2) 性质 2: 内部节点

证明. 用数学归纳法, 对 RB 和 ARB 的黑色高度进行归纳: 记 $P(k)$: ARB_k 有不超过 $\frac{1}{2}4^k - 1$ 个内部节点, RB_{k-1} 有不少于 $4^{k-1} - 1$ 个内部节点.

Basis. $P(1)$: 对于 RB_0 , 由定义, 有 0 个内部节点, 满足 $\leq 4^0 - 1 = 0$. 对于 ARB_1 , 由定义, 其内部节点数为 RB_0 的两倍 +1, 因此由 1 个内部节点, 满足 $\leq \frac{1}{2}4^1 - 1 = 1$.

L.H. 假设 $P(k)$ 成立, $k \geq 1$. 即 ARB_k 有不超过 $\frac{1}{2}4^k - 1$ 个内部节点, RB_{k-1} 有不少于 $4^{k-1} - 1$ 个内部节点.

I.S. 对于 $n = k+1$, 首先考虑 RB_k , 根据定义, 其内部节点数 $\leq 2 \times (\frac{1}{2}4^k - 1) + 1 = 4^k - 1$, 满足. 对于 ARB_{k+1} , 其内部节点 $\leq 2 \times (4^k - 1) + 1 = \frac{1}{2}4^{k+1} - 1$.

综上所述, 对于任意 $k \geq 1$, $P(k)$ 成立, 即原命题成立.

\square

3) 性质 3: 高度

证明. 一个黑色节点的普通高度等于以该点为根节点的子树的所有叶子节点的深度的最大值, 从数值上等于从该节点到某个叶子节点的路径上经过的节点数-1(不包括叶子节点本身)的最大值. 一个黑色节点的黑色高度为该节点到某个叶子节点的路径上经过的黑色节点数-1(不包括叶子节点本身), 由于 RBT 和 ARBT 中, 所有叶子节点的黑色深度相等, 因此黑色高度可以是任意一条这样的从该点到叶子节点的路径.

下面考察这条从该黑色节点 v 到叶子节点 l 且经过的节点数最多的路径 (即决定该黑色节点高度的路径) $P: v \rightarrow v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k \rightarrow l$, 根据上述, 该黑色节点的普通高度 = 路径上的节点数-1, 该黑色节点的黑色高度 = 路径上的黑色节点数-1.

利用 RBT 和 ARBT 的定义, 红色节点不能连续出现, 于是有, 对于 P 上任意红色节点, 其前面一定是一个黑色节点, 而 P 的起点和终点是黑色, 因此, P 中红色节点的数量严格小于黑色节点的数量. 记 v 的黑色高度为 h_b , 则 P 中的黑色节点数为 $h_b + 1$, 因此 P 上总的节点数 $n < 2(h_b + 1) \Rightarrow n \leq 2h_b + 1$, 因此 v 的普通高度 $h \leq 2h_b$.

\square

P 6.5

基本原理

注意到一个性质：排好序的数组被分为 3 段，左边一段满足 $A[i] < i$ ，中间一段满足 $A[i] = i$ ，右边一段满足 $A[i] > i$ ，因此具有较好的二分性质。

因此通过二分查找是否存在 $A[i] = i$.

Algorithm 14: FIND-PROPER-NUM(A[1..n])

Output: 如果存在 $A[i] = i$ ，则返回 True；否则返回 False

```
1 int l := 1, r := n;
2 while l ≤ r do
3     int m := (l + r) / 2;
4     if A[m] = m then return True;
5     else if A[m] < m then
6         l := m+1;
7     else
8         r := m-1;
9 return False;
```

算法时间复杂度分析

算法通过二分查找，每次迭代使得问题规模缩减一半，因此时间复杂度为 $O(\log n)$.

P 6.8

1)

不失一般性，假设 $x > y$ ，因此 $\max(|x - y|) = \max x - \min y$.

因此分别以 $O(n)$ 的时间复杂度查找数组中的最大和最小元素，两者作差即为答案.

2)

如果数组有序，则 $\min y = A[1], \max x = A[n]$ ，因此直接返回 $A[n] - A[1]$ 即可.

3)

基本原理

对数组进行升序排序，显然，对于任意一个元素 x ，与其绝对差最小的元素一定与其相邻（反证法容易证明），因此再遍历数组，不断计算并尝试更新 $|A[i] - A[i+1]|$ ，最终得到最小的绝对差。

伪代码

Algorithm 15: MIN-ABS-DIFF($A[1..n]$)

```
output:  $\min_{x,y \in A[1..n], x \neq y} |x - y|$ 
1 MERGE-SORT( $A$ );
2 int ret := 0;
3 for  $i := 1$  to  $n-1$  do
4   int del := abs ( $A[i] - A[i+1]$ );
5   if  $del > ret$  then  $ret := del$ ;
6 return  $ret$ ;
```

时间复杂度分析

算法的时间复杂度为 $O(n \log n + n) = O(n \log n)$.

4)

算法与第 (3) 问相同。只是不需要排序了。

Algorithm 16: MIN-ABS-DIFF($A[1..n]$)

```
output:  $\min_{x,y \in A[1..n], x \neq y} |x - y|$ 
1 int ret := 0;
2 for  $i := 1$  to  $n-1$  do
3   int del := abs ( $A[i] - A[i+1]$ );
4   if  $del > ret$  then  $ret := del$ ;
5 return  $ret$ ;
```

算法的时间复杂度为 $O(n)$.

P 6.14

注：相关题目

<https://www.acwing.com/problem/content/115/>

这题就用到了这个性质.

1)

证明. 反证法. 假设不存在局部最小元素, 则要么数组严格单调增, 这与 $A[n-1] \leq A[n]$ 矛盾; 要么严格单调减, 这与 $A[1] \geq A[2]$ 矛盾.

因此, 假设不成立, 至少存在一个局部最小元素. \square

2) $O(\log n)$ 时间找到一个极小值

基本原理

注意到一个性质: 对于一个位置 pos , 如果查询得到 $A[pos] \geq A[pos+1]$, 则在 $A[pos..n]$ 之间一定存在一个极小值; 如果 $A[pos] \leq A[pos + 1]$, 则在 $A[1..pos+1]$ 之间一定存在一个极小值.

这个性质其实已经在第 (1) 问中得到证明 (将 $A[1..pos+1]$ 看做是 $B[1..n']$, 实际上是一样的).

因此, 可以对数组进行二分. 每次判断 $A[m]$ 与 $A[m+1]$ 的大小关系, 如果大于则舍弃左半边, 否则舍弃右半边.

最后当数组规模为 3 时结束, 由于数组中一定存在一个极小值, 因此此时的中间元素即为一个极小值.

伪代码

Algorithm 17: PARTIAL-MIN($A[1..n]$)

output: 一个极小值

```
1 int l := 1, r := n;
2 while r - l > 3 do
3     int m := (l + r) / 2;
4     if A[m] ≥ A[m+1] then l := m;
5     else
6         r := m+1;
7 return A[l+1];
```

时间复杂度分析

二分查找, 每次查找使问题规模缩减一半, 故复杂度 $O(\log n)$.

Chapt. 7 分治算法设计要素

P 7.1

1) BF 算法

复杂度 $O(n^2)$:

Algorithm 18: GENERALIZED-INVERSION-PAIRS($A[1..n]$, C), $C \in \mathbb{Z}^+$

output: number of generalized inversion pairs

```
1 初始化: cnt:= 0; /* 用于记录逆序对数 */  
2 for i ← 1 to n-1 do  
3   for j ← i + 1 to n do  
4     if A[i] > C · A[j] then  
5       cnt:= cnt + 1;  
6 return cnt;
```

注：法二：归并排序同时统计逆序对

由于本题没有限制时间复杂度，因此可以直接 $O(n^2)$ 朴素遍历，也可以利用教材7.2.1和7.2.2中基于归排和堆排的算法实现，时间复杂度分别为 $O(n \log n)$ 和 $O(n \log^2 n)$ ；

以下基于归排，给出一种实现的伪码：

```
Algorithm: INVERSION_MERGE(A,l,mid,r)
// init
n1 := mid-l+1; n2 := r-mid;
for i:= 1 to n1 do
    L[i] := A[l+i-1];
for j:= 1 to n2 do
    R[j] := A[mid+j];
L[n1+1] = +∞; R[n2+1] = +∞;
i:=1; j:= 1; inv := 0; // inv for the number of inversions
// merge
for k:= l to r do
    if L[i] > R[j] then // for merge in normal order
        A[k] := R[j];
        if L[i] > C*R[j] and L[i] != +∞ then // C >= 1
            inv := inv + (mid - l - i + 1); // all the L[i+p] > L[i] > C*R[j]
            j := j + 1;
    else
        A[k] := L[i];
        i := i + 1;
return inv;
```

理解该算法为什么可以被拓展到广义逆序对的关键在于理解归并排序是如何“顺带”完成对逆序对的统计的。

相关题目：

- <https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>
- <https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/>

2) 法三: (离散化) 动态树状数组统计逆序对

典型题目:

- [计算右侧小于当前元素的个数 - 计算右侧小于当前元素的个数 - 力扣 \(LeetCode\)](#) ↗ ✓
- [剑指 Offer 51. 数组中的逆序对 - 力扣 \(LeetCode\)](#) ↗ ✓

这两题可以相互规约, 思路相同:

0. 逆序对就是出现在该元素右边的比该元素小的元素. 如果从右往左遍历并统计每个元素的出现次数, 之后只要区间求和即可.
1. 首先离散化处理原数组 --> 否则空间不够
2. 然后类似使用计数排序的原理, 利用树状数组维护每个元素的出现次数, 可以做到 $O(\log n)$ 的时间求前缀和. --> 否则时间不够
3. 从右向左遍历数组, 每次query比当前元素小的前缀和, 时间复杂度为 $O(\log n)$, 并增加该元素的计数(add操作, 复杂度 $O(\log n)$).
4. 因此总的时间复杂度就是 $O(n \log n)$.

个人感觉 **统计逆序对数** 问题, 使用该解法相比归并排序会更容易理解, 也更容易想到.

P 7.2 芯片检测算法的细节

1) 当芯片的数目为奇数时

任意取出一片芯片, 将其余芯片两两配对, 在不考虑这片芯片的前提下先做处理, 剩下来的芯片对都是好好/坏坏. 此时将所有芯片对中取一片出来检测未配对的芯片, 如果这片芯片是好的, 则至少有一半的芯片检测结果是好的; 如果是坏的, 则有少于一半的芯片检测结果是好的. 因此可以在 $O(n)$ 时间内区分出这片芯片的好坏. 如果是好的, 则算法立即终止; 否则舍弃这个芯片.

2) 最后的情况

最后可能剩下 3 片芯片或者 1 片芯片.

如果剩下 3 片芯片, 且好好配对, 坏落单时, 如果按照算法执行, 会导致下一步变成 1 好 1 坏. 因此在常数时间内将 3 个芯片两两测试一次. 如果是 3 好, 则应该全部指示好; 如果 2 好 1 坏, 则存在一个芯片被其余两个芯片均检测为坏; 因此可以特判后者的情况并剔除坏的芯片.

P 7.4

1)

对于将长度为 mk ($1 \leq m \leq k - 1$) 与长度为 k 的数组合并的过程, 其时间复杂度是 $T_m(n, k) = c \cdot (m + 1)n$, 故总的时间复杂度为 $T(n, k) = \sum_{m=1}^{k-1} T_m(n, k) = cn \sum_{i=2}^k i = c \cdot \frac{(k+2)(k-1)}{2}n = \Theta(k^2n)$.

2)

分治算法, 时间复杂度 $O(nk \log k)$.

Algorithm 19: K-MERGE($A[1..nk]$, k , n)

Input : k pairs of sorted arrays with length of n

Output: sorted array

```
1 MERGE-SORT(A, 0, k-1, kn, n);
2 return;
3 MERGE-SORT(arr[], l, r, n, len) begin
4     // l, r 分别表示合并的开头和结尾是在哪一段, n 表示这段数组的总
5     // 长度, len 表示每段初始排数组的长度
6     if l = r then
7         /* 长度为 1, 直接返回 */
8         return ;
9     int m = (l + r) / 2;
10    MERGE-SORT(arr, l, m, n/2, len);
11    MERGE-SORT(arr, m+1, r, n/2, len);
12    // 将排好序的两个数组合并
13    MERGE(arr, l, m+1, n/2, len);
14
15 MERGE(arr[], l1, l2, n, len) begin
16     int temp[]; /* 预留用于排序的空数组 */
17     int pos1 = l1 * len + 1;
18     int pos2 = l2 * len + 1;
19     for i← 1to 2 * len do
20         if pos1 ≤ (l1 + 1) * len AND pos2 ≤ (l2 + 1) * len then
21             if arr[pos1] < arr[pos2] then
22                 temp[i] = arr[pos1];
23                 pos1← pos1 + 1;
24                 i← i+1;
25             else
26                 temp[i] = arr[pos2];
27                 pos2← pos2 + 1;
28                 i← i+1;
29
30         else if pos1 = (l1 + 1) * len then
31             copy the rest of A[pos2..(l2 + 1)*len -1] to temp[];
32         else
33             copy the rest of A[pos1..(l1 + 1)*len -1] to temp[];
34
35     copy temp[1..2n] to arr[l1 * len + 1..(l2 + 1)* len -1];
```

3) 算法时间复杂度分析

令 $m = kn$, 假设 k 是 2 的幂, 则有

$$T(m) = 2T(n/2) + \Theta(m)$$

递归树的高度 $h = \log k$

由主定理, case2, $T(m) = \Theta(m \times h) = \Theta(nk \log k)$.

P 7.5

1)

$O(n)$ 算法计算树的高度:

Algorithm 20: TREE-HEIGHT(Treenode *cur)

output: the height of a tree

```
1 int height := 0;  
2 if cur->left ≠ null then  
3   height := max(height, TREE-HEIGHT(cur->left))  
4 if cur->right ≠ null then  
5   height := max(height, TREE-HEIGHT(cur->right))  
6 return height;
```

2)

$O(n)$ 算法计算树的直径:

Algorithm 21: TREE-DIAMETER(Treenode *cur)

output: the diameter of a tree

```
1 初始化: int ret := -1; /* 整棵树的最长路径 */  
2 HEIGHT (cur);  
3 return ret;
```

Algorithm 22: HEIGHT(Treenode *cur)

output: the height + 1 of cur subtree

```
1 int l := 0, r:= 0; /* 左子树的高度 +1, 右子树的高度 +1 */  
2 if cur->left ≠ null then  
3     l:= HEIGHT (cur->left);  
4 if cur->right ≠ null then  
5     r:= HEIGHT (cur->right);  
6 int len := l + r; /* 当前节点为中继的最长路径 */  
7 if len > ret then ret := len;  
8 return max (l, r) + 1;
```

P 7.7 常见元素

1)

反证法, 假设常见元素 e 不是这两个子数组中某一个子数组的常见元素.
于是有 $f(e) = f_1(e) + f_2(e) < \lfloor \frac{n}{13} \rfloor + 1$, 因此 e 不是常见元素, 矛盾.

2)

反证法, 假设常见元素的个数大于 13, 则所有常见元素的个数大于所有元素的个数,
部分大于整体, 矛盾.

3)

伪代码

Algorithm 23: FREQUENT-NUMS($A[1..n]$), k

output: 数组 $A[1..n]$ 中出现次数大于 $\lfloor \frac{n}{k} \rfloor$ 的元素

```
1 return find-frequent-nums( $A$ , 1,  $n$ ,  $k$ );
2 find-frequent-nums( $A[]$ ,  $l$ ,  $r$ ,  $k$ ) begin
3     set:= null; /* 用于存储候选的常见元素 */ 
4     int  $n := r - l + 1$ ;
5     if  $n \leq 1$  then
6         if  $l = r$  then set.insert( $A[l]$ );
7         return ;
8     int  $m := (l + r) / 2$ ;
9     sl:= find-frequent-nums( $A$ ,  $l$ ,  $m$ ,  $k$ );
10    sr:= find-frequent-nums( $A$ ,  $m+1$ ,  $r$ ,  $k$ );
11    // check the candidates
12    foreach  $e$  in  $sl$  do
13        cnt:= 0; /* 统计次数 */
14        for  $i := 1$  to  $n$  do
15            if  $A[i] = e$  then  $cnt := cnt + 1$ ;
16            if  $cnt > n/k$  then
17                set.insert( $e$ );
18    foreach  $e$  in  $sr$  do
19        cnt:= 0; /* 统计次数 */
20        for  $i := 1$  to  $n$  do
21            if  $A[i] = e$  then  $cnt := cnt + 1$ ;
22            if  $cnt > n/k$  then
23                set.insert( $e$ );
24
25 return set;
```

时间复杂度分析

由前 2 问证明得到, 每轮 candidates 的个数不超过 13, 因此 $f(n) = O(n)$, 因此写出递归式 $T(n) = 2T(n/2) + O(n)$, 由主定理.case2, $T(n) = \Theta(n \log n)$.

4)

当常数 $k = 2$ 时, 可以使用 $O(n)$ 的摩尔投票算法. 大致的思路是每次挑选当前数组中两个不相等的元素删除, 直到无法进行. 若数组非空, 则剩余的元素就是常见元素, 否则不存在常见元素.

5) 不会写

P 7.8 maxima

1)

法一

对所有点按照横坐标升序排序, 从右往左扫描每个点, 自然满足该点不被所有其左边的顶点支配 (x 更大), 因此只需要检验该点的 y 值是否能够大于右边所有的点, 即 $y_i > \max\{y_j | j > i\}$ 即可. 因此可以动态维护一个最大的 y 值, 如果该点的 y 小于 y_{max} 则不是 maxima, 否则是, 且更新 y_{max} . 这个算法的时间复杂度是 $O(n \log n) + O(n) = O(n \log n)$.

法二

$T(n) = 2T(n/2) + O(n)$. 对所有的顶点按照 x 是否大于中位数进行划分 (注意: 这里用到了 $O(n)$ 寻找中位数的选择算法), 得到两个部分后, 右边的 maxima 自然满足是全局的 maxima, 对于左边的, 需要逐个检验是否 $y > y_{max}$ (这个过程只需要维护一个当前区域的 y_{max} 即可), 因此分的代价是 $O(n)$, 合的代价是 $O(n)$, 因此总的时间复杂度由主定理.case2, 为 $O(n \log n)$

2)

问题主要出在: 无法保证划分总是均匀的. 因此递归式实际上是错的.

注

虽然看起来区分的划分时均匀的, 但是我们的研究对象是区域里面的点, 因此实际上不能保证这些点在几个区域中是均匀分布的.

P 7.9 多数政党

1)

基本原理

当剩余的 candidate 数量 > 2 时, 每次选择两个 candidate A, B, 如果 A, B 同政党, 则任意去除一个. 否则两个都去除, 直到最后只剩下一个人, 这个人的政党就是主要政党, 接下来将所有人与这个人进行 check, 就可以判断每个人是否属于主要政党.

时间复杂度分析

由于第一个步骤, 每次比较至少删掉一个 candidate, 因此是 $O(n)$ 的. 第二个步骤也是 $O(n)$, 因此总的时间复杂度为 $O(n)$.

2)

该问题可以规约为 k-majority 问题.

$O(kn)$ 的做法

0. 初始化: ‘flag[1..n]’数组用于表示每个元素属于的政党编号, 初始时全部为 0. ‘cnt[k]’ 用于统计每个政党的人数. 1. 遍历 k 遍数组, 第 i 遍时, 从左到右找到第一个 flag 为 0 的元素, 将其下标改为 i , 继续遍历, 将所有与其同政党的人标记为 i . 同时统计所有下标为 i 的元素数量, 置于 ‘cnt[i-1]’ 中. 这一步的时间复杂度 $O(kn)$. 2. 最后找到 cnt 数组的最大值, 然后在数组中找到一个代表元素返回即可. 这一步的时间复杂度为 $O(k + n)$.

因此, 总的时间复杂度为 $O(kn)$.

P 7.12 寻找缺失的比特串

1)

基本思路

分别遍历每一个 bit 的 n 个元素, 统计 0 和 1 的个数, 最后较少的那个就是缺失的比特串这一位的值.

2)

基本思路

每一次遍历 1 个 bit 位, 统计 1 和 0 个数的同时将元素划分为两部分, 显然这个操作可以在 $O(n)$ 的时间内完成, 统计出较少的是 1 还是 0 后, 抛弃较多的那一半比特串, 递归做下一次.

时间复杂度分析

$$T(n) = T\left(\frac{n}{2}\right) + O(n)$$

由主定理, $T(n) = O(n)$.

Chapt. 8 图的深度优先遍历

P 8.1

1)

当 uv 为:

- TE: 说明此时的 v 是白色顶点;
- BE: 说明 v 是 u 的祖先, 但是此时的 u 的 DFS 尚未结束, 因此 v 的 DFS 同样尚未结束, 因此为灰色;
- DE: 说明 v 是 u 的子孙, 显然 v 不可能是白色, 否则是 TE, 此外也不可能为灰色, 因为 v 是 u 的子孙, 则 v 的 DFS 节点活动区间是 u 的子集, 因此当回到 u 的访问时, v 的 DFS 必然已经结束. 因此 v 是黑色;
- CE: 显然 v 不可能是白色, 否则是 TE, 如果是灰色, 说明 u 和 v 的节点活动区间有重叠, 这与 CE 的性质矛盾. 因此必然是黑色.

2)

当 v 为:

- 白色: 则边 uv 必然为 TE;
- 灰色: 则 uv 必然是 BE;
- 黑色: 则 uv 可能是 CE, DE;

3)

存在, 当遍历边 uv 并刚刚访问到 v 时,

(1) v 为白色 $\iff uv$ 是 TE;

(2) v 为灰色 $\iff uv$ 是 BE;

P 8.2 DFS 算法基于栈的非递归实现

伪代码

Algorithm 24: DFS-NON-RECURSIVE(v)

```
1 初始化: 一个用于维护调用过程的栈  $stk$ , 初始为空;
2  $stk.push(v, 0, \text{size of neighbors of } v, \text{Null})$ ;
3 while  $stk$  非空 do
4    $cur := stk.top()$ ;
5    $v := cur[0]$ ,  $idx := cur[1]$ ,  $n := cur[2]$ ,  $parent := cur[3]$ ;
6   if  $v.color = WHITE$  then
7     <Exploratory processing of edge  $v-parent$ >;
8      $v.color := GRAY$ ;
9     <Preorder processing of node  $v$ >
10  if  $idx < n$  then
11     $idx := idx + 1$ ;
12     $u := v$  的第  $idx$  个邻居;
13    if  $u.color = WHITE$  then
14       $stk.push(u, 0, \text{size of neighbors of } u, v)$ ;
15  else
16    /* 说明已经遍历结束所有子节点 */  
     $v.color := BLACK$ ;
17    <Postorder processing of node  $v$ >;
18    <Backtrack processing of edge  $v-parent$ >;
19     $stk.pop()$ ;
```

P 8.3 定理 8.1 证明

1)

w 是 v 在 DFS 树中的后继节点, iff. $\text{active}(w) \subset \text{active}(v)$.

证明. 1. 必要性 (\Rightarrow). 由于 v 是 u 的祖先, 因此在 DFS 调用过程中, 必然是 v 直接或递归地访问其后继并访问到 w , 因此 $\text{discoverTime}(v) < \text{discoverTime}(w)$; 另外, DFS 保证必须在 v 所有后继节点访问结束后才能结束 v 的 DFS, 因此 $\text{finishTime}(w) < \text{finishTime}(v)$.

2. 充分性 (\Leftarrow).

于是, 反设 w 不是 v 的后继节点. 显然, w 不可能是 v 的祖先节点, 否则, $\text{discoverTime}(w) < \text{discoverTime}(v)$. 于是, w, v 不存在祖先后继关系, 则因为在 $\text{discoverTime}(w) < t < \text{finishTime}(w)$ 的 t 时刻, 两个顶点均为灰色, 因为 v, w 处在两棵不同的 DFS 子树上, 因此不论先开始谁的 DFS 访问, 在结束其变为黑色之前, 不可能开始另一个顶点的 DFS 访问, 即不可能同时满足 u, w 均为灰色, 矛盾. 故假设不成立, w 是 v 的后继节点.

□

2)

首先证明引理: 对于任意两个顶点 u, v , 不可能满足 $\text{active}(u), \text{active}(v)$ 部分重叠, 即 $\text{discoverTime}(u) < \text{discoverTime}(v) < \text{finishTime}(u) < \text{finishTime}(v)$. 该引理的等价表述为: 对于任意两个顶点的活动区间, 要么完全包含, 要么完全不重叠.

证明. 反证法, 假设存在这样的 u, v . 则在 $\text{discoverTime}(v) < t < \text{finishTime}(u)$ 的 t 时刻, u, v 两个顶点均为灰色 (开始 DFS 遍历且未结束), 如果 v, u 为祖先后继关系, 则根据第 (1) 问必要性的证明, 两者的 active 区间为包含关系, 矛盾; 否则, v, u 在两个不同的 DFS 子树上, 则根据 DFS 的性质, 若先开始 u 的 DFS 访问, 则在结束 u 的 DFS 访问之前, 不可能开始 v 的 DFS 访问, 反之同理. 即, 此时不可能满足 u, v 均为灰色, 矛盾. 因此不存在这样的 u, v . □

因此, 由第 (1) 问证明, 若 u, w 没有祖先后继关系, 则两者的活动区间不是完全包含关系, 因此只能是完全不重叠的关系; 如果两个活动区间没有重叠, 则不满足活动区间的包含关系, 因此 u, w 也不存在祖先关系.

3)

(1) 因为 vw 是 CE, 则两个顶点不存在祖先后继关系, 因此活动区间不是包含关系, 由引理, 活动区间只能是完全不重叠的关系, 由于 vw 是 CE, 则 w 是黑色顶点而 v 是灰色, 即 $\text{finishTime}(w) < \text{finishTime}(v)$, 因此 $\text{active}(w)$ 在 $\text{active}(v)$ 前面; 因为 $\text{active}(w)$ 在 $\text{active}(v)$ 前面, 则两个顶点不存在祖先后继关系, 因此必然为 CE.

(2) (先证明 (3).) 由 (3), 本问题立即得证.

(3) 因为 vw 是 TE, 则由第 (1) 问证明, $\text{active}(w) \subset \text{active}(v)$, 反设 $\text{active}(v)$ 不覆盖 $\text{active}(w)$, 假设存在 x , $\text{active}(w) \subset \text{active}(x) \subset \text{active}(v)$. 则由第 (1) 问, x 是 v 的子孙, x 是 w 的祖先, 则 x 必然是 DFS 树中 vw 路径上的点, 因此 w 不是 v 的儿子, 矛盾. 故 $\text{active}(v)$ 覆盖 $\text{active}(w)$; 因为 $\text{active}(v)$ 覆盖 $\text{active}(w)$, 则 w 是 v 的子孙, 如果是 DE, 则在 DFS 树上的 vw 路径上取一个中介点 x , 容易得到 $\text{active}(w) \subset \text{active}(x) \subset \text{active}(v)$, 这与覆盖关系矛盾. 故为 TE.

(4) 因为 vw 是 BE, 则 $\text{active}(v) \subset \text{active}(w)$; 因为 $\text{active}(v) \subset \text{active}(w)$, 则 w 是 v 的祖先, 即 vw 是 BE.

P 8.4

CE.

P 8.5 割点的等价表述

证明. (\Rightarrow)

因为 v 为割点, 在图 $G' = G \setminus v$ 中存在至少两个非空的连通分支 C_1, C_2 , 分别取 $w \in C_1, x \in C_2$, 显然, 这两个顶点在 G 中存在通路, 而删除 v 后不连通, 则说明 v 出现在 w 到 x 的所有路径上.(反证法, 假设存在一条 w 到 x 的路径 P , 且 x 不在 P 上, 则删除 v 后两个顶点仍然连通.)

(\Leftarrow)

假设存在 w, x, v , 满足 v 出现在所有 w 到 v 的路径上, 则显然删除 v 后, 不存在 w 到 x 的通路, 即 w 与 x 不连通, 于是 v 为割点.

□

P 8.6

由题设立即得到两个顶点的活动区间没有重叠, 由定理 8.1 立即得到边 uv 是 CE.

P 8.7 有向图的收缩图无环

证明. 反证法. 容易证明, 对于有向图, 如果存在环, 则该环上的任意两个顶点存在双向通路.

因此, 对于有向图的收缩图 G' , 如果存在环, 考虑换上的任意两个顶点 c_1, c_2, c_1, c_2 间存在双向通路.

考虑收缩图的原图 G . c_1, c_2 在 G 中分别代表一个连通分量 C_1, C_2 . 由于存在 $c_1 \rightarrow c_2$ 的通路, 则说明 $\exists u \in C_1, w \in C_2$, 存在一条 $u \rightarrow w$ 的单向通路.(这里严格要使用数学归纳法, 对两个收缩顶点的距离进行归纳). 现在考虑 $\forall v_i \in C_1, v_j \in C_2$, 因为 C_1, C_2 为强联通分量, 因此存在 $v_i \rightarrow u, w \rightarrow v_j$ 的通路, 于是将三段通路拼接, 就得到了 $v_i \rightarrow u \rightarrow w \rightarrow v_j$ 的通路, 即存在 $v_i \rightarrow v_j$ 的单向通路.

同理, 存在 $v_j \rightarrow v_i$ 的单向通路. 于是存在 v_i, v_j 的双向通路, 这说明 C_1, C_2 中任意两个顶点均有双向通路, 即 $C_1 \cup C_2$ 为更大的强联通分量, 这与 C_1, C_2 为强联通分量的极大性矛盾.

因此假设不成立, 原命题成立. \square

P 8.8

第一次 DFS 不能用 BFS 替代.

因为第一次遍历时需要保证每个人入队的元素都满足自己是 (逻辑) 尽头, 从而保证遍历不会错误地包含从当前节点可达的其他强联通分量中的点. 对于 BFS 算法, 很容易发现, 对于任意顶点的遍历, 会依次将所有相邻的顶点入栈, 这使得入栈的先后顺序不能区分不同的强连通分量的顶点.

第二次 DFS 能用 BFS 替代. 因为第二次的 BFS 实际上是对可达性的判断, 因此使用任意能够判断连通性的遍历算法都能够实现目的.

P 8.9

根节点 v 为割点的充要条件为在 DFS 生成树中, v 的出度大于 1.

证明. 必要性 考虑逆否命题, 如果根节点的出度小于等于 1, 则显然删除根节点后, 不会产生更多的连通分支, 即根节点不是割点.

充分性 如果 v 的出度大于 1, 则说明 v 在对其第一个子节点 u 结束其 DFS 调用时, 存在以 v 的第二个子节点 w 为根的子树尚未访问, 这说明 u 到 w 的所有路径都必须经过 v , 因为此时只有 v 是灰色, 如果存在 u 到 w 的路径, 一定是白色路径, 由白色路径定理, u 在结束 DFS 前一定会遍历到 w , 这与 w 是 v 在生成树中的子节点矛盾. 于是, 删除顶点 v , 至少能使得 u 与 w 不连通, 于是 v 是割点.

\square

P 8.10

在求割点算法中, back 属性的含义是该顶点及其子树中顶点能够到达的最早访问的顶点的 discoverTime 的最小值, 当某个顶点 v 存在一个子树, 该子树中所有顶点都不存在 BE 连接 v 的祖先时, 就认为 v 是割点. 因此, 将所有的顶点的 back 值初始化得更

大 (大于 $discoverTime$) 对结果并没有影响, 因为如果该点存在 BE 连接其祖先时, 就一定能够满足上述判断条件 (即更新后的 back 小于其父节点的 $discoverTime$), 因此仍然能够充分判断所有割点, 对算法没有影响.

P 8.11 Tarjan 找桥算法的正确性

1)

求证: 给定遍历树中的 TE 边 uv , uv 是桥当且仅当以 v 为根的所有遍历子树中没有 BE 指向 v 的祖先 (包括 u).

证明. **充分性.** 由于无向图 BFS 中仅存在 TE 和 BE, 如果不存在这样的 BE, 则说明以 u 为根的子树对应的顶点集关于图 G 的导出子图是一棵树, 因此删除边 uv 后, 至少 u 和 u 的子孙不连通. 于是图 G 不连通, 即 uv 为桥.

必要性. 考虑逆否命题, 假设存在一个以 v 为根的遍历子树, 其中存在一条 BE 连接 v 的祖先, 则删除边 uv 后, 对于以 v 为根的子树中的任意顶点, 可以先到达 v , 在从 v 到达存在 BE 的这个子树中存在 BE 的顶点, 经过 BE 到达 v 的祖先 (即 u 和 u 的祖先), 因此删除 uv 后, 图 G 仍然连通, 因此 uv 不是桥. \square

2)

由引理 8.9, 边 uv 为桥当且仅当以 v 为根的 DFS 子树中没有边关联 u 或 u 的祖先. 因此只要证明后者与算法判断割边条件的等价性. 如果 $v.back > u.discoverTime$, 等价于以 v 为根的子树中没有边关联 u 或 u 的祖先.

P 8.12 无向图和有向图的连通性

1) 无向连通图中至少有两个 non-cut vertex

参考资料: https://www.youtube.com/watch?v=9DYMIX_wGsU

证明. 假设无向连通图 G 至少有 4 个顶点, 否则容易证明结论. 对于图 G , 显然存在两个顶点 u,v , s.t. $dist(u,v) = diam(G)$, 即这两个顶点的最短路是图 G 中任意两个顶点的最短路径长度中最大的. 下面证明, 这两个顶点都不是割点.

不妨取 u , 反设图 $G' = G-u$ 不连通, 则至少存在两个顶点 x,y , s.t. x,y 不连通.

如果 v 不是 x,y 中的顶点, 则考虑连通图 G , 必然存在 $x-v$ 最短路, 同理必然存在 $v-y$ 最短路. 且 u 不在任意一条最短路上, 否则与 u,v 的距离是直径矛盾, 因此删除 u 后, 这两条最短路仍然存在, 因此图 G' 中存在 $x-v-y$ 路径, 即 x,y 连通, 矛盾.

如果 v 是其中的一个顶点, 不失一般性, 假设 $x = v$, 则在图 G 中必然存在 $v-y$ 的最短路, 且 u 不在最短路上, 因此这条最短路在图 G' 中仍然存在, 即 x,y 连通, 矛盾.

因此, 图 $G' = G-u$ 是连通图. 同理图 $G'' = G-v$ 也是连通图. \square

2)

考虑一个单向的轮图即可.

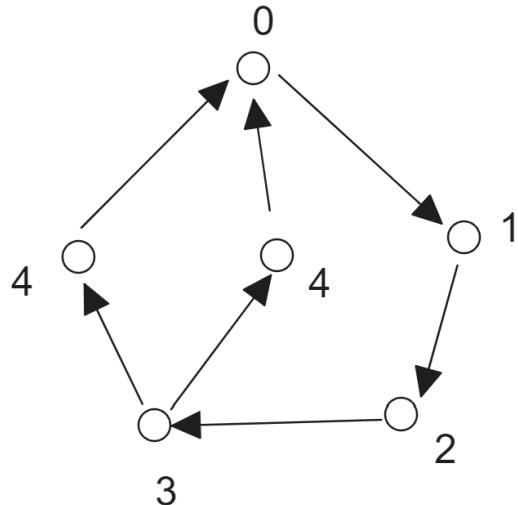
3)

注：这题很容易错

可能会认为, 任意一个包含两个 SCC 的有向图, 添加任何一条边都不能得到强连通图, 但是注意与无向图不同, 非强联通图的底图课可能是连通的.

考虑两个孤立点构成的图.

P 8.13



如图所示, 该算法最终输出最小环的长度为 4, 但是正确的答案是 3. 因此算法错误.

问题出在有时会出现“环套环”的情况, 即一条边属于多个环, 而 DFS 算法保证每条边只会被遍历一次, 因此一个 BE 只能找到一个环, 上图的例子中, 只有两条 BE, 但是存在 3 个环, 因此可能导致真正的最小环没有被发现.

P 8.14

基本原理

首先运行一遍 BFS 算法, 对于每个连通分量, 通过是否存在 CE 来找到一个环. 如果找不到, 则说明图 G 是无环图, 其每个连通分量都是一棵树, 对于树满足边数等于顶点数-1. 于是假设存在每个顶点的入度都至少为 1 的分配, 则边的数量大于等于顶点数, 矛盾, 因此此时不存在任何合法的添加方向的方案.

否则, 则找到了一个环. 对这个环中任意一个顶点 v 设为起点, 删除一条与 v 关联的边 e 得到图 G' , 显然, 删除后图 G' 仍然连通. 然后对 G' 进行以 v 为起点的深度优先遍历 (或者广度优先遍历), 最后得到的遍历调用关系的生成树的方向就是树上每条边分配的方向. 然后将最初删除的那条边的方向指向根顶点. 对于剩余未分配方向的边, 随意分配方向即可.

显然, 算法是线性时间复杂度的.

伪代码

Algorithm 25: ASSIGN-DIRECTION($G = (V, E)$)

Output: 可行的分配方向的方案, 或者返回不存在

```
1 运行 DFS 找环算法;
2 if 不存在环 then
3   return 不存在;
4 else
5   v := 环上任意一个顶点;
6   e := v 关联的一条环上的边;
7   G' := G - e;
8   对 G' 进行 DFS, 按照 DFS 树的方向为树上的每条边分配方向;
9   令 e 指向 v;
10  遍历剩余未分配方向的边, 任意分配方向;
11  return 边的方向数组;
```

正确性证明

注意到一个事实: DFS 树中除根节点外, 每个节点的入度均为 1, 因此只要找到一个度大于 1 且存在一条边满足删除后图仍然连通的节点作为根节点, 进行 DFS 后, 再将删除的边指向根节点, 就能保证每个顶点的入度为 1, 剩余的边随意分配即可.

P 8.15

1)

考虑删除所有桥后的图 G' , 假设图 G' 中原先每个连通片都得到了一个 SCC 定向 (否则显然加上删除的桥也不能得到图 G 的 SCC 定向)

考虑任意两个原先通过桥 e 连接的 SCC 定向子图 U, V , 现在由于给 e 定义一个方向, 则必然存在 $U \rightarrow V, V \rightarrow U$ 之一不能满足, 于是 U, V 不能合成一个更大的 SCC. (注意到如果 $U \not\rightarrow V$ 且 U 能够通过其余删除的桥定向后到达 V , 则与 e 为桥矛盾, 于是 U 一定无法到达 V)

于是 G 不存在 SCC 定向.

2)

因为 G 中不存在桥, 于是图 G 为 2-边连通图, 任意选取图 G 中一个顶点 r 为根运行 DFS 算法, 并为 TE 组成的生成树上的边按照 TE 的方向定向. 这样保证了 DFS 生成树上 r 能够到达任何其余顶点.

接下来要求在 DFS 过程中将所有遇到的 (真)BE(即不包括 TE) 的边按照 BE 的方向进行定向. 由于图 G 不存在割边, 则意味着任意顶点 v , 其子节点 w 的子树中一定存在一条 BE 连接 v 或者 v 的祖先, 由于 v 的祖先也能够到达 v , 于是对于任意顶点 w , 在 w 为根顶点的子树中一定存在一条 BE 连接 w 的父节点 v , 于是 w 一定能够到达其父节点 v , 不断重复这个过程, 即对于任意非根节点, 都能够到达根节点 r .

现在考虑图 G 中任意两个不相等的顶点 u, v , 如果其中有一个是根节点, 则根据上述可以互相到达. 否则存在 $u \rightarrow \dots \rightarrow r \rightarrow \dots \rightarrow v \rightarrow \dots \rightarrow r \dots \rightarrow u$ 的回路, 即 u, v 能够互相到达. 于是找到了一个图 G 的 SCC 定向.

P 8.16

基本原理

如果图 G 不连通, 则显然不存在, 因为问题转化为线性时间判断一棵树中是否存在哈密顿通路.

显然, 树中任何一个顶点的度数必须为 1 或 2, 否则若大于等于 3, 则必然无法只经过该点一次, 访问所有其邻居. 若满足上述要求, 则图 G 中 1 度顶点恰有 2 个, 其余都是 2 度顶点. 通过结构归纳法容易得出这样的图必然是一条链, 因此显然存在哈密顿通路.

因此只需要统计每个顶点的度, 如果恰符合上述要求, 输出 True, 否则输出 False.

P 8.17

反设存在公共边, 则一定存在距离 c 最近的公共边 x-y, 此时两条路径可以表示为:
 $c-\dots-x-y-\dots-u$, $c-\dots-x-y-\dots-v$, 其中, $c-\dots-x$ 路径上没有公共边.

则出现两个矛盾: 1. y 是离 u,v 更近的公共祖先, 因此 c 一定不是 u,v 的最近公共祖先. 2. c 到 x 点存在两条边不交的路径, 这与树的特性: 从根节点出发, 有且只有唯一一条路径到达树中的每个顶点矛盾.

因此不存在公共边.

P 8.18

基本思想

容易证明: 如果边 e 不是割边, 则 e 在一个环中. 因此可以使用 Tarjan 找桥算法在线性时间内判断 e 是否为割边, 也可以构造图 $G' = G - e$, 判断是否为连通图.

P 8.19

实现思路

利用一个线性表 (例如栈) 来记录当前度为 0 的顶点. 初始时扫描并统计每个顶点的入度, 将所有入度为 0 的顶点入栈. 然后不断出栈栈顶元素, 输出该节点, 并删除所有与其关联的边, 删除边的同时将对应的顶点的度减一, 如果为 0 则入栈. 重复直至栈空, 就得到了拓扑排序的结果.

如果存在环

如果存在环, 则会存在顶点没有入栈过. 因为环中任意一个顶点入栈的前提是该顶点的入度为 0, 这意味着其在环中的前驱已经入栈, 不断传递, 得出结论是: 当前顶点入栈的前提是当前节点已经入栈, 显然矛盾, 故当前节点不可能入栈.

P 8.20 顶点间的 "one-to-all" 可达性问题

1)

直接以该顶点为根节点做 BFS/DFS 即可.

2)

基本原理

首先对图 G 运行 SCC 算法, 将每个 SCC 收缩为一个顶点, 得到 G 的收缩图 G' . 对于图 G' , P8.7 已经证明不存在环, 于是一定存在入度为 0 的顶点. 如果入度为 0 的顶点数大于 1, 则不存在能够到达其余所有顶点的顶点 s (这些入度为 0 的顶点间就无法到达). 如果入度为 0 的顶点数为 1, 则该顶点就是 s . 因为其余顶点的入度均大于 0, 于是以 s 为起点的 DFS 一定能够到达图 G' 中所有顶点, 而图 G' 中每个顶点与其对应图 G 中 SCC 中的任意顶点是双向可达的. 于是对于图 G , s 对应的 SCC 中任意一个顶点 v 能够到达其余所有的 SCC 中的某个顶点, 因此能够到达其余所有 SCC 中的任意顶点. 对于 v 所在的 SCC, 显然也能够任意到达.

P 8.21

基本原理

拓扑排序, 按照拓扑排序的逆序进行更新, 即初始时所有顶点的代价均为该点的权重, 从入度为 0 的顶点开始, 删除该顶点, 将该点当前的代价作为该点最终的代价, 并用该点的代价尝试更新其关联的顶点的代价.

分析

算法的时间复杂度为 $O(m + n)$.

P 8.22 影响力值

基本原理

维护一个表, 用于记录每个 SCC 中任意一个顶点的影响力值 (显然都相等)

首先对图 G 运行 SCC 算法, 得到图 G 的收缩图 G' , 并且统计每个强连通分量的顶点个数.

由于图 G' 是 DAG, 对 G' 的每个入度为 0 的顶点运行 DFS 算法, 对每个顶点的 DFS 过程, 在结束对所有子节点的遍历以后, 显然当前节点在图 G' 中能够到达的顶点就是该顶点及其子孙节点. 利用该顶点以及其子节点返回的能够到达的顶点集合做集合运算, 这一步的时间复杂度是 $O(V')$, V' 表示图 G' 中顶点个数. 运算完成后, 用所有该顶点能够到达的 SCC 计算该顶点的影响力值返回该顶点能够到达的顶点集合.

伪代码

Algorithm 26: IMPACT($G = (V, E)$)

```
1 初始化: 维护一个表  $T$  用于记录每个  $SCC$  中任意一个顶点的影响力值, 维
  护一个表  $C$  记录每个  $SCC$  包含的顶点; 初始均为空;
2 运行 Tarjan-SCC 算法, 更新  $C$ , 并构造收缩图  $G'$ ;
3 统计  $G'$  中每个顶点的入度, 加入集合  $S$ ;
4 foreach  $v \in S$  do
5   DFS ( $v$ );
6 DFS ( $v$ ) begin
7    $v.color := GRAY;$ 
8   维护一个记录所有可达的顶点的布尔数组  $R$ , 初始仅自身为 True;
9   foreach  $u \in neighbors of v$  do
10    if  $u.color = WHITE$  then
11       $R := R \cup \text{DFS}(u);$ 
12      /* 两个数组做并集 */
```

```
13   int sum := 0;
14   for  $i := 1$  to  $\text{Sizeof } R$  do
15    if  $R[i] = True$  then
16      sum := sum + C[i];
17   T[i] := sum;
18   return R;
```

时间复杂度

调用 DFS 算法的此时是 $O(m + n)$ 的, 而每个 DFS 函数内部的时间复杂度是 $O(n)$ 的, 于是总的时间复杂度是 $O((m + n) \cdot n)$.

1)

在结束上述过程后, 对维护的表找最小值, 最小值对应的 SCC 中的所有点就是影响力最小的点. 这一步不影响总体时间复杂度.

2)

在结束上述过程后, 对维护的表找最大值, 最大值对应的 SCC 中的所有点就是影响力最大的点. 这一步不影响总体时间复杂度.

P 8.23 线性时间找奇圈

基本原理

对图 G 运行 DFS 算法, 并标记每个顶点的访问时间 (或者用黑白两色进行染色), 如果发现 uv 为 BE 且 uv 同色, 则说明存在奇圈, 否则不存在.

P 8.24

基本原理

等价于求解 AOV 网络上的关键路径长度. 直接调用书本的算法 24:CRITICAL-PATH 即可. 将每门课程视为图 G 中的顶点, 课程先修关系视为有向边, 边的权值为 1. 对图 G 求解关键路径 P , P 的长度即为至少需要的学期数.

P 8.25

1)

将十字路口看做顶点, 所有的街道看做单向边, 因此得到一张有向图. 市长的主张正确当且仅当有向图 G 是强连通图. 因此问题转化为在线性时间内判断一张图是否为强联通图. 可以使用 Tarjan-SCC 算法在线性时间内找出所有的强联通分量, 如果恰有 1 个, 说明市长的主张正确.

2)

问题转化为: 对于有向图 G , 给定一个 G 中顶点 v , 问图 G 是否满足任意再给定一个顶点 u , 如果存在 v 到 u 的有向通路, 则也存在 u 到 v 的有向通路.

首先对图 G 运行一遍以 v 为起点的 DFS, 记录所有能够到达的顶点为集合 S ;

然后将图 G 中所有的边反向得到图 G' , 对图 G' 同样以 v 为起点运行一遍 DFS, 记录所有能够到达的顶点 S' ;

如果 $S = S'$, 则市长的主张可行, 否则不可行.

显然, 该算法运行了两边 DFS, 时间复杂度是线性的.

P 8.26 小孩排队问题

1)

基本原理

将 i 恨 j 视作有向图中的一条有向边 (i, j) , 因为拓扑排序的结果满足所有 j 的入边对应的另一个顶点都出现在 j 的前面, 因此满足题目中这样的 i 需要排在 j 之前的要求. 因此按照上述规则建图后求解拓扑排序即可. 如果不存在拓扑排序, 则说明不存在排队的方案.

2)

基本原理

对于任何一条“憎恨传递链”, 都等价于有向图中的一条单向路径, 显然, 排队的行数不少于任何一条这样的单向路径的长度. 于是排队的行数至少为关键路径的长度 n .

另一方面, 对于关键路径上的顶点, 显然可以分配到对应长度个数的行上去. 对于非关键路径, 如果路径上的所有顶点均与之前已经安排好的不重复, 则显然直接安排即可; 否则存在一些顶点与之前安排好的顶点重复, 将这若干条含有重合顶点的路径合并并找到其中最长的路径, 如果这条路径上的顶点能够分配进 n 行中, 则其余的也一定能够分配. 否则, 则得到了一条长度大于 n 的路径, 这与关键路径的长度为 n 矛盾. 因此一定能够分配成功.

于是问题等价于求解 AOV 网络上的关键路径长度. 直接调用书本的算法 24:CRITICAL-PATH 即可.

注

感觉说的不是很清楚, 后面再完善.

P 8.27 神秘文字的字母排序问题

基本原理

建立一个以所有字母为顶点的有向图 G , 遍历所有相邻的单词, 对于两个相邻的单词 w_i, w_{i+1} , 找到两者第一个分歧的字母 l, l' , 向图 G 中添加一条 $l \rightarrow l'$ 的有向边.

完成建图以后, 运行拓扑排序, 由于存在全序, 因此拓扑排序一定能够得到一个(可能是部分)字母的序.

时间复杂度为 $O(mW + n^2)$, 其中 W 为单词的最大长度.

P 8.28 2-SAT 问题

1)

通过编写程序枚举所有的五元组并验证得到, 有且仅有两组五元组赋值使得上述子句都被满足:

- (True, False, False, True)
- (True, True, False, True)

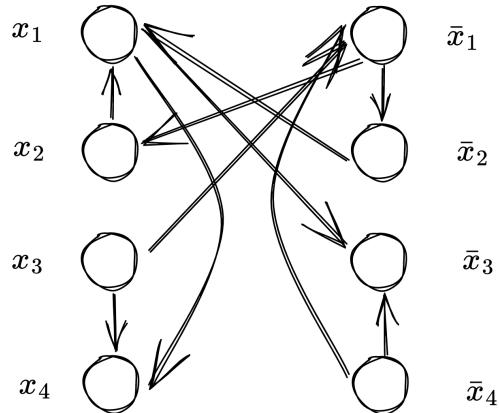
2)

构造的一组不存在满足赋值的实例为 $(\alpha \vee \beta) \wedge (\bar{\alpha} \vee \beta) \wedge (\gamma \vee \bar{\beta}) \wedge (\bar{\gamma} \vee \bar{\beta})$

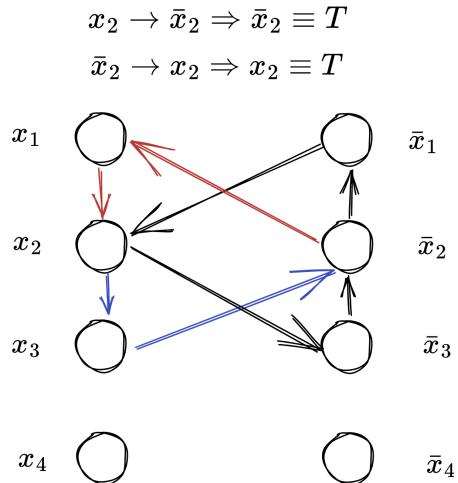
上述实例中, 前两个子句蕴含 $\beta = \text{True}$, 后两个子句蕴含 $\beta = \text{False}$, 故不存在合法的赋值.

3)

1. 题目中的实例对应的有向图如下:



2. 第 (2) 问中的实例对应的有向图如下:



4)

证明. 反证法, 假设存在使得所有子句都满足的赋值. 如果存在一个强联通片中同时包含 x, \bar{x} , 则说明存在 $x \rightarrow \bar{x}$ 与 $\bar{x} \rightarrow x$ 的有向通路, 由于蕴含关系具有传递性, 于是有 $x \rightarrow \bar{x} \equiv T \Rightarrow \bar{x} \equiv T$, 以及 $\bar{x} \rightarrow x \equiv T \Rightarrow x \equiv T$, 而 $x \wedge \bar{x} \equiv F$ 矛盾. 于是不存在使得所有子句都满足的赋值. \square

5)

证明. 构造证明法. 由于对于任意文字 x , G 中不同时存在 $x \rightarrow \bar{x}$ 与 $\bar{x} \rightarrow x$ 的有向通路, 于是可以按照如下规则构造一组赋值:

- 如果图 G 中仅包含 $x \rightarrow \bar{x}$ 的有向路径, 则说明 $x \rightarrow \bar{x} \equiv \bar{x} \equiv T$, 于是令文字 x 赋值为 False.
- 如果图 G 中仅包含 $\bar{x} \rightarrow x$ 的有向路径, 则说明 $\bar{x} \rightarrow x \equiv x \equiv T$, 于是令文字 x 赋值为 True.
- 否则, 说明两个方向的路径均不存在, 此时任意赋值, 不妨令文字 x 赋值为 True.

\square

注 : 待完善

此处证明不充分.

6)

基本原理

首先以 $O(m + n)$ 的时间建图, 然后以 $O(m + n)$ 的时间运行 Kosaraju 算法并判断每个连通分量中是否同时存在某个文字及该文字的取反. 如果存在则直接返回不存在. 否则接下来对图 G 进行 DFS. 当 DFS 遍历到某个顶点时, 判断该顶点对应文字的取反顶点的颜色, 如果是白色则不处理; 如果是灰色则令当前节点对应文字为 True; 如果为黑色则令当前节点对应文字的取反为 True.

最后遍历所有文字, 对于未赋值的文字可以任意赋值, 不妨令所有文字为 True.

伪代码

Algorithm 27: 2-SAT

Input : 子句集合 S

Output: 一种使得所有子句均满足的赋值方案, 或不存在

```
1 初始化: int vis[] 数组用于标记访问状态, 初始时均为白色. res[] 数组用于存
  储赋值结果, 初始均为 Undefined.;

2 根据 S 建图;

3 Kosaraju 算法并判断每个强连通分量中是否同时包含某个文字及其取
  反; /* 可以利用哈希表实现判断 */
```

```
4 if 存在某个连通分量包含某个文字及其取反 then
  5   return 不存在赋值;

6 foreach u ∈ G do
  7   if u 是白色 then
  8     DFS (u);
```

```
9 foreach u ∈ G do
  /* 处理无法到达的顶点 */
```

```
10  if res[u] is Undefined then
  11    res[u] := True;
```

```
12 return res;
```

```
13 DFS(u) begin
  14   u := 灰色;
  15   if u 取反的顶点为灰色 then
    /* 说明  $\bar{u} \rightarrow u$  */
```

```
16     res[u] := True;
```

```
17   else if u 取反的顶点为黑色 then
    /* 说明不存在  $\bar{u} \rightarrow u$  */
```

```
18     res[u] := False;
```

```
19   foreach u 的邻居 v do
  20     if v 是白色 then
  21       DFS (v);
```

```
22   u := 黑色;
```

时间复杂度分析

建图, Kosaraju 求 SCC, 以及 DFS 和最后的遍历都是线性的, 因此算法的总时间复杂度是线性的.

Chapt. 9 图的广度优先遍历

P 9.1

证明无向图的广度优先遍历过程中, 不存在 BE 和 DE.

证明. • BE: 假设存在灰色顶点 u 通过 BE 边 uv 发现了顶点 v , 因为 v 是 BFS 树中 u 的祖先节点, 在 v 出队前将 u 入队, 于是 uv 是 TE, 与 uv 是 BE 矛盾.

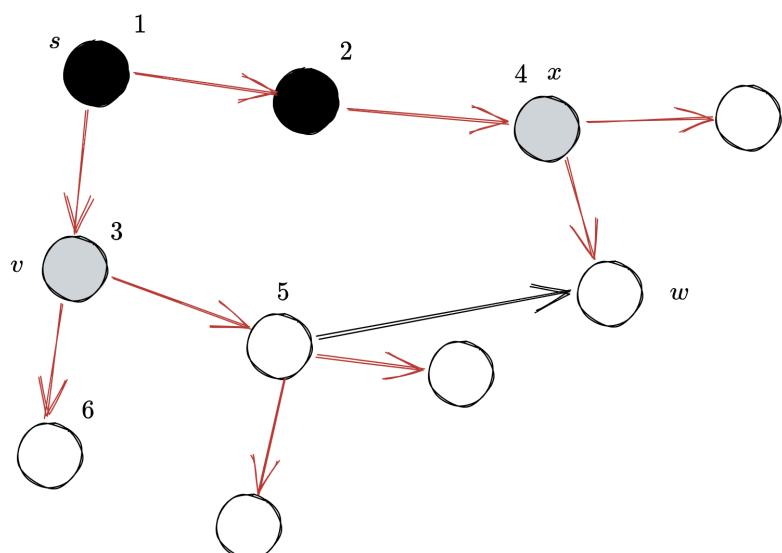
• DE: 假设存在灰色顶点 u 通过 DE 边 uv 发现了顶点 v , 对 v 的颜色进行讨论:

- v 为白色, 则说明 uv 是 TE, 矛盾
- v 为灰色, 则说明 v 已经入队, 这说明要么 uv 是 TE, 要么 v 不是 u 的子孙, 于是 uv 不是 DE.
- v 为黑色, 则说明 v 不是 u 的子孙, 与 uv 是 DE 矛盾.

□

P 9.2

白色路径定理: 在 BFS 树中, 节点 v 是 w 的祖先, 当且仅当在遍历过程中刚刚发现点 v 的时刻, 存在一条从 v 到 w 的全部由白色节点组成的路径.



对 BFS 不成立, 如上图所示, 图中数字标号为入队顺序, 红色边为 BFS 树边, 可见, v 到 w 满足存在一条白色节点组成的路径, 但是 v 并不是 w 的祖先.

之所以白色路径定理对 DFS 成立, 是因为当遍历到 v , 且存在一条到 w 的白色路径时, 在结束 v 的遍历前, 一定会访问 w , 这就说明 v 是 w 的祖先. 而对于 BFS, 虽然存在 v 到 w 的白色路径, 但是 v 并不会立即访问 w , 这可能导致 w 被其余顶点先遍历到, 导致 w 不是 v 的子孙.

P 9.3 DFS 能否判断二分图

无向图 G 为二分图当且仅当图 G 存在一个 2-染色, 即, 存在一种用黑白两色对所有顶点染色的方案, 使得所有相邻顶点的颜色不同.

不论是 BFS 还是 DFS, 都能确保遍历所有的边, 同时在第一次遍历时对未访问过的顶点染确定的颜色 (只要起点颜色确定), 因此都能确保检查了所有可能矛盾的情况, 如果遍历结束均无矛盾, 则说明存在正确的染色方案.

因此 DFS 能够判断无向图 G 是否为二分图.

从对手论证的角度, 总是能够构造出对 DFS 有利或者对 BFS 有利的输入, 因此对于具体的输入, 两者的性能不好比较, 但是总体来说, 由于 BFS 能够确保立即利用所有获得的信息, 且 (从感觉上说)DFS 容易“一条道走到黑”, 在不是二分图的情况下, 可能需要花费更多的时间. 对于是二分图的情况, 则两者的效率一致的.

P 9.4

1) DFS

有向图

有向图中存在环当且仅当存在 BE. 因此只要遍历到某个点 u 时, 发现邻居颜色为灰色, 则说明边 uv 为 TE, 存在环.

无向图

由于无向图的 DFS 中不存在 DE, CE, 因此当遍历到一个顶点已经访问过时, 就说明存在环.

2) BFS

有向图

如果利用 BFS 框架的话, 需要在 u 遇到黑色节点 v 时判断 $LCA(u, v)$ 是否等于 v . 因为此时 uv 可能是 BE, 也可能是 CE. 但是这样的实现的复杂度不是线性的.

可以考虑首先统计每个顶点的入度, 如果不存在入度为 0 的顶点, 则说明一定存在环. 否则, 将所有入度为 0 的顶点入队. 然后依次从队列中取出元素, 将该点的出边关联的顶点的入度减一, 如果变为 0 则入队. 最后将队首元素弹出. 重复上述过程并统计出队元素个数. 如果直到队空, 出队的元素个数少于 n , 则说明存在环.

无向图

无向图 BFS 中, 一旦遍历到某个顶点的邻居为非白色顶点, 则说明存在环. 否则不存在.

P 9.5

如果图 G 中存在环, 则显然删除环上任意一条边都能保证删除后的图 G' 仍然连通; 如果图 G 中不存在环, 则图 G 为树, 树是极小的连通图, 即删除任何一条边都会使得图不连通. 因此无向连通图 G 存在一条删除后任然连通的边当且仅当图 G 中存在环.

因此用 P9.4 的无向图 DFS 找环算法, 如果找到了一条 BE, 则说明存在环. 删除这条 BE 就能保证图 G' 连通.

P 9.6 离线处理祖先后继关系

基本原理

对二叉树进行一次 DFS, 并记录每个节点的活动区间. 对于任意的查询, 检查区间是否是包含关系即可.

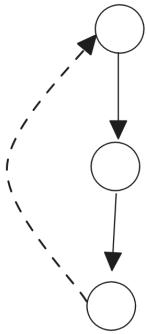
P 9.7

1) 无向图的情况

证明. 如果 $G \neq T$, 则对于 DFS 生成树, 存在边 uv 是 BE, 但是, 对于 BFS 生成树来说, 不存在 BE, 因为如果存在的话, 将会导致 v 直接将 u 作为 v 的儿子, 但是在 DFS 树中, u 是 v 的非儿子的后继节点, 矛盾. \square

2) 有向图的情况

可能存在, 如图所示:



P 9.8

1) 边权为 1 的 MST

MST 是包含了所有顶点且边权和最小的树。注意到，对于任意一棵图 G 的生成树，其边的数目都为 $n - 1$ ，因此边权和均为 $n - 1$ 。因此任意一棵图 G 的生成树都是图 G 的 MST。

因此对图 G 运行 BFS/DFS 算法，将得到的生成树记录下来，即为结果。

2) $m = n + 10$

注意到此时的边数仅比生成树的边数多 11 条，因此可以先运行一遍 DFS/BFS，先将得到的生成树记为 M_0 。

接下来迭代 11 条非树边，对于第 i 条边 $e_i = (u, v)$ ，使用 Tarjan 离线 LCA 算法计算 u, v 关于 M_{i-1} 的 LCA 为顶点 r （这一步的时间复杂度为 $O(n+1) = O(n)$ ），则 $u - r - v - u$ 构成一个环，遍历环上的所有边，找到边权最小的树边 e' ，如果 $\text{val}(e_i) < \text{val}(e')$ ，则将 e_i 替换 e' 得到 M_i 。

最后得到的 M_{11} 就是 MST。

3) 边权为 1 或 2

利用 01-BFS 算法的思想。首先将所有边的边权-1。这样所有边的边权为 0 或 1。且这个变换不影响 MST 边的选择。准备一个双端队列 deque，在进行 BFS 时，如果在遍历到 u 时访问到了一个白色顶点 v 且 uv 的边权为 0，则将 v 从队列的队首入队，否则正常从队尾入队。这样仍然保证了队列 dis 的单调性以及首尾元素 dis 的差距不超过 1。最后完成 01-BFS 得到的生成树就是图 G' 的生成树，然后将所有的边的边权加 1 即得到图 G 的 MST。

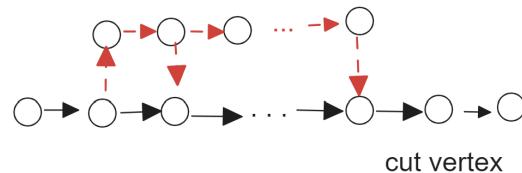
P 9.9

1)

证明. 即证明存在节点 t , 使得所有 $v-w$ 路径都经过 t .

如果 n 为偶数, 则 $v-w$ 最短路 P 上的顶点一共有 $\frac{n}{2} + 2$ 个 (包含端点), 因此还剩余 $\frac{n}{2} - 2$ 个顶点. 最坏情况下, 这些顶点都与 P 上的某些顶点关联. 如下图所示. 不妨对 P 上的顶点做标定 $P: v_0 - v_1 - \dots - v_{n-1} - v_n$, 考虑剩余的某个顶点 u 与 v_i 关联, 则要么 u 继续关联一个不在 P 中的剩余顶点, 要么与 v_{i+1} 或 v_{i-1} 关联, 若与 P 中其余顶点关联, 则与最短路径为 P 矛盾. 据此, 利用数学归纳法, 对这条添加的链上的顶点数进行归纳, 容易得到, 若用不在 P 中的 k 个顶点至多关联 k 个 P 上的顶点, 因此剩余的顶点至多关联 $\frac{n}{2} - 2$ 个顶点, 而 P 上有 $\frac{n}{2} - 1$ 个中间节点, 因此至少存在 1 个顶点没有被其余顶点关联, 即所有 $v-w$ 路径都经过该点. 因此删除该点, v,w 不连通.

如果 n 为奇数, 类似的, P 上有 $\frac{n+1}{2} + 1$ 个顶点 (包含端点), 则剩余 $\frac{n-3}{2}$ 个顶点, 而 P 上有 $\frac{n-1}{2}$ 个中间节点, 因此至少存在 1 个顶点没有被任何顶点关联. 同理可得删除该点, v,w 不连通.



□

注

感觉不是很严谨.

2)

首先用 BFS 算法在线性时间内找到 P . 然后遍历 P 上的每个中间顶点, 如果度恰为 2, 则该点就是所求.

P 9.10

(没太理解题目的意思, 就直接给出做法)

基本原理

思路就是为每个顶点维护一个 s 到该点 u 的距离 $dis[u]$, 在进行该点的 BFS 扩散时, 尝试用 $dis[u] + 1$ 来更新 $dis[v]$, 如果

- $dis[v] + 1 = dis[u]$, 则说明经过 u 到达 v 的同样也是最短路, 因此 $c[u] += c[v]$;
- $dis[v] + 1 < dis[u]$, 则说明之前计算的到达 v 的“最短路径”不是最短的, 而经过 u 到达 v 的可能是最短的, 因此更新 $dis[u] := dis[v] + 1$, 同时令 $c[v] := c[u]$;
- otherwise, 说明经过 u 到达 v 的不可能是最短路径, 什么也不做.

运行 BFS 结束后, 即得到所有的最短路径条数. 时间复杂度为线性.

P 9.11 会员制舞会

基本原理

即要求得到一个 m 阶子图, 要求子图中每个顶点的度数 k 满足 $5 \leq k < n - 5$, 因此利用 k 度子图的思想, 首先统计每个顶点的度, 将显然不满足要求的顶点 (即度数 $< 5, \geq n - 5$ 的顶点入队), 然后每次不断取出队首元素, 将该元素删除, 并将其邻居的度-1, 如果其邻居也变得不满足, 则将其邻居入队. 重复过程直到队空. 如果此时所有元素均被删除, 则不存在非平凡的合法子图. 否则未被删除的顶点均满足了要求, 于是这些顶点就是最优的被邀请的客人名单.

时间复杂度分析

该算法与 k 度子图类似, 都是基于 BFS, 于是算法是线性时间复杂度, $O(m + n)$.

Chapt. 10 图优化问题的贪心求解

P 10.3

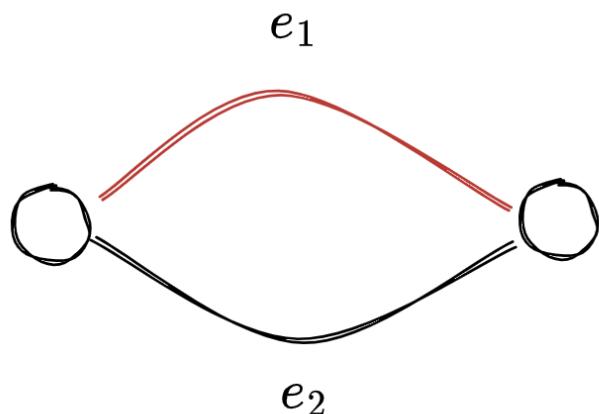
1)

一定, 可以通过反证法证明: 如果权值最小的边 e_1 不在任何一个 MST 中, 则任取一个 MST, 添加该边必然得到一个环, 删除环上任意一条边都会使得剩下的图构成 ST 且边权和更小, 与原图为 MST 矛盾. 因此边 e_1 一定在 G 的某一个最小生成树中.

2)

如果不允许出现重边 (即两个顶点间连接多条边) 则一定, 可以通过反证法证明: 如果权值第 2 小的边 e_2 不在任何一个 MST 中, 则任取一个 MST, 添加该边必然得到一个环, 注意到该环上必然存在边权大于 e_2 的边, 否则该环仅由 e_1, e_2 组成, 而根据假设, 任意两个顶点间不存在重边, 因此可以找到一条边权大于 e_2 的边, 删掉该边得到的图构成 ST 且边权和更小, 与原图为 MST 矛盾. 因此 e_2 一定在 G 的某个最小生成树中.

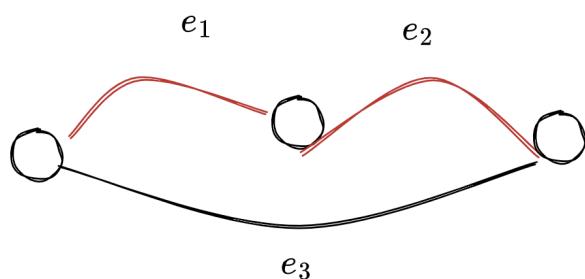
但是如果允许重边, 则不一定, 反例如下图:



图中, e_2 不属于任何一个 MST 中.

3)

不一定存在, 反例如下:



图中, e_3 不属于任何一个 MST 中.

P 10.6 比较 Prim 和 Kruskal 算法的好坏

1) Prim 算法

$$T(n, m) = O(n \cdot C_{\text{extract-min}} + n \cdot C_{\text{insert}} + m \cdot C_{\text{decrease-key}})$$

基于数组实现优先队列

- Extract-min 通过遍历数组实现, 因此单次操作是 $O(n)$ 的.
- Insert 和 Decrease-key 可以直接访问并修改数组对应元素的值, 因此单次操作是 $O(1)$ 的.

因此 $T(m, n) = O(n^2 + n + m) = O(n^2 + m)$.

基于二叉堆实现优先队列

- Extract-min 直接获取二叉堆堆顶元素, 单次操作为 $O(1)$.
- Decrease-Key 可以 $O(1)$ 时间内做到 (还不需要 Fix-heap)
- Insert 的插入操作代价为 $O(1)$, 但是在插入元素后, 下一次 Extract-Min 之前, 需要进行一次 Fix-heap 操作, 不妨将这个操作记在 Insert 操作上, 因此单次代价为 $(\log n)$.

因此 $T(m, n) = O(n \log n + m)$.

2) Kruskal 算法

基于数组或矩阵实现并查集

排序的代价为 $O(m \log m) = O(m \log n)$. 并查集的代价为 $O(mn)$, 因此 $T(m, n) = O(m \log n + mn) = O(mn)$.

基于根树的并查集

排序的代价为 $O(m \log m) = O(m \log n)$.

接下来需要对大小为 n 的并查集进行 $O(m)$ 次的并查操作.

- 如果使用普通并 + 普通查来实现, 可能因为树高度不平衡导致这部分的代价恶化到 $O(mn)$.

- 如果使用启发式合并 (按数目/秩合并)+ 普通查来实现, 代价为 $O(m \log n + n) = O(m \log n)$.
- 如果使用启发式合并 + 路径压缩查来实现, 则代价为 $O(m\alpha(m, n)) \approx O(m)$.

因此, 只要使用启发式合并 (也可以使用路径压缩) 来实现并查集的并查操作, 就能够保证总的时间复杂度为 $O(m \log n)$.

3) 比较

仅考虑 Prim 算法和 Kruskal 算法的最佳实现, 时间复杂度分别为 $O(n \log n + m)$, $O(m \log n)$.

- 当图为稀疏图, 即 $m = \Theta(n)$ 时, 两者的时间复杂度均为 $O(n \log n)$, 两者从渐进时间复杂度层面上说是相同的.
- 当图为稠密图, 即 $m = \Theta(n^2)$ 时, Prim 算法的时间复杂度为 $O(n^2)$, Kruskal 算法的时间复杂度为 $O(n^2 \log n)$, 此时 Prim 算法优于 Kruskal 算法.

P 10.10

$$1) e \notin E' \wedge \hat{w}(e) > w(e)$$

如果将不在 T 中的一条边的边权增加, 不会影响 T 的最小性, 因此直接返回 T 即可.

$$2) e \notin E' \wedge \hat{w}(e) < w(e)$$

如果将不在 T 中的一条边的边权降低, 则将 e 加入 E' , 此时 T' 有且仅有一个环, 使用 DFS 找环算法可以在线性时间内找到环并标记环上的边, 遍历所有换上的边, 如果边权最大的边 e' 不是 e , 则删除 e' , 得到 T' 为新的 MST.

$$3) e \in E' \wedge \hat{w}(e) < w(e)$$

如果将在 T 中的一条边的边权降低, 不会影响 T 的最小性, 因此直接返回 T 即可.

$$4) e \in E' \wedge \hat{w}(e) > w(e)$$

如果将在 T 中的一条边的边权增加, 则在图 T 中删除边 e , 这样会得到两个连通分量, 用并查集 (启发式合并 + 路径压缩)/DFS 标记两个连通分量中的顶点. 然后遍历每条边, 判断该边关联的两个顶点是否属于不同的连通分量 (这样就说明这条边可能构成新的 MST 中的边, 否则成环), 维护这样的边的最小值. 最后选择边权最小的边 e' (可能等于 e) 加入 T 得到 T' 即为新的 MST.

P 10.13

1) 法一: Kruskal 算法

基本原理

首先将所有 S 中的边加入 T , 用并查集维护每个顶点属于的连通分量.

然后对 $E(G) - S$ 中的边按照边权进行升序排序, 依次选择每条边, 用并查集判断添加该边是否成环, 若不成环则选择, 否则舍弃. 重复直到一共选择了 $n-1$ 条边.

时间复杂度分析

与 Kruskal 算法的时间复杂度相同, 为 $O(m \log n)$.

P 10.14

记 $e = (u, v)$, 遍历所有边, 删掉边权大于等于 $l(e)$ 的边. 然后对图 G' 以 u 为起点进行 DFS,

- 如果 u 能够访问到 v , 则说明已经存在一条 u 到 v 的路径 P , P 上边权最大值 $< l(e)$, 若 e 在 MST 中, 考虑这条路径上边, 必然有一条边 e' 不属于 MST, 于是 $T' = T - e + e'$ 同样构成树, 且具有更小的边权和, 与 T 为 MST 矛盾. 因此 e 不在 MST 中.
- 否则, e 在 MST 中. 反证法, 假设 e 不在任何 MST 中, 则任何 MST 中加入 e 都会得到一个环, 这意味着删除 e , uv 仍然连通, 与 u 无法到达 v 矛盾.

注

上述证明对 case1 的证明不够严谨.

P 10.15

1)

错, 若该边为割边, 则必须选择.

2)

正确. 反证法, 假设 e 属于某个最小生成树 T , 则这个环上的边必然存在一条边 e' 不属于 T , 因为这条边的边权一定小于 e , 因此构造 $T' = T - e + e'$, T' 为生成树且边权更小, 与 T 为 MST 矛盾.

3)

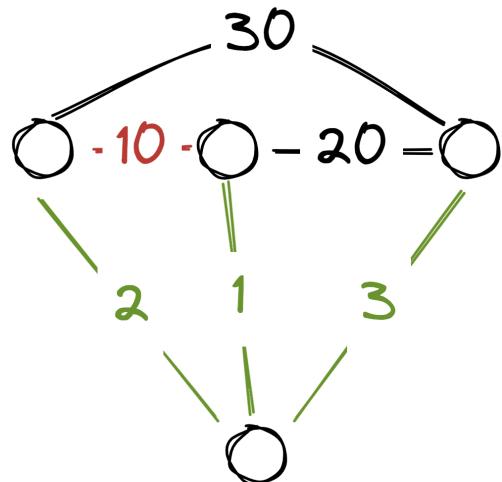
正确. 反证法, 如果图不连通或者无环则显然, 否则, 如果 e 不属于任何 MST, 则任取一个最小生成树 T , 将 e 加入 T , 必然产生一个环 (长度大于等于 2), 环上必然存在边权大于等于 e 的边 e' , 令 $T' = T - e' + e$, T' 为生成树且边权和更小, 与 T 为 MST 矛盾.

4)

正确. 反证法, 如果图不连通或者无环则显然, 否则, 假设存在一个最小生成树 T 不包含 e , 则添加 e 必然产生一个环, 且环上其余的边的边权均大于 e , 因此删除任意一条边会得到更小的生成树, 与 T 为 MST 矛盾.

5)

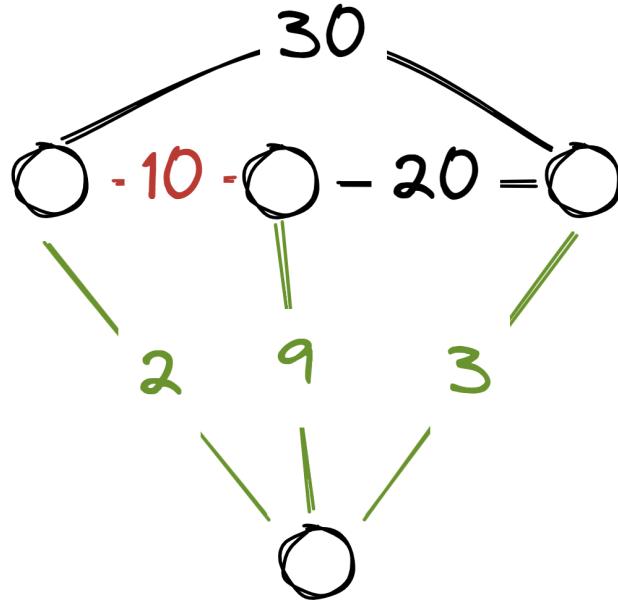
错误. 反例如下图:



图中边权为 10 的边 e 处在一个环中且是最轻边, 但显然 e 不属于任何 MST 中.

6)

错误, 反例如下:



如图所示, 边权为 10 的边 $e = (u, v)$ 是 u, v 顶点间的最短路径, 但是 e 不属于任何 MST 中.

7)

正确. Prim 算法正确性的关键在于保证局部生成树是局部最小生成树, 在 $T^{k-1} \rightarrow T^k$ 的过程中, 如果存在负权边, 不影响第 k 个顶点的选择 (因为总是选择边权最小的), 而后续的证明只关心边的大小关系, 而不关心边的具体权重, 因此不影响 Prim 算法的正确性.

注 : 这题非常易错

考虑到 Prim 算法与 Dijkstra 算法“师出同门”, 容易想当然的认为 Prim 算法同样无法处理带负权边的图.

P 10.16

命题正确, 证明如下:

证明. 因为每条边的权重均不相同, 因此对边集 E 进行排序后可以得到一个严格单调增加的序列 $S : e_{i_1}, e_{i_2}, \dots, e_{i_m}$, 容易证明一个性质: $\forall x, y \in \mathbb{N}^+, x < y \iff x^2 < y^2$, 因此对平方后的边集 E' 重复上述操作后得到的序列 $S' : e_{j_1}, e_{j_2}, \dots, e_{j_m}$, 有 $\forall 1 \leq k \leq m, e_{i_k} = e_{j_k}$, 即两次排序的结果相同.

考虑 Kruskal 算法执行过程, 算法总是按照序列中的边的顺序进行选择, 容易通过数学归纳法证明, 对 G 和 G' 运行 Kruskal 算法时, 每一轮选择的边都是相同的. 因此两次算法最终得到的 MST 包含相同的边. 即, T 是 G 的 MST 当且仅当 T' 是 G' 的 MST.

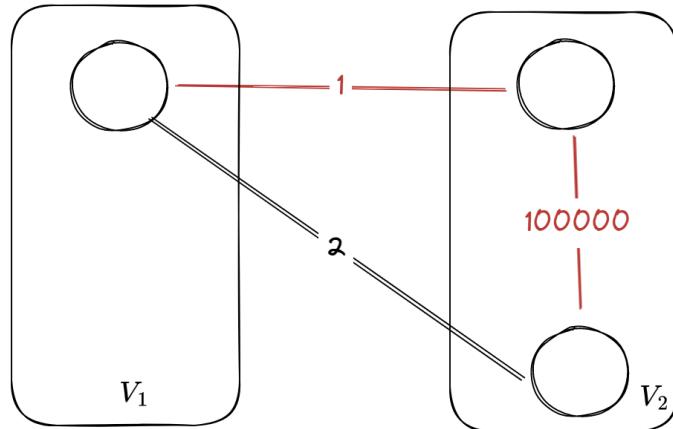
□

P 10.17

证明. 反证法. 首先观察图 $T \cap H$, 由于所有的边都属于 T , 因此如果选择的子图 H 中包含的顶点在 T 中不是一个连通分量, 则得到的图 $T \cap H$ 就不是连通图, 考虑图 $T \cap H$ 的每个连通分量, 如果 $T \cap H$ 不属于任何图 H 的 MST 的一部分, 则至少存在一个连通分量 C_i , C_i 不是 C_i 对应的顶点集的 MST, 即, 存在一个更小的 MST, 记为 C'_i . 对于原图 G , 构造 $T' = T - C_i + C'_i$ (这里的-表示删除对应的边). 因为 C_i, C'_i 都是生成树, 因此 T' 同样是生成树, 同时, T' 的边权和小于 T , 与 T 为图 G 的最小生成树矛盾. □

P 10.21

不能, 反例如下图所示:



如图所示, 分治算法不能得出正确的 MST.

P 10.23

1)

基本原理

对于确定的打井的房子, 总体造价 = 该房子打井的代价 + MST 的边权和, 而 MST 的边权和是定值. 调用 Kruskal 算法求解 MST, 则最小造价为 $\min\{W_i\} + w(\text{MST})_i$.

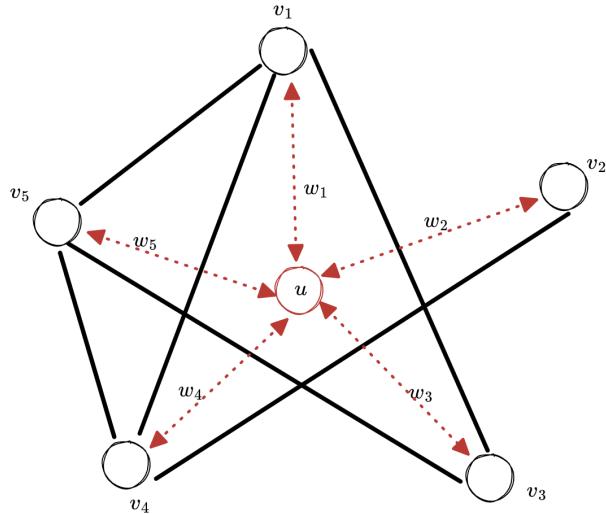
时间复杂度分析

算法的时间复杂度为 $O(m \log n)$.

2)

基本原理

构造图 $G' \leftarrow G$, 然后添加顶点 u , $\forall v_i \in V(G), G' \leftarrow (u, v_i, w_i)$ (注意, 这些边是双向边). 然后对图 G' 运行一次 Kruskal 求解 MST 即可.

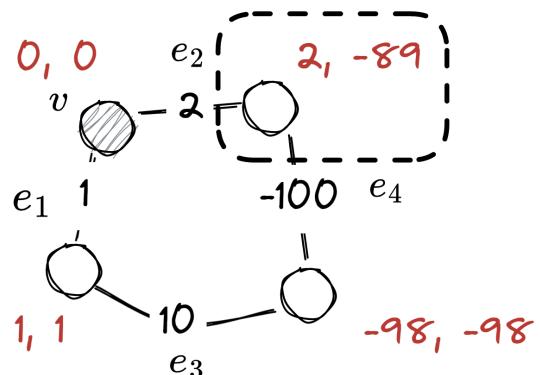


时间复杂度分析

算法的时间复杂度为 $O(m \log n)$.

P 10.25

反例如下图:



其中, 框出的顶点的 dist 为 2, 但是实际的最短距离为-89. Dijkstra 算法得到的结果错误.

P 10.27

基本原理

Dijkstra 算法在处理带负权边的图时会出错的原因在于可能滞后发现了负权边, 导致无法保证当前的最优解 = 全局的最优解.

因此对于只含有 1 条负权边的图, 可以特判这条负权边, 即, 首先遍历找到的这条负权边 $e = (u, v)$, 分别对 u, v 为起点做一次 Dijkstra 算法计算所有顶点分别到 u, v 的最短距离 $dist(u, v_i), dist(v, v_i)$. 然后删除这条边 e 得到图 G' , 对图 G' 以 s 为起点做一次 Dijkstra 算法计算所有顶点到 s 的最短距离 $dist(s, v_i)$. 最后, 遍历每个顶点, 尝试更新 $dist(s, v_i) = \min\{dist(s, v_i), \max\{dist(u, s) + dist(v, v_i), dist(v, s) + dist(u, v_i)\} + w(e)\}$ 即可.

算法正确性证明

分类讨论 s 到 t 的最短路径 P ,

- 若 $e \notin P$, 则删除 e 这条边对 P 上的每个顶点的最短距离都不影响, 因为 s 到 P 上的每个顶点的路径都是最短路径, 可以用数学归纳法证明上述性质的正确性.
- 若 $e \in P$, 则 P 可以表示为 $s..u-v..t$ 或者 $s..v-u..t$ 的形式, 容易证明 $dist(u, v_i), dist(v, v_i)$ 都是正确的, 因此 s 到 u 或 v 的最短距离 + t 到 u 或 v 的最短距离 + $w(e)$ 一定是最短路径的值. 这里需要取 \max 来避免重复计算负权边 e 的情况.

时间复杂度分析

算法调用了三次 Dijkstra, 后续的更新的代价是 $O(n)$ 的, 因此总的时间复杂度仍为 $O(m \log n)$.

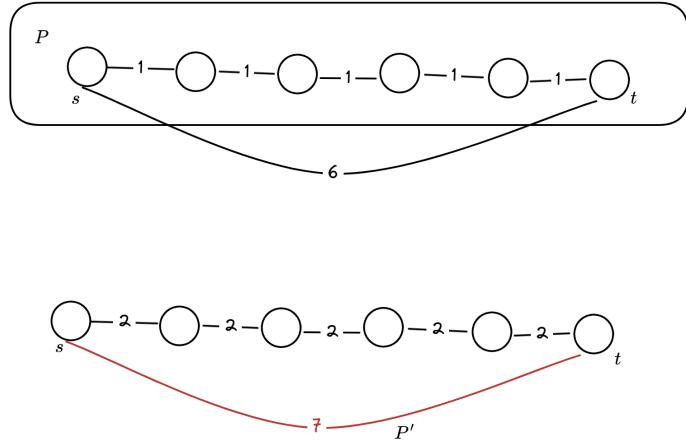
P 10.31

1)

不变. 考虑 Kruskal 算法分别对变化前和变化后的图进行求解. 由于两者遍历边的顺序完全相同, 且图的构造完全相同, 容易用数学归纳法证明得到的 MST 的边集相同. 因此不会变化 (当然权值和会增加 $n-1$).

2)

可能会变, 反例如下:



变化的原因在于最短路径的边数如果较多，则变化导致的增加同样较多，可能导致不是最短.

P 10.33 推广的最短路径问题

基本原理

对于确定的出发点 s , 推广后的最短路径 $dist(P) = c(s) + [l(s, v_1) + c(v_1)] + [l(v_1, v_2) + c(v_2)] + \dots + [l(v_k, t) + c(t)]$. 因此可以对 Dijkstra 算法稍作修改. 具体来说, 在进行 Decrease-key 操作时, 用 $dist[u] + l(u, v) + c(v)$ 来更新 $dist[v]$. 其余的 Extract-Min 和 Insert 操作不变. 之后遍历每个顶点 t , 令 $dist'[t] = dist[t] + c(s)$, 于是 $dist'[t]$ 就是推广后的最短路径长度.

正确性证明

因为 Dijkstra 可以看做带边权的 BFS, 因此遍历时同样具有”定向”属性. 而一旦定向, 则每条边附加的点权就确定了, 因此可以视作是新的边权, 而这个转换不影响 Dijkstra 正常求解最短路.

时间复杂度分析

与 Dijkstra 算法的时间复杂度一致.

P 10.34

能. 证明如下:

证明. 证明 Dijkstra 的关键在于证明 Dijkstra 每一次选择的 Fringe 中的顶点 u , 计算出的局部最短距离 $dist[u]$ 等于全局 s 到 u 的最短距离. 用数学归纳法来证明, 对 dijkstra

的第 i 次选择和处理进行归纳. 记 $P(i)$: Dijkstra 第 i 次选择的顶点 u , 计算的局部最短距离等于全局最短距离.

Basis. $P(1)$: 第一次选择顶点 s , 局部最短距离 = 全局最短距离 = 0.

I.H. 假设对于 $k \geq 1, \forall 1 \leq i \leq k, P(i)$ 均成立.

I.S. 现在考虑第 $k+1$ 次选择, 记 Dijkstra 算法贪心选择了 fringe 中的顶点 u , 于是有 $\forall v \in \text{fringe} \wedge v \neq u, \text{dist}[u] < \text{dist}[v]$. 由于 $k+1 > 1$, s 顶点必然已经是 Finished 的状态, 不在 fringe 中. 于是对于所有处于 fringe 或 fresh 状态的顶点, 都不存在负权出边. 于是对于其余任意到达 u 的路径, 都必然经过 fringe 中的某个顶点 v , 在经过若干 fringe 和 fresh 中顶点最后到达 u , 而这些顶点的出边均非负, 于是 $\text{dist}'[u] > \text{dist}[v] > \text{dist}[u]$, 即不存在更短的到达 u 的路径, 因此 $\text{dist}[u]$ 是全局最短的 s 到 u 的路径长度.

□

P 10.36

基本原理

修改 Dijkstra 算法维护的堆的结构: 每个节点为一个二元组 $(\text{dist}[u], \text{best}[u])$, 按照 dist 属性有小到大, 若相同则按 best 属性由小到大确定优先级.

对于每个节点 u , 维护两个属性 dist , best , 分别为 s 到 u 的局部最短距离和局部最短路径边数. 修改 Dijkstra 算法的 Decrease-key 操作: 遍历 Finish 顶点 u 的每个邻 [v 居 v , 若

- $\text{dist}[v] > \text{dist}[u] + l(u, v)$, 则 $\text{dist}[v] := \text{dist}[u] + l(u, v)$, $\text{best}[v] := \text{best}[u] + 1$;
- $\text{dist}[v] = \text{dist}[u] + l(u, v)$, 则 $\text{best}[v] := \min(\text{best}[v], \text{best}[u] + 1)$;
- $\text{dist}[v] < \text{dist}[u] + l(u, v)$, 则不作任何操作.

算法正确性证明

证明. 首先, 显然该算法能够得出正确的最短距离, 因为上述修改对 Dijkstra 求最短距离没有影响. 同样使用数学归纳法对 best 属性的正确性进行证明, 记 $P(i)$: 该算法第 i 次选择的顶点 u 的 best 属性等于全局从 s 到 u 含边最少的最短路径的边的数目.

Basis. 第一次选择顶点 s , $\text{best}=0$, 显然成立.

I.H. 假设对于 $k \geq 1, \forall 1 \leq i \leq k, P(i)$ 均成立.

I.S. 现在考虑 $P(k+1)$, 记 Dijkstra 贪心策略选择了顶点 u . 于是对于其余的 fringe 中的顶点 v , 若 $\text{dist}[v] > \text{dist}[u]$, 容易证明通过这些顶点到达 u 的总长度均大于 $\text{dist}[u]$, 因此这些顶点出发不能产生更小的 $\text{best}[u]$ 值. 对于 $\text{dist}[v] = \text{dist}[u]$ 的顶点 v , 由于算法没有选择 v , 说明 $\text{best}[v] \geq \text{best}[u]$, 显然, 若想通过 v 到达 u 的路径长度最短, 则 v 必须经过若干边权为 0 的边 (至少 1 条) 到达 u , 则这条路径的长度大于 $\text{best}[u]$, 因此也不

影响 $\text{best}[u]$ 值 (不会变得更小). 同时, 注意到 $\text{best}[u]$ 已经是所有 Finished 顶点的 best 值更新得到的最小值, 由归纳假设, 这些顶点的 best 值已经是全局最小, 因此导出的顶点 u 的 best 值是局部最小. 而上述分析表明其余的非局部最小的情况无法推出更小的 best 值, 因此 $\text{best}[u]$ 是全局最小.

□

时间复杂度分析

时间复杂度与 Dijkstra 算法的时间复杂度一致.

P 10.38

1)

遍历每条边, 将边权 $> L$ 的边删除 (标记即可), 然后运行 DFS/BFS 判断 s, t 是否连通即可. 显然算法的时间复杂度是线性的.

2)

基本原理

判断能否通过 s 到达某个顶点 u , 只需要满足存在一条 s 到 u 的路径 P , 其中 P 上的每条边的边权的最大值不超过 L 即可. 因此可以维护每个顶点的 val 属性, $\text{val}[u] \equiv \min_{P_i(s,u)} \max_{e_i \in P_i} l(e)$. 注意到 val 属性满足 “局部最小” = “全局最小”. 因此只需要修改 Dijkstra 算法的 Decrease-Key 环节: 当前贪心选择 fringe 中 val 最小的顶点 u 访问其邻居 v , 并尝试更新其 $\text{val}[v] := \min(\text{val}[v], \max(\text{val}[u], l(u,v)))$. 这样通过拓展 Dijkstra 算法能够求出 $\text{val}[t]$ 即为满足 s 能够到达 t 的最小油箱容量.

时间复杂度

与 Dijkstra 算法的时间复杂度一致, 使用堆优化可以做到 $O((n+m) \log n)$.

Chapt. 11 贪心算法设计要素

P 11.1

基本原理

将 S 表示为 c 进制: $a_{n-1}a_{n-2}\cdots a_1a_0$, 于是面值为 c^k 的硬币需要 a_k 个.

算法正确性证明

证明. 反证法, 首先证明引理: 对于 S 的最优分配方案 A , 其 c 进制表示的每一个位小于 c . 否则, 可以将该位- c , 该位的高 1 位 +1(可能递归进行), 每做一次这样的操作, 总硬币数会减少 $c-1 \geq 1$ 个, 与当前方案为最优方案矛盾.

假设存在另一种硬币数更少的方案 $B: b_{n-1}b_{n-2}\dots b_1b_0$. 由于总硬币数更少, 必然存在某一位的 $b_i < a_i$. 取满足 $b_i < a_i$ 的最大的 i , 显然, i 的更高位 $b_j = a_j$, 这是因为如果 $b_j > a_j$, 则 B 表示的总金额 $> A$ 表示的总金额 $= S$, 矛盾; 如果 $b_j < a_j$, 与 i 最大矛盾. 因为 i 的更高位上 A 和 B 的硬币数均相等, 于是第 i 位上 B 少于 A 的硬币面值只能通过低于 i 的位上有超过 A 对应位的硬币来补充.

注意到, 即使 $b_{i-1}b_{i-2}\dots b_1b_0$ 的每一位均为 $c-1$ (由引理证明, 不可能大于 $c-1$), 总的面值 $(c-1) \times \frac{c^i-1}{c-1} = c^i - 1 < c^i$, 即, 即使低于 i 的每一位均达到最大值, 也无法补偿 i 位的差值, 矛盾. 因此不存在硬币数少于 A 方案的方案. 即该算法得到的方案最优. \square

Chapt. 12 图优化问题的动态规划求解

P 12.1

1)

Algorithm 28: FLOYD-WARSHALL-GO(W)

```
1  $D^{(0)} := W$ ; /* 初始情况下, 两点间的最短路径长度就是它们之间的权重 */  
2  $GO^{(0)} := 0$ ;  
3 for  $i := 1$  to  $n$  do  
4    $GO_{ij}^{(0)} := j$ ; /* 初始情况下, 两点间最短路径的第一条边的终点就是  $j$  */  
5 for  $k := 1$  to  $n$  do  
6   for  $i := 1$  to  $n$  do  
7     for  $j := 1$  to  $n$  do  
8       if  $D_{ik}^{(k-1)} + D_{kj}^{(k-1)} < D_{ij}^{(k-1)}$  then  
9          $D_{ij}^{(k)} := D_{ik}^{(k-1)} + D_{kj}^{(k-1)}$ ;  
10         $GO_{ij}^{(k)} := GO_{ik}^{(k-1)}$ ;  
11      else  
12         $D_{ij}^{(k)} := D_{ij}^{(k-1)}$ ;  
13         $GO_{ij}^{(k)} := GO_{ij}^{(k-1)}$ ;  
14 return  $D^{(n)}$ ;
```

2)

Algorithm 29: FLOYD-WARSHALL-GO(W)

```
1  $D^{(0)} := W; /*$  初始情况下，两点间的最短路径长度就是它们之间的权重 */  
2  $GO^{(0)} := 0;$   
3 for  $i := 1$  to  $n$  do  
4    $GO_{ij}^{(0)} := i; /*$  初始情况下，两点间最短路径的最后一条边的起点就是 i */  
5 for  $k := 1$  to  $n$  do  
6   for  $i := 1$  to  $n$  do  
7     for  $j := 1$  to  $n$  do  
8       if  $D_{ik}^{(k-1)} + D_{kj}^{(k-1)} < D_{ij}^{(k-1)}$  then  
9          $D_{ij}^{(k)} := D_{ik}^{(k-1)} + D_{kj}^{(k-1)};$   
10         $GO_{ij}^{(k)} := GO_{kj}^{(k-1)};$   
11      else  
12         $D_{ij}^{(k)} := D_{ij}^{(k-1)};$   
13         $GO_{ij}^{(k)} := GO_{ij}^{(k-1)};$   
14 return  $D^{(n)};$ 
```

P 12.2 最大吞吐率

1)

基本原理

该问题与 P10.38 是对偶问题，因此可以相似处理。可以维护每个顶点的 val 属性， $\text{val}[u] \equiv \max_{P_i(s,u)} \min_{e_i \in P_i} l(e)$ 。注意到 val 属性满足“局部最小”=“全局最小”。因此只需要修改 Dijkstra 算法的 Decrease-Key 环节：当前贪心选择 fringe 中 val 最小的顶点 u 访问其邻居 v，并尝试更新其 $\text{val}[v] := \max(\text{val}[v], \min(\text{val}[u], l(u,v)))$ 。这样通过拓展 Dijkstra 算法能够求出 $\text{val}[t]$ 即为 s 到 t 的所有路径的最大吞吐率。

时间复杂度分析

与 Dijkstra 算法的时间复杂度一致，使用堆优化的 Dijkstra 算法的时间复杂度为 $O((m+n) \log n)$ 。

2)

基本原理

伪代码

Algorithm 30: Pipeline(W)

```
1  $D^{(0)} := W; /*$  初始情况下，两点间的 val 就是它们之间的边权 */  
2 for  $k := 1$  to  $n$  do  
3   for  $i := 1$  to  $n$  do  
4     for  $j := 1$  to  $n$  do  
5        $D_{ij}^{(k)} := \max(D_{ij}^{(k-1)}, \min(D_{ik}^{(k-1)}, D_{kj}^{(k-1)}));$   
6 return  $D^{(n)};$ 
```

算法正确性证明

证明. 通过数学归纳法证明 $P(k)$: 第 k 轮迭代后得到的 $D_{ij}^{(k)}$ 是 i 到 j 的所有 (可能) 经过顶点 $\{1, 2, \dots, k\}$ 中顶点的路径的吞吐率的最大值.

Basis. $P(0)$: 第 0 轮迭代后, 即第 1 轮迭代开始前, i 到 j 的路径不能经过任何中继点, 因此唯一的路径就是 $P = i \rightarrow j$, 因此此时的最大吞吐量为 W_{ij} 正确.

I.H. 假设对于 $k \geq 1$, $P(k)$ 成立.

I.S. 第 $k+1$ 轮迭代, 对于任意顶点 i, j , 如果 i 到 j 的最大吞吐率路径不经过 $k+1$, 则显然 $D_{ij}^{(k+1)} = D_{ij}^{(k)}$; 否则, $D_{ij}^{(k+1)} = \min(D_{ik+1}^{(k)}, D_{k+1j}^{(k)})$, 由归纳假设, $D_{ik+1}^{(k)}, D_{k+1j}^{(k)}$ 都已经正确表示, 因此 $D_{ij}^{(k+1)}$ 也是正确的. \square

P 12.4

基本原理

首先遍历所有顶点, 将属于 S 的顶点的 $dist$ 值置为 0 并加入优先队列, 将属于 T 的顶点标记. 然后开始 Dijkstra 算法, 当某次迭代时 Extract-Min 的顶点 u 为被标记过的 T 中顶点时, 循环终止, 此时的 $dist[u]$ 即为 S 到 T 的最短路径.

算法正确性证明

注意到, Dijkstra 找到的最短路径的长度是随着迭代的顺序递增的, 因此当第一次 Extract-Min 发现 T 中顶点的最短路径时, 一定是最短的到达 T 的路径.

P 12.7

基本原理

在图 G 中以顶点 v_0 为起点进行一次 Dijkstra 算法, 将所得的最短路径记为 $\text{dist}[u]$. 在图 G 的转置图 G' 以顶点 v_0 为起点进行一次 Dijkstra 算法, 将所得的最短距离记为 $\text{dist}'[u]$.

对于任意两个顶点的“最短路径”请求 $\text{QUERY}(s,t)$, 返回 $\text{dist}'[s] + \text{dist}[t]$ 即可.

Chapt. 13 动态规划算法设计要素

P 13.1

1) 算法设计

Algorithm 31: COMPATIBILITY

```
1 for i := 1 to n do
2   for j := i+1 to n do
3     for k := i to n-1 do
4       if  $f_i \leq s_k \wedge f_k \leq s_j$  then
5         /* 说明  $a_k$  可选; */  
         c[i][j] := max (c[i][j], c[i][k] + c[k+1][j] + 1);
```

2) 算法分析

根据算法设计, 算法的状态数为 $O(n^2)$, 因此空间复杂度为 $O(n^2)$, 时间复杂度为 $O(n^2) \times O(n) = O(n^3)$.

P 13.2

基本原理

用 $f(i, j)$ 表示利用数组中的 $A[i..n]$ 部分能否表示 j , 若能则为 True, 否则为 False. 于是 $f(1, S)$ 就是整个问题的回答.

容易分析得到, 对于每个状态 $f(i, j)$, 状态转移方程如下:

- 如果 $i = n$, 则当 $A[n]$ 为 j 时为 True, 否则为 False.

- 否则, 如果 $j \geq A[i]$, 则 $f(i, j) = f(i+1, j) \vee f(i+1, j - A[i])$, 否则 $f(i, j) = f(i+1, j)$.

分析

算法的状态数为 $O(nS)$, 因此空间复杂度为 $O(nS)$, 时间复杂度为 $O(nS) \times O(1) = O(nS)$.

P 13.3

基本原理

定义状态 $f[i]$ 为正整数 i 变为 1 至少需要的操作数. 显然, $f[n]$ 就是整个问题的解.
状态转移方程:

- $f[1] = 0$;
- $f[i] = f[i-1] + 1$, for $i > 1$;
- if $i \equiv 0 \pmod{2}$, $f[i] = \min(f[i], f[i/2] + 1)$;
- if $i \equiv 0 \pmod{3}$, $f[i] = \min(f[i], f[i/3] + 1)$;

递推的顺序按照 i 从小到大即可.

P 13.4

基本原理

定义状态 $f[i]$ 为 $A[1..i]$ 部分的最长非递减子序列长度. 于是 $f[n]$ 就是整个问题的解答.

状态转移方程:

- $f[1] = 1$
- $f[i] = \max(1, \max_{1 \leq j < i}(f[j]) + 1)$

分析

状态的个数为 $O(n)$, 因此空间复杂度为 $O(n)$, 时间复杂度为 $O(n) \times O(n) = O(n^2)$.

P 13.5

1) 法一 DP

基本原理

定义状态 $f(i, j)$ 为数组 $A[1..i]$ 进行 j -划分的最小代价。于是整个问题的解就是 $f(n, k)$ 。

状态转移方程：

- if $j = 1$, 说明不能进行切分, $f[i][j] = \text{sum}(A[1..i])$, 这可以通过预处理前缀和后 $O(1)$ 回答。
- else if $i \leq j$, 则说明可以全部切开, 于是 $f[i][j] = \min_{1 \leq k \leq i} A[k]$, 这个操作可以通过预处理计算 $A[1..k]$ 的最小值后 $O(1)$ 回答。
- else, 枚举所有可以切分的点, $f[i][j] = \min_{1 \leq k < i} \max(f[k][j - 1], \text{sum}[k + 1..i])$. 其中, $\text{sum}(A[k+1..i])$ 可以通过预处理前缀和后 $O(1)$ 回答。

分析

状态的个数为 $O(nk)$, 因此空间复杂度为 $O(nk)$, 因为状态转移方程中 $f[j]$ 仅依赖 $f[j-1]$, 因此空间复杂度可以降低到 $O(n)$, 时间复杂度为 $O(nk) \times O(n) = O(n^2k)$.

2) 法二 二分 + 贪心

基本原理

由于答案为 $\min_{1 \leq k \leq n} \max_{1 \leq i \leq n} C(P)$, 因此可以考虑二分答案。对于每次 check 的假设答案 S , 从左到右遍历数组, 用双指针记录 $A[l..r]$ 的区间和, 一旦超过 S , 则说明必须切一刀, 更新 l, r , 并统计“切割”的次数 t , 如果 $t < k$, 则说明 S 是可能的答案。由于 S 具有单调性, 因此答案就是最小的合法答案。

分析

假设二分的答案区间为 $[0, S]$ (其实可以理论分析并缩小), check 的时间复杂度为 $O(n)$, 因此时间复杂度为 $O(n \log S)$, 空间复杂度为 $O(1)$.

P 13.6

1) 均为正数的情况

枚举滑动窗口的长度, 枚举滑动窗口左端点, 时间复杂度 $O(n^2)$.

2) 有正有负的情况

枚举滑动窗口的长度, 枚举滑动窗口左端点, 时间复杂度 $O(n^2)$.

P 13.7 打家劫舍

基本原理

定义状态 $f[i]$ 表示子问题: 当正整数序列 $X' = X[i..n]$ 时的权重最大的合法下标集合. 显然, $f[1]$ 就是整个问题的解.

状态转移方程:

- $f[n] = X[n], f[n+1] = 0;$
- $f[i] = \max(f[i+1], f[i+2] + X[i]).$

由于输出要求给出具体的选择方案, 因此:

- 方法 1: 还需要开一个数组 $I[1..n]$ 用于记录 $f[i]$ 在状态转移时是否选择了 $X[i]$;
- 方法 2: 利用 $f[1]$ 是否等于 $f[2]$, 是否等于 $f[3] + X[1]$ 来判断是否选择 $X[1]$, 后面的元素同理可以判断, 时间复杂度还是 $O(n)$.

P 13.8 公共子序列问题

1) LCS

基本原理

定义状态 $f(i, j)$ 表示 $X[1..i]$ 与 $Y[1..j]$ 的 LCS. 于是整个问题的解就是 $f(m, n)$.

状态转移方程:

- $i = 0$ 或者 $j = 0, f[i][j] = 0;$
- $X[i] = Y[j], f[i][j] = f[i-1][j-1] + 1;$
- otherwise, 失配, $f[i][j] = \max(f[i-1][j], f[i][j-1]);$

分析

状态的个数为 $O(mn)$, 于是空间复杂度为 $O(mn)$, 时间复杂度为 $O(mn) \times O(1) = O(mn)$.

2) LCS 变式 1

基本原理

定义状态 $f(i, j)$ 表示 $X[1..i]$ 与 $Y[1..j]$ 的 LCS. 于是整个问题的解就是 $f(m, n)$.

状态转移方程:

- $i = 0$ 或者 $j = 0$, $f[i][j] = 0$;
- $X[i] = Y[j]$, $f[i][j] = \max(f[i-1][j-1], f[i][j-1]) + 1$;
- otherwise, 失配, $f[i][j] = \max(f[i-1][j], f[i][j-1])$;

分析

状态的个数为 $O(mn)$, 于是空间复杂度为 $O(mn)$, 时间复杂度为 $O(mn) \times O(1) = O(mn)$.

3) LCS 变式 2

基本原理

定义状态 $f(i, j, k)$ 表示 $X[1..i]$ 与 $Y[1..j]$ 的 LCS, 其中 $X[i]$ 已经匹配了 k 次 (不超过 K 次). 于是整个问题的解就是 $f(m, n, 0)$.

状态转移方程:

- $i = 0$ 或者 $j = 0$, $f[i][j][k] = 0$;
- $X[i] = Y[j]$,
 - if($k+1 < K$) $f[i][j][k] = \max(f[i-1][j-1][0], f[i][j-1][k+1]) + 1$;
 - else($k+1 = K$), 此时不能继续使用这个 $A[i]$, 因此 $f[i][j][k] = f[i-1][j-1][0]$;
- otherwise, 失配, $f[i][j][k] = \max(f[i-1][j][0], f[i][j-1][k])$;

分析

状态的个数为 $O(mnk)$, 于是空间复杂度为 $O(mnk)$, 时间复杂度为 $O(mnk) \times O(1) = O(mnk)$.

P 13.9 不重叠的最长回文串

基本原理

定义状态变量 $f(i, j)$ 为字符串 $S[i..j]$ 部分的最长回文子串 (不能连续) 的长度. 于是 $f(1, n)$ 就是整个问题的解.

状态转移方程:

- $\text{len} = j - i + 1 \leq 1$, 由于要求子串不能重叠, 因此 $f[i][j] = 0$;
- if $S[i] = S[j]$, $s[i][j] = s[i+1][j-1] + 1$;
- else, $s[i][j] = \max(s[i+1][j], s[i][j-1])$;

分析

状态的个数为 $O(L^2)$, 因此空间复杂度为 $O(L^2)$, 时间复杂度为 $O(L^2) \times O(1) = O(L^2)$, 其中 L 为 S 的长度.

P 13.10 公共超序列

基本原理

定义状态 $f(i, j)$ 表示 $A[1..i]$ 和 $B[1..j]$ 的 SCS(Shortest Common Supersequence), 于是 $f[m][n]$ 就是整个问题的解答.

状态转移方程:

- $f[0][j] = B[1..j]$, $f[i][0] = A[1..i]$;
- if $A[i] = B[j]$, $f[i][j] = f[i-1][j-1] + A[i]$ ("+" 表示字符串拼接操作)
- else
 - if $\text{len}(f[i-1][j]) < \text{len}(f[i][j-1])$ $f[i][j] = f[i-1][j] + A[i]$;
 - else $f[i][j] = f[i][j-1] + B[j]$;

分析

状态的个数为 $O(mn)$, 因此空间复杂度为 $O(mn)$, 时间复杂度为 $O(mn) \times O(1) = O(mn)$.

P 13.11 LCS 变式

1)

不正确. 反例如下:

$$\begin{aligned} X &= \underline{\underline{ABC}} \\ Y &= \underline{CBA} & Y' &= \underline{BCA} \\ Z &= \underline{\underline{AB}} \underline{\underline{CBA}} \underline{\underline{C}} \end{aligned}$$

$\text{LCS}(X, Z)$

如图所示, 删除后得到的 Y' 与 Y 不等, 输出 False, 但是正确结果为 True.

2)

基本原理

定义状态变量 $f(i, j)$ 为 $X[1..i]$, $Y[1..j]$ 能否产生 $Z[1..i+j]$, 若能则为 True, 否则为 False. 于是整个问题的解为 $f[m][n]$.

状态转移方程:

- $f[0][j] = IY[1..j]$ 是否等于 $Z[1..j]$. 同理 $f[i][0] = IX[1..i]$ 是否等于 $Z[1..i]$. (可以在 $O(m+n)$ 的时间内预处理出结果)
- if $Z[i+j] = A[i]$ 且 $Z[i+j] = B[j]$, 此时不能判断是谁, 因此 $f[i][j] = f[i-1][j] \vee f[i][j-1]$;
- else if $Z[i+j] = A[i]$, $f[i][j] = f[i-1][j]$;
- else $f[i][j] = f[i][j-1]$;

分析

状态的个数为 $O(mn)$, 因此空间复杂度为 $O(mn)$, 预处理的时间复杂度为 $O(m+n)$, 因此总的时间复杂度为 $O(mn) \times O(1) + O(m+n) = O(mn)$.

3)

基本原理

定义状态变量 $f(i, j, k)$ 表示通过 $X[1..i]$, $Y[1..j]$ 得到 $Z[1..k]$ 需要删除的元素最少的删除元素集合。于是 $f(m,n,m+n)$ 就是整个问题的解。

状态转移方程：

- $f[0][0][0] = 2m + 2n;$
- if $Z[k] = A[i]$, $f[i][j][k] = \min(f[i][j][k], f[i-1][j][k-1]);$
- if $Z[k] = B[j]$, $f[i][j][k] = \min(f[i][j][k], f[i][j-1][k-1]);$
- if $i > 0$, $f[i][j][k] = \min(f[i][j][k], f[i-1][j][k] + X[i]);$
- if $j > 0$, $f[i][j][k] = \min(f[i][j][k], f[i][j-1][k] + Y[i]);$
- if $k > 0$, $f[i][j][k] = \min(f[i][j][k], f[i][j][k-1] + Z[i]);$

(上式的“ \min ”表示集合的大小比较, “ $=$ ”表示集合的赋值)

分析

状态的个数为 $O(mnk)$, 如果假设一个集合的空间大小为 $O(1)$, 集合的赋值/比较大小操作均为 $O(1)$, 则空间复杂度 = 时间复杂度 = $O(mnk)$.

P 13.12

1)

基本原理

定义状态变量 $f(i)$ 表示 $s[1..i]$ 子串能够重建为由合法单词组成的序列, 用 True/-False 表示。

状态转移方程：

- $f[0] = \text{True}$, 表示空字符串能够表示;
- $f[i] = \bigvee_{0 \leq j \leq i-1} (f[j] \wedge \text{dict}(s[j+1..i])).$

分析

状态的个数为 $O(n)$, 因此空间复杂度为 $O(n)$, 时间复杂度为 $O(n) \times O(n) = O(n^2)$.

2)

基本原理

初始化一个辅助数组 $\text{last}[1..n]$. $\text{last}[i]$ 表示如果子串 $s[1..i]$ 能够重建, 则 $s[1..i]$ 最后一个单词的索引的前一位, 否则, 用-1 表示.

在第 (1) 问的算法设计框架中, 在递推 $f[i]$ 时, 首先将 $\text{last}[i]$ 置-1. 如果存在 j , $f[j] = \text{dict}[s[j+1..i]] = \text{True}$, 则更新 $\text{last}[i] = j$;

最后, 如果 $\text{last}[n]$ 不为-1, 则可以通过 last 数组迭代找到每个单词的划分, 这一操作的时间复杂度为 $O(n)$.

P 13.13 最长回文子序列

1) 法一: DP

基本原理

定义状态 $f(i, j)$ 表示子问题 $S[i..j]$ 的最长回文子序列. 于是 $f(1, n)$ 即为原问题的解.

状态转移方程:

- $f[i][i] = 1$; $f[i][i+1] = 2 * \text{IS}[i] = S[i+1]$;
- if $S[i] = S[j]$, $f[i][j] = f[i+1][j-1] + 2$;
- else $f[i][j] = \max(f[i+1][j], f[i][j-1])$;

分析

状态的个数为 $O(n^2)$, 空间复杂度为 $O(n^2)$, 时间复杂度为 $O(n^2) \times O(1) = O(n^2)$.

2) 法二: 转化为 LCS

通过 $O(n)$ 时间将 S 翻转为 T , 然后求解 S 和 T 的 LCS 即可. 空间复杂度为 $O(mn)$, 时间复杂度为 $O(mn)$.

3) 回文串分解

基本原理

首先定义 $f(i, j)$ 表示子串 $S[i..j]$ 为回文串. 状态转移方程如下:

- $f[i][i] = \text{True}$;
- $f[i][i+1] = \text{IS}[i] = S[i+1]$;

- $f[i][j] = IS[i] = S[j] \wedge f[i+1][j-1];$

从而可以在 $O(n^2)$ 时间内得到每个子串是否为回文串. 接下来定义状态 $c(i, j)$ 表示将子串 $S[i..j]$ 分解成多个回文串的最小数量, 显然, $c(1, n)$ 为整个问题的解.

状态转移方程如下:

- if $f[i][j] = \text{True}$, $c[i][j] = 1;$
- else $f[i][j] = \min_{i \leq k < j} c[i][k] + c[k+1][j] + 1;$

分析

状态的个数为 $O(n^2)$, 因此空间复杂度为 $O(n^2)$, 时间复杂度为 $O(n^2)$.

P 13.14

1) 区间划分 DP

基本原理

区间划分型 DP. 枚举第一刀切在哪里, 得到两个子区间作为子问题. 定义状态 $f[i][j]$ 表示区间 $S[i..j]$ 的最小代价, 因此 $f[1][n]$ 即为整个问题的解. 但是考虑到可能 $m \ll n$, 因此对数组进行离散化. 根据切分的位置将数组改为 $m+1$ 个元素的数组, 每个元素的值等于对应最小区间的长度. 相应的, $f[i][j]$ 表示 $S'[i..j]$ 的最小代价. 于是整个问题的解就是 $f[1][m+1]$.

状态转移方程:

- $f[i][i] = 0.$
- $f[i][j] = \min(f[i][k] + f[k][j]) + \text{sum}(S'[i..j]).$

由于 $\text{sum}(S'[i..j])$ 与 k 的选取无关, 因此可以在 $O(m)$ 的时间内预处理算出, 这样后续可以做到 $O(1)$ 的状态转移. 因此时间复杂度为 $O(m) \times O(m^2) = O(m^3)$.

2) 贪心?

P 13.15 零钱兑换变式

1)

基本原理

(类似完全背包问题)

用 $f(i, t)$ 表示仅用 $X[1..i]$ 面值的硬币能够兑换总金额为 t , 若能则为 True, 否则为 False. 于是 $f(n, v)$ 就是整个问题的解.

状态转移方程如下:

- $f[0][0] = \text{True}$, $f[0][x] = \text{False}$, $x \neq 0$;
- $f[i][j] = f[i][j - X[i]] \vee f[i-1][j]$ if $j \geq X[i]$ else $f[i-1][j]$;

分析

状态的个数为 $O(nv)$, 算法的空间和时间复杂度均为 $O(nv)$.

2)

基本原理

(类似 01 背包问题)

用 $f(i, t)$ 表示仅用 $X[1..i]$ 面值的硬币能够兑换总金额为 t , 若能则为 True, 否则为 False. 于是 $f(n, v)$ 就是整个问题的解.

状态转移方程如下:

- $f[0][0] = \text{True}$, $f[0][x] = \text{False}$, $x \neq 0$;
- $f[i][j] = f[i-1][j - X[i]] \vee f[i-1][j]$ if $j \geq X[i]$ else $f[i-1][j]$;

分析

状态的个数为 $O(nv)$, 算法的空间和时间复杂度均为 $O(nv)$.

3)

基本原理

(类似双重背包问题)

用 $f(i, j, k)$ 表示用 $X[1..i]$ 部分面值的不超过 k 枚硬币能否表示面值 j , 如果能则为 True, 否则为 False. 显然, $f(n, v, k)$ 就是整个问题的解.

状态转移方程如下:

- $f[0][0][*] = \text{True}$, $f[0][x][*] = \text{False}$, $x \neq 0$;
- if $k > 0$ and $j \geq X[i]$: $f[i][j][k] = f[i][j - X[i]][k-1] \vee f[i-1][j][k]$;
- else: $f[i][j][k] = f[i-1][j][k]$.

分析

状态的个数为 $O(nvk)$, 算法的空间和时间复杂度均为 $O(nvk)$.

P 13.16 Vertex Cover

基本原理

即求树 T 的最小点覆盖集.

定义函数 $\text{dfs}(\text{cur}, \text{state})$, 表示以 cur 为根的子树在 cur 节点为 state 状态下的最小点覆盖集. 其中, $\text{state} = \text{True}$ 表示该点必须选择, False 表示该点不必选择. 显然, $\text{dfs}(\text{root}, \text{False})$ 是整个问题的解.

状态转移方程如下:

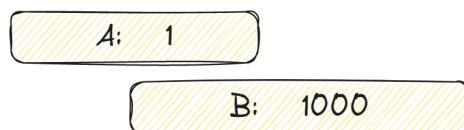
- if $\text{cur.state} = \text{True}$, $\text{ans} = 1 + \sum_u \text{DFS}(u, \text{False})$;
- if $\text{cur.state} = \text{False}$, $\text{ans} = \min(1 + \sum_u \text{DFS}(u, \text{False}), \sum_u \text{DFS}(u, \text{True}))$;

分析

用哈希表或者线性表记录每个状态的计算结果, 因此每个状态只会被计算一次, 因此时间复杂度为 $O(n)$.

P 13.17 带权任务调度问题

1) 贪心失败

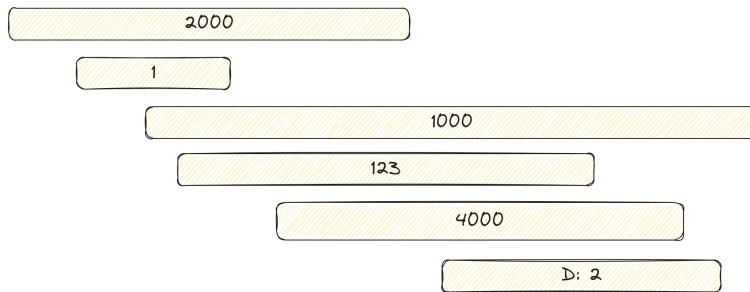


反例如图所示, 按照贪心思路, 会选择 A, 因而导致 B 无法选择, 显然这不是最优解.

2) $O(n^2)$ 解法

3) 动态规划解法

基本原理



如上图所示, 将所有元素按照开始时间进行排序, 时间复杂度为 $O(n \log n)$.

然后维护一个 $\text{next}[n]$ 数组, $\text{next}[i]$ 表示开始时间大于 $A[i]$ 的结束时间的所有元素. 由于将所有元素按照开始时间进行排序, 因此如果 $A[j]$ 满足, 则 $A[j]$ 后面所有的元素都满足, 因此可以根据 $A[i]$ 的结束时间二分查找找到第一个满足的元素下标 (如果没有, 则为 $n+1$), 这样预处理所有元素的 next 属性的时间复杂度为 $n \times O(\log n) = O(n \log n)$.

然后进行动态规划, 定义状态 $f[i]$ 表示 $A[i..n]$ 可选的情况下最大学分, 于是 $A[1]$ 就是整个问题的解. 状态转移方程:

- $f[n+1] = 0$ (表示没有可选的课程)
- $f[i] = \max(f[i+1], f[\text{next}[i]] + A[i])$.

因此 DP 的时间复杂度为 $O(1) \times O(n) = O(n)$.

因此总的时间复杂度为 $O(n \log n)$.

P 13.18

基本原理

划分型 DP, 定义状态 $f(i)$ 表示子问题 $A[1..i]$ 的最小惩罚. 显然, $f(n)$ 就是整个问题的解答.

状态转移方程如下:

- $f[0] = 0$;
- $f[i] = \min_{A[i] - A[k] \leq 200} f[k] + (200 + A[k] - A[i])^2$;

分析

状态的个数为 $O(n)$, 空间复杂度为 $O(n)$, 时间复杂度为 $O(n) \times O(n) = O(n^2)$.

P 13.19

基本原理

$O(n)$ 时间预处理出 $\text{next}[n]$ 数组, 其中 $\text{next}[i]$ 表示如果选择 $A[i]$, 那么下一个最早可以选择的元素下标. 由于间隔长度 k 为定值, 因此可以通过滑动窗口来线性预处理出所有的 next 属性.

然后定义状态 $f[i]$ 表示子问题: 数组 $A[i..n]$ 的最大理论, 于是 $A[1]$ 即为整个问题的解.

状态转移方程:

- $f[n+1] = 0;$
- $f[i] = \max(f[i+1], f[\text{next}[i]] + A[i]);$

状态转移的代价是 $O(1)$ 的, 因此 DP 的时间复杂度是 $O(n)$.

因此总的时间复杂度为 $O(n)$.

P 13.23

基本原理

树形 DP, 树上打家劫舍问题.

伪代码

Algorithm 32: BANQUET(root)

```
1 初始化: 二维数组  $f[i][state]$  用于记录以  $i$  为根节点的子树, 根节点的可选状态为  $state$  时的最大友好度, 初始化为 -1;
2 return DFS(root, True);
3 DFS(cur, state)begin
4     if  $cur = \text{NULL}$  then
5         return 0;
6     if  $f[cur][state] \neq -1$  then
7         return  $f[cur][state]$ ;
8     int ans := 0;
9     if  $state = \text{True}$  then
10        /* 说明可以选择该点 */
11        ans := cur.val + DFS(cur.left, False) + DFS(cur.right, False);
12        /* 总是可以选择不选该点 */
13        ans := max(ans, DFS(cur.left, True) + DFS(cur.right, True));
14    f[cur][state] := ans;
15    return ans;
```

分析

状态的个数为 $O(n)$, 每个状态只会被计算 1 次, 因此总的时间复杂度为 $O(n)$.

Chapt. 14 堆与偏序关系

P 14.1

$$\lceil \log(\lfloor \frac{1}{2}h \rfloor + 1) \rceil + 1 = \lceil \log(h + 1) \rceil$$

Pf. 首先, 由堆的定义, 显然有, 节点数为 h 的堆的高度为 $\lceil \log(h + 1) \rceil - 1$.

考虑一个由 h 个节点的堆, 注意到, 去除其所有叶子结点后得到的子树高度-1. (这是因为所有的叶子结点都在最后一层以及倒数第二层, 且倒数第二层至少有 1 个内节点)

又因为堆中至多有一个 1 度节点, 因此内节点的数目为 $\lfloor \frac{h}{2} \rfloor$, 因此堆的内节点构成的子树的高度可以表示为 $\lceil \log(\lfloor \frac{h}{2} \rfloor + 1) \rceil - 1$.

于是有 $\lceil \log(\lfloor \frac{h}{2} \rfloor + 1) \rceil - 1 + 1 = \lceil \log(h+1) \rceil - 1$, 即, $\lceil \log(\lfloor \frac{1}{2}h \rfloor + 1) \rceil + 1 = \lceil \log(h+1) \rceil$. \square

P 14.2

注：课上的讲的算法

由于前 k 大元素只可能出现在堆的前 2^k 个元素中, 因此只利用前 2^k 个元素进行建堆, 后面逐渐 pop 即可. 算法的时间复杂度为 $O(k \log 2^k) = O(k^2)$.

参考资料:<https://www.geeksforgeeks.org/k-th-greatest-element-in-a-max-heap/>

Algorithm 33: K-LARGEST(Heap h, k)

output: the k th largest element in the heap

```
1 初始化: Heap temp = null; /* 一个空的大顶堆 */  
2 temp.insert(h.top);  
3 for i ← 1 to k - 1 do  
4     int cur = temp.top;  
5     temp.pop;  
6     if cur.left ≠ null then  
7         temp.insert(cur.left);  
8     if cur.right ≠ null then  
9         temp.insert(cur.right);  
10    return temp.top;
```

算法正确性分析

首先证明, 每次 pop 出的元素依次为第 $1, 2, 3, \dots, k-1$ 大的元素.

由于堆能够维护最大的元素总是在 top 的性质, 只要说明, 在第 i 次 pop 前, 第 i 大的元素总是在 temp 堆中即可.

初始化 第一次 pop 前, temp 中只有 $h.top$, 由 h 堆的性质, $h.top$ 是第 1 大元素.

维持 对于第 m 次前的每一次 pop, 都弹出第 i 大的元素, 假设第 m 次弹出前, 第 m 大的元素不在 temp 堆中, 即, 第 m 大的元素的父节点不是第 $1, 2, \dots, m-1$ 大的元素, 那么第 m 大的元素的父节点至少是第 m 大的元素, 由堆的偏序性, 该元素至少是第 $m+1$ 大的元素, 这就构成矛盾. 因此第 m 次 pop 前, 第 m 大的元素一定在 temp 堆中.

终止 在第 k 次 pop 前, 第 k 大的元素一定在堆中.

注：说明

这里需要指出，“在堆中”是指至少已经进入过 temp 堆了，那么有没有可能在第 k 次 pop 之前，第 k 大的元素已经被 pop 出去了呢？显然是不可能的，由上述论证， $\forall i$, 第 i 次 pop 前，第 i 大的元素也已经在堆中，由堆的性质，pop 出去的一定是当前堆中的最大元素，因此可以归纳得到每一次 pop 严格弹出第 i 大的元素。

因此，在执行 $k-1$ 次 pop 之后（即第 k 次 pop 之前），temp 堆的 top 元素就是第 k 大的元素。□

时间复杂度分析

每次 Fix-heap 的时间复杂度为 $O(\log k)$ ，一共执行 $k-1$ 次，总的时间复杂度为 $O(k \log k)$ 。

P 14.3

对下标进行变换，令 $m = i - 1$ ，则两个结论转换为

1)

$$\text{Parent}(m) = \lfloor \frac{m-1}{d} \rfloor$$

2)

$$\text{Child}(m, j) = dm + j$$

先证明结论 2).

1)

由结论 2).，对于任意节点下标 $x = d \times m + j$, $\text{Parent}(x) = \lfloor \frac{dm+j-1}{d} \rfloor = m + \lfloor \frac{j-1}{d} \rfloor = m$ ，因此 $\text{Parent}(x) = \lfloor \frac{x-1}{d} \rfloor$

2)

通过数学归纳法证明，对 m 归纳，记 $P(m)$: $\forall j, 1 \leq j \leq d, \text{Child}(m, j) = dm + j$.

Basis. $P(0)$: 显然, $\text{Child}(0, j) = j$;

I.H. 假设对于 $\forall m = k, k \geq 0, P(k)$ 成立。

I.S. 对于 $P(k+1)$, 注意到由于堆的性质, $k+1$ 的子节点是紧接着 k 的最后一个子节点的，因此 $\text{Child}(k+1, j) = \text{Child}(k, d) + j = kd + d + j = (k+1)d + j$.

综上所述, $\forall k \geq 0, P(k)$ 成立，即原命题成立.□

P 14.4

1) 法一(失败)

记堆中的元素依次标号 $1, 2, \dots, n$, 对应节点的高度为 H_i , 堆的所有节点的总高度为 H .

首先证明, 对于高度为 h 且具有最多节点的堆(不妨称为”满堆”, 即 $n = 2^{h+1} - 1$), 当 h 增加 1 时(即增加了 2^{h+1} 个节点), 有 $\Delta H = \Delta n - 1$.

用数学归纳法, 对 h 归纳:

Basis. $h = -1 \rightarrow h = 0, \Delta H = 0, \Delta n = 1$, 满足.($h = -1$ 即为空堆)

I.H. 假设对于任意的 $k(k \geq 0), h = k \rightarrow h = k + 1$, 均有 $\Delta H = \Delta n - 1$ 成立,

I.S. 则对于 $h = k + 1 \rightarrow h = k + 2$, 注意到, 相比于 $h = k \rightarrow k + 1$ 的过程, $\Delta n' = 2 \times \Delta n = 2^{k+2}$, 可以将这次的增加看做两个部分, 即 root 的左子树增加了 2^{k+1} 个节点, 以及 root 的右子树增加了 2^{k+1} 个节点, 这个两个子过程与 $h = k \rightarrow k + 1$ 完全相同, 因此由假设可得, 对祖先节点的高度增长的贡献为 $2 \times \Delta H = 2\Delta n - 2$, 此外, 注意到这样的理解没有考虑 root 高度的变化, root 的高度同样 +1, 因此总的 $\Delta H' = 2 \times \Delta n - 2 + 1 = 2 \times \Delta n - 1 = \Delta n' - 1$.

综上所述, 对于”满堆”, 高度每增加一层, 增加的高度就相比增加的节点数少一个. 将 $k = -1 \rightarrow 0, k = 0 \rightarrow 1, \dots, k = h - 1 \rightarrow h$ 累加起来, 得到对于高度为 h 的”满堆”, $n - H = h + 1$.

下面, 考虑仅在高度为 h 的”满堆”上增加一个节点, 这样会得到一棵高度 +1 的树($h' = h + 1$), 显然, 会且仅会使得该节点的所有祖先节点的高度 +1, 那么对整个堆的高度和的贡献为 $h' = h + 1$, 但是又因为增加了一个顶点数, 因此在该节点加入后, $n' - H' = (n + 1) - (H + h + 1) = 1$, 即 $H' = n' - 1$.

下面证明对于向当前的堆上继续增加节点直到再次成为”满堆”的过程, $H < n - 1$.

//TODO(初步的思路是, 说明总是”入不敷出”)

注

上述方法在解释非完美二叉树的地方说不清楚, 失败. 但是取得的进展在于证明了在比完美二叉树恰多一个节点时, $n - H$ 取到最大值.

2) 法二 (上课介绍)

引理：完美二叉树满足 $n - H = h + 1$

利用数学归纳法证明如下，对 h 归纳，记 $P(n)$ ：对于高度为 h 的完美二叉树，有 $H - n = h$.

Basis. $P(0)$, $n = 1$, $h = 0$, $H = 0$, 成立.

I.H. 假设 $\forall k \geq 0$, $P(n = k)$ 成立.

I.S. 对于 $n = k + 1$, 注意到对于完美二叉树，有 $n = 2^{h+1} - 1$, 因此 $n' - n = 2^{k+2} - 1 - (2^{k+1} - 1) = 2^{k+1}$, 而树的高度增加一层后，原先的所有节点的高度均 +1，因此 $H' = H + n$, 因此 $n' - H' = (n + 2^{k+1}) - (H + n) = (n - H) + (2^{k+1} - n) = k + 1 + 1 = (k + 1) + 1$.

综上所述, $\forall k \geq 0$, $P(k)$ 成立. \square

由引理容易证明对于堆是完美二叉树的情况。下面证明对于非完美二叉树的情况，可以视作是在一棵完美二叉树下一层紧密排列的一些节点（但未排满）。注意到这样的二叉树具有一个性质：考虑当前根节点的左右子树，其中有一棵为完美二叉树，另一棵是堆。

（注意到，对于完美二叉树的堆，上述性质同样成立。）

证明如下：对堆进行分类讨论：

- 最底层节点数 $< 2^{h-1}$ 由于堆的定义，其右子树必然是一棵高度为 $h - 2$ 的完美二叉树，而左子树高度为 $h - 1$ ，且同样满足最后一层节点从左到右紧密排列，其余层均排满。根据堆的定义，左子树是堆。
- 最底层节点数 $\geq 2^{h-1}$ 由于堆的定义，其左子树必然是一棵高度为 $h - 1$ 的完美二叉树，而右子树高度为 $h - 1$ ，最底层的节点满足从左到右紧密排列，且其余层均排满，根据堆的定义，右子树是堆。

下面用数学归纳法证明对于非完美二叉树的情形也满足。对树的高度归纳，记 $P(n)$ ：高度为 n 的堆满足 $H \leq n - 1$ 。

Basis. $P(0)$, 显然成立。

I.H. 假设 $\forall k \geq 0$, $P(n=k)$ 成立。

I.S. 对于 $n = k+1$, 根据性质，可以对 root 的左右子树进行讨论：

- 左子树为堆，右子树为完美二叉树。此时 $H(\text{root} \rightarrow \text{left}) \leq n(\text{root} \rightarrow \text{left}) - 1$ （归纳假设）， $H(\text{root} \rightarrow \text{right}) = n(\text{root} \rightarrow \text{right}) - (h - 2 + 1)$ （引理），于是有 $H = h + H(\text{root} \rightarrow \text{left}) + H(\text{root} \rightarrow \text{right}) \leq h + n(\text{root} \rightarrow \text{left}) - 1 + n(\text{root} \rightarrow \text{right}) - (h - 2 + 1) = n(\text{root} \rightarrow \text{left}) + n(\text{root} \rightarrow \text{right}) = n - 1$
- 左子树为完美二叉树，右子树为堆。此时 $H(\text{root} \rightarrow \text{right}) \leq n(\text{root} \rightarrow \text{right}) - 1$ （归纳假设）， $H(\text{root} \rightarrow \text{left}) = n(\text{root} \rightarrow \text{left}) - (h - 1 + 1)$ （引理），于是有

$$H = h + H(\text{root} \rightarrow \text{right}) + H(\text{root} \rightarrow \text{left}) \leq h + n(\text{root} \rightarrow \text{right}) - 1 + n(\text{root} \rightarrow \text{left}) - (h - 1 + 1) = n(\text{root} \rightarrow \text{left}) + n(\text{root} \rightarrow \text{right}) - 1 = n - 2 < n - 1.$$

综上所述, $\forall k \geq 0$, $P(k)$ 成立. 即原命题成立. \square

3) 法三 (上课介绍)

证明. 记在一个堆中, 所有节点的高度和为 h , 节点数为 n . 首先证明引理.

引理：对于任意堆, 叶节点的数量大于等于内节点的数量

证明. 即证明堆满足 $n_0 \geq n_1 + n_2$.

首先对于二叉树, 有边数 $|E| = n_1 + 2n_2$, 又因为对于树, 满足 $|V| = |E| + 1$, 于是有 $n_0 + n_1 + n_2 = n_1 + 2n_2 + 1 \Rightarrow n_0 = n_2 + 1$.

又, 由堆的定义, $0 \sim h-1$ 层全部填满, h 层从左到右紧密排列, 因此显然 $0 \sim h-2$ 层所有节点都是 2 度节点, h 层都是 0 度节点, 而因为 h 层节点紧密排列, 因此 $h-1$ 层至多只有 1 个 1 度节点. 因此, 在整个堆中, $n_1 \leq 1$.

于是, $n_0 \geq n_2 + 1 \geq n_1 + n_2$. \square

下面考察一个堆 H , 定义堆的层数 L 如下:

$$L(H) = \begin{cases} 1, & H \text{ 为仅含一个根节点的堆,} \\ L(H') + 1, & H' = H \text{ 去掉所有叶子节点后得到的堆.} \end{cases}$$

注

这里略过了一个堆删除所有叶子节点后得到的依然是堆的证明.

然后, 定义节点 v 的层数 $l(v) \equiv L(H), v$ 为 H 的叶子节点.

于是, 对于任意给定的非空的堆, 我们可以将其所有节点按照其层数进行划分. 注意到上述逐渐删除叶子节点过程的逆过程就是递推得到当前堆的过程, 即, 可以将当前的堆看做是从层数为 $1, 2, \dots, L(H) - 1$ 的对应的堆 $H_1, H_2, \dots, H_{L(H)-1}$ 不断增加叶子节点得到. 为了证明对于当前的堆 $H_{L(H)}$ 满足 $h \leq n - 1$, 下面对堆的层数进行归纳: 记 $P(k)$: 堆 H_k 满足 $h \leq n - 1$.

Basis. $P(1)$: 即仅含有一个根节点的堆, 此时 $n=1, h = 0$, 满足.

I.H. 假设对于 $k \geq 1$, $P(k)$ 成立

I.S. $P(n = k+1)$, 对于堆 H_{k+1} , 可以视为是从 H_k 增加了 $l = k + 1$ 的节点得到, 这些节点的高度为 0, 其余所有的节点的高度均 +1, 因此 $h' = h + n$. 又因为 H_k 中所有的节点都不是 H_{k+1} 中的叶子节点 (否则就会被删除了), 因此 H_{k+1} 中的所有叶子节点就是从 H_k 到 H_{k+1} 增加的节点, 且 H_{k+1} 中的其余节点均为内节点, 由引理, $\Delta n = n' - n = n_0(H_{k+1}) \geq n_1(H_{k+1}) + n_2(H_{k+1}) = n$. 因此 $n' - h' = (n + \Delta n) - (h + n) = \Delta n - h \geq n - h \geq 1$, 即 $h' \leq n' - 1$.

综上所述, $\forall k \geq 1$, $P(k)$ 均成立. 即对于任意堆, $h \leq n - 1$.

□

P 14.5 k 个链表的合并

1) 法一

将每个链表的表头按照表头对应的下一个元素大小构建最小堆, 每次取出并替换堆顶元素, 并 $\text{fix-heap}(O(\log k))$, 因此总的时间复杂度为 $O(n \log k)$.

2) 法二

利用与合并链表相似的方法, 归并排序合并分别合并左右两边的链表, 最后以 $O(\alpha k)$ 的时间复杂度合并左右链表, $\alpha = \frac{n}{k}$. 需要额外开一个数组来记录表头, 方便跳转.

$$T(k) = 2T(k/2) + O(\alpha k)$$

$$\text{令 } S(k) = T(k)/\alpha$$

$$S(k) = 2S(k/2) + O(k)$$

$$\Rightarrow S(k) = O(k \log k)$$

$$\Rightarrow T(k) = O(n \log k)$$

P 14.6

假设有 n 个元素 $A[1..n]$,

- n 为奇数, 则中位数 $m = A[\lceil \frac{n}{2} \rceil]$
- n 为偶数, 则中位数 $m = \frac{A[n/2] + A[n/2 + 1]}{2}$

因此维护一个大顶堆和一个小顶堆, 假设已经插入的元素有 n 个, 为 $A[1..n]$ (升序排列), 则大顶堆中维护 $A[1..\lceil \frac{n}{2} \rceil]$, 小顶堆中维护 $A[\lceil \frac{n}{2} \rceil + 1..n]$, 对插入, 删除和求中位数的操作描述如下:

Algorithm 34: TWO-HEAP-OPERATIONS

```
1 初始化: Max-Heap maxh = null, Min-heap minh = null; /* 空的大顶堆和小顶堆
   */
2 INSERT(val) begin
3   if maxh.size = minh.size then
4     maxh.insert(val);
5   else
6     // 说明此时大顶堆元素个数比小顶堆多 1 个
7     minH.insert(val);
8
9 DELETE(val) begin
10   if val is in maxh then maxh.delete(val);
11   else if val is in minh then minh.delete(val);
12   else
13     // 说明元素不存在
14     return ;
15
16 QUERY-MEDIAN begin
17   if maxh.size = minh.size then
18     return (maxh.top + minh.top) / 2;
19   else
20     return maxh.top;
```

注 : Heap with handler

若要求能以 \log 级别时间复杂度实现删除任意元素，则需要对大顶堆和小顶堆作出补充：要求能以 $O(1)$ 时间查询堆中元素的位置，可以通过增加一个 hashmap 来进行维护。

Chapt. 15 并查集与动态等价关系

P 15.1 并查集的蛮力实现

1) 基于矩阵

伪代码

Algorithm 35: UFS-MATRIX(A[1..n]), Q[1..l]

```
1 初始化: 二维布尔数组  $F[1..n][1..n]$ ,  $F[i][j] = \text{True}$  表示  $i, j$  为同类, 否则不同  
类. 初始时仅有  $A[i][i] = \text{True}$ , 其余为  $\text{False}$  output: 输出查找的  
结果  
2 FIND( $k$ ) begin  
3   /* 找到所有类别索引与第  $k$  个元素相同的元素 */  
4   for  $i := 1$  to  $n$  do  
5     if  $A[i][k] = \text{True}$  then  
6       输出  $i$ ;  
7  
6 UNION( $a, b$ ) begin  
7   /* 将第  $a$  个元素代表的类和第  $b$  个元素代表的类合并 */  
8   for  $i := 1$  to  $n$  do  
9     if  $A[i][a] = \text{True}$  then  
10       $A[i] = A[i] \vee A[b]$ ;  
11    else if  $A[i][b] = \text{True}$  then  
12       $A[i] = A[i] \vee A[a]$ ;
```

时间复杂度分析

对于查找操作, 时间复杂度为 $O(1)$.

对于合并操作, 时间复杂度为 $O(n^2)$. 如果采用 bit 数组作为矩阵的每一行, 并假设一个 bit 数组的单次或运算时间复杂度为 $O(1)$, 则可以认为合并操作的时间复杂度为 $O(n)$.

2) 基于数组

伪代码

Algorithm 36: UFS-ARRAY(A[1..n]), Q[1..l]

```
1 初始化: 数组  $F[1..n]$ ,  $F[i] = i$ , 表示每个元素归属的类索引;  
    output: 输出查找的结果  
2 FIND( $k$ ) begin  
    /* 找到所有类别索引与第  $k$  个元素相同的元素 */  
3     int key = A[ $k$ ];  
4     for  $i := 1$  to  $n$  do  
5         if  $A[i] = key$  then  
6             输出  $i$ ;  
7 UNION( $a$ ,  $b$ ) begin  
    /* 将第  $a$  个元素代表的类和第  $b$  个元素代表的类合并 */  
8     int p1 = A[ $a$ ];  
9     int p2 = A[ $b$ ];  
10    for  $i := 1$  to  $n$  do  
11        if  $A[i] = p2$  then  
12            A[i] = p1;
```

时间复杂度分析

对于任意查找和合并操作, 都需要遍历一次数组, 因此单次查询的时间复杂度是 $\Theta(n)$, 总的查询的时间复杂度为 $\Theta(nl) = O(nl)$.

P 15.3

基本原理

首先将所有的约束关系进行分类, 用 $O(l)$ 的时间复杂度将所有的关系分为相等的一组和不等的一组. 然后对于所有的相等关系维护一个并查集, 最后以不等关系为输入进行 check, 如果发现不等关系的两个元素已经属于同一个划分, 则说明不可能满足, 如果全部的 check 都通过, 则说明这些约束能够同时满足.

伪代码

Algorithm 37: CONSTRAINT-RELATION(R[1..l])

Output: 这一组约束关系能否同时满足, 如果能则输出 True, 否则输出 False

```
1 初始化: 并查集  $F[1..n]$ , 初始时所有元素的代表元素即为自己;
2 将所有约束关系分为相等和不等的两部分:  $E, NE$ ;
3 foreach  $R(a, b) \in N$  do
4   UNION(( $a, b$ ));
5 foreach  $R(a, b) \in NE$  do
6   if  $\text{FIND } (a) = \text{FIND } (b)$  then
7     输出 False;
8   return;
9 输出 True;
10 return ;
```

正确性分析

首先交换关系的输入顺序显然不会影响最终结果, 因此将所有的关系划分为相等和不等关系. 对于首先输入的相等关系, 显然任意的输入总是不会产生矛盾. 对于所有的相等关系输入完毕后, 所有的相等关系就形成的对所有元素的一个划分, 不等关系只能发生在不同等价类的元素之间. 因此接下来的不等关系就检查这两个元素是否属于同一个等价类, 如果属于, 则说明这两个元素既相等又不等, 发生矛盾. 如果所有的 check 均通过, 则说明所有的约束能够同时满足.

时间复杂度分析

基于并查集实现, 如果并查集采用按秩合并和路径压缩, 则时间复杂度为 $O((n + l) \log^* n) \approx O(n + l)$.

Chapt. 16 哈希表与查找

P 16.1

证明. 将所有元素的插入结果 (即插入到哪一个位置的索引) 按照插入顺序排成一个序列, 即得到了一个长度为 n 的串, 每个元素的范围是 $1..n$, 假设这个特定的位置的索引为 x , 即要求这 n 个元素中恰有 k 个为 x . 因此, 首先确定这 k 个元素, 有 $\binom{n}{k}$ 种可

能, 要求这 k 个元素的值为 x , 且其余不为 x , 概率是 $(\frac{1}{n})^k(1 - \frac{1}{n})^{n-k}$, 因此總的概率为 $(\frac{1}{n})^k(1 - \frac{1}{n})^{n-k} \binom{n}{k}$.

□

P 16.4 不同哈希表的存储效率

1)

- $\alpha = 0.25$. 则需要 $0.25h_c$ 个节点, 因此消耗的空间为 $(1 + 2\alpha)h_c = 1.5h_c$ 个单位的存储空间, 此时用开放寻址法时, 负载因子是 $\frac{\alpha h_c}{(1+2\alpha)h_c} = \frac{\alpha}{1+2\alpha} = \frac{1}{6}$.
- $\alpha = 0.5$, 此时需要的空间为 $2h_c$, 负载因子是 $\frac{1}{4}$.
- $\alpha = 1$, 此时需要的空间为 $3h_c$, 负载因子是 $\frac{1}{3}$.
- $\alpha = 2$, 此时需要的空间为 $5h_c$, 负载因子是 $\frac{2}{5}$.

2)

对于封闭寻址法, 若有 h_c 大小的表头, 则需要有 αh_c 个节点, 因此總的空间为 $h_c + 5\alpha h_c = (1 + 5\alpha)h_c$ 的空间, 对于开放寻址法, 此时的负载因子是 $\frac{4\alpha}{1 + 5\alpha}$.

α	0.25	0.5	1	2
空间	2.25	3.5	6	11
负载因子	4/9	4/7	2/3	8/11

Chapt. 18 平摊分析

P 18.1 湖景房

1) 算法正确性分析

证明. 循环不变量第 i 次迭代开始前, S 满足所有的元素自栈底到栈顶降序排列. 且 $A[1..i-1]$ 中的元素是“局部湖景房”元素 (即以 $A[1..i-1]$ 为问题规模时的湖景房元素) 当且仅当该元素在栈 S 中.

初始化第 1 次迭代开始前, S 为空, $A[1..0]$ 同样为空. 满足.

维持假设 第 i 次迭代开始前, 满足循环不变量, 则对于第 i 次迭代, 根据算法, $A[i]$ 会不断与栈顶元素比较并弹出较小的栈顶元素直到栈空或栈顶元素大于 $A[i]$, 最后 $\text{push}A[i]$. 如果当前迭代结束后, S 中不满足单调性, 则必然是 S 中栈顶下面的一个元素大于栈顶元素 $A[i]$ (因为其余位置已经根据假设满足了降序性, 而 pop 操作不会改变这一性质), 而这与算法矛盾, 因为 $A[i]$ 入栈时就保证了栈空或者大于栈顶元素, 因此第 i 次迭代后, S 中元素仍然满足自栈底到栈顶降序. 再考虑对于元素 $A[1..i-1]$ 在栈中的元素 x , 如果在本轮中出栈, 则说明满足 $A[i] > x$, 因此 x 不可能是 $A[1..i]$ 中的“局部湖景房”. 考虑在本轮中没有出栈的元素, 根据单调性以及栈顶元素大于 $A[i]$, 因此这些元素均大于 $A[i]$, 根据假设, 这些元素已经大于 $A[1..i-1]$ 中它们右边的所有元素, 因此也就大于 $A[1..i]$ 中它们右边的所有元素, 即为 $A[1..i]$ 中的“局部湖景房”元素. 因此第 i 次迭代后, $A[1..i]$ 中的元素是“局部湖景房”元素当且仅当该元素在栈 S 中.

终止 当 n 轮迭代结束(第 $n+1$ 轮迭代开始前), 根据不变量, $A[1..n]$ 中的元素是“局部湖景房”元素当且仅当该元素在栈中. 注意到此时的局部 = 全局, 因此 $A[1..n]$ 中的元素是“湖景房”元素当且仅当该元素在栈中. \square

2)

法一：对操作平摊

该算法涉及两种操作:

- 将 $A[i]$ 元素入栈. $C_{act} = 1$, 为了后面出栈的考虑, 额外多计算 1 次操作用于支付后面出栈的代价, 因此 $C_{acc} = 1$, $C_{amo} = 1 + 1 = 2$.
- 将当前的栈顶元素不断弹出直至栈空或者栈顶元素大于 $A[i]$, 假设此时有 k 个元素在栈中, 这个操作的代价可能是 0(若第一个元素就大于 $A[i]$), 也可能是 k (即全部弹出), 但是弹出的元素一定不超过 k 个, 由于所有入栈的元素都已经额外支付过 1 个单位的代价用于支付该元素出栈, 因此假设该操作弹出了 x 个元素, $C_{act} = x$, $C_{acc} = -x$, 因此 $C_{amo} = 0$.

但是由于并不一定所有的元素最后都被弹出, 因此 $\sum C_{acc} > 0$, 因此 $T(n) = \sum C_{act} < \sum C_{amo} = 2n = O(n)$.

法二：对元素平摊

每个元素进且只进一次栈, 至多出一次栈. 因此平摊到每个元素的操作不超过 2. 因此平摊代价为 $O(1)$.

P 18.3

注：参考资料

建议阅读这篇文章：<https://jeffe.cs.illinois.edu/teaching/algorithms/notes/09-amortize.pdf>

假设初始时数组长度为 0, 进行了 n 次插入操作, 最后得到长度为 n 的数组. 下面证明, 单次插入操作的分摊代价 C_{amo} 为 $1 + 2 \times \frac{1}{2} + 2^2 \times \frac{1}{2^2} + \dots + 2^{\lfloor \log n \rfloor} \times \frac{1}{2^{\lfloor \log n \rfloor}} = 1 + \lfloor \log n \rfloor$.

证明. 只需要证明, 对于任意的 k 次操作 ($k \leq n$) 之后, 分摊代价总是大于等于实际代价, 即 $\sum C_{act} = \sum C_{amo} - \sum C_{act} \geq 0$.

注意到, 在进行了 k 次插入操作后, 总共产生了 k 次产生包含 1 个元素的新数组的操作, $\lfloor \frac{k}{2} \rfloor$ 次将两个长度为 1 的数组合并的操作, $\lfloor \frac{k}{4} \rfloor$ 次将两个长度为 2 的数组合并的操作, \dots , $\lfloor \frac{k}{2^i} \rfloor$ 次将两个长度为 2^{i-1} 的数组合并的操作 ($k \geq 2^i$). 因此, $i_{max} = \lfloor \log k \rfloor$.

因此,

$$\sum_{i=1}^k C_{act,i} = k + \lfloor \frac{k}{2} \rfloor \times 2 + \lfloor \frac{k}{4} \rfloor \times 4 + \dots + \lfloor \frac{k}{2^{\lfloor \log k \rfloor}} \rfloor \times 2^{\lfloor \log k \rfloor}$$

$$\sum_{i=1}^k C_{amo,i} = k + \frac{k}{2} \times 2 + \frac{k}{4} \times 4 + \dots + \lfloor \frac{k}{2^{\lfloor \log k \rfloor}} \rfloor \times 2^{\lfloor \log k \rfloor} + \dots + \frac{k}{2^{\lfloor \log n \rfloor}} \times 2^{\lfloor \log n \rfloor}$$

不难注意到, $\forall i \leq k, C_{amo,i} \geq C_{act,i}$, 因此 $\forall k \leq n, \sum C_{amo} \geq \sum C_{act}$, 即平摊分析合理.

□

因此, 插入操作的平摊代价为 $1 + \lfloor \log n \rfloor = O(\log n)$.

P 18.5

基本原理

选择数组作为数据结构, 每次插入就直接在最末尾 append 新增元素. 删除操作时, 首先使用线性时间选择算法找到第 $\lfloor \frac{|S|}{2} \rfloor$ 大的元素作为 pivot, 然后再使用 partition 算法将数组进行划分, 然后删去后一半的元素 (直接将指向末尾的指针指向 pivot 即可)

伪代码

Algorithm 38: MULTISSET

```

1 初始化,  $A[1..n]$ , 初始时  $n = 0$ ;
2 INSERT( $S, x$ ) begin
3    $n := n + 1$ ;
4    $A[n+1] := x$ ;
5 DEL-LARGER-HALF( $S$ ) begin
6   int pivot := SEL ( $A, \lfloor \frac{|S|}{2} \rfloor$ ); /* 选择第  $\lfloor \frac{|S|}{2} \rfloor$  大的元素 */
7    $A := \text{PART} (\text{pivot})$ ; /* 按照该元素将数组划分 */
8    $n := \text{pivot}$ ; /* 舍弃后半部分的  $\lceil \frac{|S|}{2} \rceil$  个元素 */

```

时间复杂度分析

显然, 对于 INSERT 操作, $C_{act} = 1$, 对于 DEL-LARGER-HALF 操作, 由于 SEL 算法和 PART 算法都是 $O(|S|)$ 的, 因此 DEL-LARGER-HALF 操作的 $C_{act} = c|S|$, 其中 c 为某个常数.

假定 INSERT 操作的 $C_{acc} = 2c$, DEL-LARGER-HALF 操作的 $C_{acc} = -c|S|$. 于是, INSERT 操作的 $C_{amo} = 1 + 2c$, DEL-LARGER-HALF 操作的 $C_{amo} = 0$.

下面证明, 对于任意的操作序列, $\sum C_{acc} \geq 0$.

显然, 对于 INSERT 操作, 总是会使得 $\sum C_{acc}$ 增加, 而对于 DEL-LARGER-HALF 操作, 总是会让 $\sum C_{acc}$ 下降, 因此只要证明对于任意状态, 即使下面的所有操作均为 DEL-LARGER-HALF, 最终也能满足 $\sum C_{acc} \geq 0$.

首先证明引理.

引理 : 对于任意时刻, $\sum C_{acc} \geq 2|S|$

显然, 如果只进行 INSERT 操作, 则每一次 $|S|$ 增加 1, 同时 $\sum C_{acc}$ 增加 $2C$. 又因为 $|S| = 0$ 时 $\sum C_{acc} = 0$, 因此容易通过数学归纳法证明: 如果是进行 INSERT 操作, 有 $\sum C_{acc} \geq 2|S|$.

下面考虑插入 DEL-LARGER-HALF 操作, 注意到每次该操作都使得 $\sum C_{acc}$ 下降 $c|S|$, 同时使得 $|S|$ 下降 $\lceil \frac{|S|}{2} \rceil$. 由于第一次执行 DEL-LARGER-HALF 操作前的操作均为 INSERT 操作, 根据上述分析, $\sum C_{acc} \geq 2|S|$. DEL-LARGER-HALF 操作结束后, $\sum C'_{acc} = \sum C_{acc} - c|S| \geq 2c|S| - c|S| = c|S|$, $|S'| \leq \frac{|S|}{2}$, 于是 $\sum C'_{acc} \geq c|S| \geq 2|S'|$.

于是, 对于任意时刻, $\sum C_{acc} \geq 2|S|$, 容易证明, 即使接下来的所有操作均为 DEL-LARGER-HALF 操作, 使得 $\sum C_{acc}$ 下降的量不超过 $c|S| + \frac{1}{2}c|S| + \frac{1}{4}c|S| + \dots < 2c|S|$, 于是最终仍然有 $\sum C'_{acc} \geq 0$.

于是, 该算法单次操作的平摊代价为 $O(1)$.

Chapt. 19 对手论证

P 19.1

注：参考资料

关于对手论证: <https://www-student.cse.buffalo.edu/~atri/cse331/support/lower-bound/index.html>

证明. 决策树是一棵二叉树, 对于二叉树, 有如下引理:

引理

假设一棵二叉树的高度为 h , 叶节点的个数为 L , 它们之间的关系为 $L \leq 2^h$.
本引理的证明见 P4.1.

决策树的叶节点对应于选择的结果, 因此决策树的叶节点至少有 n 个. 于是有:

$$n \leq L \leq 2^h$$

于是, $h \geq \log n = \Omega(\log n)$.

□

P 19.3

证明. 首先证明一个性质: 记数组中唯一存在的逆序对元素下标为 $i, j (i < j)$, 有 $j = i + 1$.

证明. 反证法, 假设 $j \neq i + 1$, 则说明 i, j 之间存在别的元素, 记该序列为 $i, m_1, m_2, \dots, m_k, j$, 则由于只存在一个逆序对, 因此由 $A[i] < A[m_1] < A[m_2] < \dots < A[m_k] < A[j]$, 这与 $A[i] > A[j]$ 矛盾. □

因此, 可以对任意基于比较的算法的比较进行分类: 记每次比较的两个元素下标为 $i, j (i < j)$.

- $j > i + 1$. 根据这个性质, $A[i]$ 总是小于 $A[j]$, 因而比较不会产生任何信息.
- $j = i + 1$. 如果发现 $A[i] > A[j]$, 则算法结束, 直接交换这两个元素即可. 否则, 则排除了一个可能的逆序对位置.

根据性质, 所有可能的逆序对 (i, j) 的位置有 $n-1$ 个. 因此可以设计一种算法: 迭代 $i := 1 \rightarrow n-2$, 比较 $A[i]$ 与 $A[i+1]$, 如果出现 $A[i] > A[i+1]$, 则说明找到了逆序对, 交换两个元素后算法结束. 否则说明逆序对一定出现在 $A[n-1]$ 与 $A[n]$ 之间, 交换这两个元素, 算法结束.

至此, 对手论证失败. 因为找到了一种在最坏情况下比较次数少于 $n-1$ 的算法. \square

注 : 第二种结局

如果不考虑上述的算法设计 (比如题目改为至多存在一个逆序对), 则需要检查所有的 $n-1$ 的可能出现的位置, 此时必须比较这 $n-1$ 对元素, 才能确保算法的正确性. 因此对于任意算法, 最坏情况下的比较次数至少为 $n-1$ 次.

注 : 题目勘误

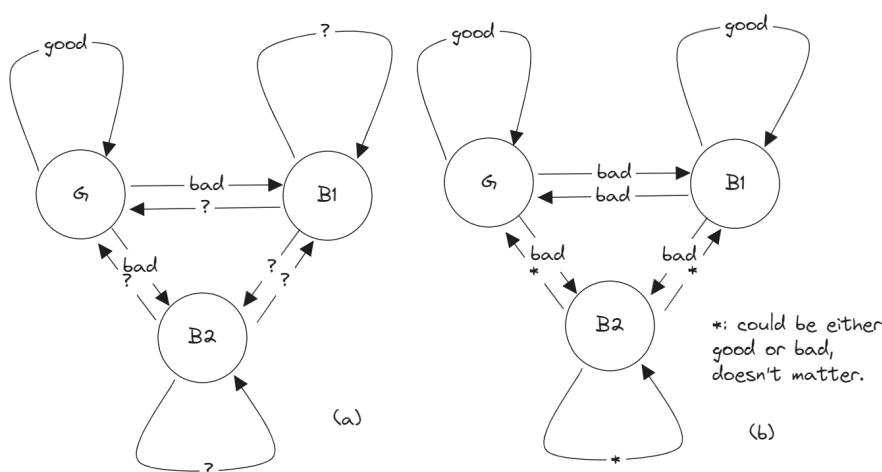
本书作者已经在勘误中修正了这个题目的表述: 至多存在一个逆序对.

P 19.4 好坏芯片的对手论证

如果坏芯片的数目不少于总数的一半, 则任何算法都不能确保所有芯片的好坏.

证明. 参考资料: <https://walkccc.me/CLRS/Chap04/Problems/4-5/>

不妨记所有好的芯片数为 g , 坏的芯片数为 b , 于是有 $g \leq b$, 因此总是在坏的芯片里面划分出一组与好的芯片 (将这些好芯片记为 G) 数量相同的芯片, 将其记为 B_1 , 其余的芯片记为 B_2 . 现在, 对两组芯片进行测试, 可以得到如图 (a) 的情况. 可以通过规定坏芯片的指派得到如图 (b) 的情况, 此时 G 和 B_1 两组芯片完全对称, 因此无法区分.



\square

P 19.5

1) 找元素最大值

基本原理

类似锦标赛排序, 对所有元素按 5 个一组 (多出来的如果能够轮空就直接轮空, 否则用 $+\infty$ 填满 5 个元素). 每组调用一次 SORT-FIVE 算法, 每组产生一个最大值, 重复上述过程, 直到最后只剩下一组, 最后调用一次 SORT-FIVE 算法就能够得到最大值.

伪代码

Algorithm 39: FINDMAX(A[1..n])

output: the max element in A[1..n]

```
1 while  $n > 1$  do
2   将所有元素每 5 个分为一组, 多出来的如果能够轮空就直接轮空, 否则用
     $+\infty$  填满 5 个元素, 调用一次 SORT-FIVE, 得到每组的最大值记为
     $B[1..n/5]$ ;
3    $A \leftarrow B$ ;
4    $n \leftarrow n / 5$ ;
5 return  $A[1]$ ;
```

时间复杂度分析

将整个过程用决策树表示, 根据比较的规则, 这棵决策树为一个 5-tree. 所有的叶子节点表示每个元素, 所有的内节点都是通过比较得出的优胜者. 因此容易计算得出 5-tree 的内节点数 n_5 与叶子节点 n_0 满足: $n_5 = \lceil \frac{n_0-1}{4} \rceil$, 而每个内节点恰好对应一次 SORT-FIVE 调用, 因此总的调用次数为 $\lceil \frac{n-1}{4} \rceil$.

最优化证明

证明. 注意到, 如果算法最终得出的最大元素 (记为 x), 则意味着 x 与其余所有的元素均直接或间接比较证明了 $\forall y \neq x, x > y$. 且 x 必然在每一次 SORT-FIVE 中都是最大元素. 对于一次 SORT-FIVE, 可以比较出 x 大于其余 4 个元素. 不妨假设每当直接获间接说明一个元素小于 x 时, 对该元素标记为 L, 则算法结束时, 这 $n-1$ 个元素必须都标记上了 L. 对于其余的 $n-1$ 个元素, 每次调用 SORT-FIVE, 最多可以新增 4 个 L, 因此至少需要调用 $\lceil \frac{n-1}{4} \rceil$ 才可以得到 $n-1$ 个 L, 因此调用次数至少为 $\lceil \frac{n-1}{4} \rceil$.

因此说明该算法已经达到了问题的下界, 因此该算法是最优的. □

2) 找第二大元素

基本原理

首先调用 FIND-MAX 算法, 以 $\lceil \frac{n-1}{4} \rceil$ 次 SORT-FIVE 调用得出最大元素 x , 然后选出 x 参与过的每一次的 SORT-FIVE 的 5 个元素中的第二大的元素. 下面会证明 FIND-MAX 算法中最多产生 $\lceil \log_5 n \rceil$ 个这样的第二大元素, 再重复调用一次 FIND-MAX 算法, 这次会调用 $\lceil \frac{\lceil \log_5 n \rceil - 1}{4} \rceil$ 次 SORT-FIVE 算法, 第二次 FIND-MAX 算法的返回值就是第二大元素.

伪代码

时间复杂度分析

由基本原理部分的阐述, 容易得出, 该算法总共调用了 $\lceil \frac{n-1}{4} \rceil + \lceil \frac{\lceil \log_5 n \rceil - 1}{4} \rceil$ 次 SORT-FIVE 算法.

算法正确性和最优性证明

正确性 首先注意到一个性质: 对于第二大元素 y , 一定与最大元素 x 在一次 SORT-FIVE 中比较过, 且仅在这一次比较中排名第二被淘汰, 即, 在该次与 x 的比较之前, y 参与的每一次 SORT-FIVE 的结果都是 y 最大. 该性质的证明可以通过反证法: 如果 y 在那次 SORT-FIVE 之前就已经被淘汰, 则不可能是第二大的元素. 如果 y 在那次比较中排名不是第二, 则要么与 x 最大矛盾, 要么与 y 第二大矛盾.

由上述性质, 所有可能的第二大元素都必须是在与 x 的 SORT-FIVE 中排名第二被淘汰. 即潜在的第二大的元素个数 $\leq x$ 参与的 SORT-FIVE 次数.

因此, 该算法接下来对这些潜在的第二大元素进行 FIND-MAX, 得到的最大元素就是全局的第二大元素.

最优性 同时, 这些被 x 淘汰的第二大元素也都可能是第二大元素. 因为假设任意一个元素 y 是第二大元素, 则必然在于 x 的比较之前一直优胜, 直到与 x 的 SORT-FIVE 中排名第二被淘汰. 因此潜在的第二大的元素个数 $\geq x$ 参与的 SORT-FIVE 次数. 综上, 潜在的第二大元素的个数即为 x 参与的 SORT-FIVE 次数.

下面证明对于任意算法, 最大元素参与的 SORT-FIVE 的次数的下界为 $\lceil \log_5 n \rceil$. 使用对手论证来进行证明.

证明. 假设每个元素对应一个选手, 找到最大元素的过程相当于是单败淘汰赛 (因为对于任意算法, 在调用一次 SORT-FIVE 后, 除了最大元素, 其余 4 个元素都不可能是最大值, 相当于被淘汰). 假设每个选手初始时都有一个金币, 当某个选手拿到所有金币时, 即取得胜利. 因此, 可以对每次调用 SORT-FIVE, 且最大元素 x 参与的过程进行分析. 记这 5 个元素为 x, a, b, c, d . 调用 SORT-FIVE 之前分别有 $w(x), w(a), w(b), w(c), w(d)$

个金币. 显然, 在进行比较后, x 的金币数变为 $w(x) + w(a) + w(b) + w(c) + w(d)$. 对于 x , 前后的金币的增长的倍数为 $xz\eta = \frac{w(x) + w(a) + w(b) + w(c) + w(d)}{w(x)}$. 因为对手总是期望 $w(x)$ 的增长速度尽可能慢, 从而需要进行更多的比较次数后 $w(x)$ 才能达到 n . 因此对手总是可以构造 $w(x)$ 大于等于其余四个元素的金币, 即 $\eta \leq 5$, 因此, 算法的设计者必须至少通过 $\log_\eta n \geq \log_5 n$ 次 x 参与的 SORT-FIVE 算法, 才能找到最大元素. 因此, 最大元素参与的 SORT-FIVE 的次数的下界为 $\lceil \log_5 n \rceil$.

因此, 产生的潜在的第二大元素的个数的下界 $= \lceil \log_5 n \rceil$. 由第 (1) 问已经证明, 对于规模为 n 的数组找到最大值需要调用 SORT-FIVE 的次数下界为 $\lceil \frac{n-1}{4} \rceil$, 因此, 对于 $\lceil \log_5 n \rceil$ 个元素, 调用 SORT-FIVE 的次数下界为 $\lceil \frac{\lceil \log_5 n \rceil - 1}{4} \rceil$.

因此, 总的调用 SORT-FIVE 的次数下界为 $\lceil \frac{n-1}{4} \rceil + \lceil \frac{\lceil \log_5 n \rceil - 1}{4} \rceil$.

因此, 给出的算法是最优的.

□

P 19.6 唯一的特殊元素

1)

基本原理

对于 n 为奇数的情况

将数组中前 $n-1$ 个元素均分为两个组, 然后两两比较, 一共比 $\frac{n-1}{2}$ 次, 如果比较的结果全部都是相同, 则说明最后一个没有比较的元素就是特殊元素; 否则, 必然存在一次比较得出的两个元素不同, 此时随意选择一个, 与其余任何一个元素比较, 如果不同, 则说明这个元素就是特殊元素, 否则另一个就是特殊元素. 比较的次数为 $\frac{n+1}{2}$ 次.

对于 n 为偶数的情况

将元素均分为两组, 然后两两比较, 比较前 $\frac{n}{2} - 1$ 次如果全部相等, 则说明特殊元素在最后 2 个元素里, 否则特殊元素在某一次结果不同的 2 个元素里. 将这 2 个元素中的任意一个与其余元素进行比较, 如果不同, 则特殊元素就是这个元素, 否则是另一个元素. 比较次数为 $\frac{n}{2}$ 次.

伪代码

Algorithm 40: UNQIUE(A[1..n])

Output: the unique element

```
1 int pos1 = 1, pos2 = ⌊ $\frac{n-1}{2}$ ⌋ + 1; /* 分别指示前半和后半元素 */  
2 while pos1 ≤ ⌊ $\frac{n-1}{2}$ ⌋ do  
3     if A[pos1] ≠ A[pos2] then break;  
4     pos1 := pos1 + 1;  
5     pos2 := pos2 + 1;  
6 if pos2 > n then  
    /* 说明 n 为奇数, 且最后一个元素就是特殊元素 */  
    return A[pos1];  
8 else  
    /* 否则, pos1 和 pos2 之一就指示特殊元素 */  
    9 int temp;  
10 if pos1 = 1 then temp := A[2];  
11 else  
12     temp := A[1];  
13 if A[pos1] = temp then  
14     return A[pos2];  
15 else  
16     return A[pos1];
```

2) 平均情况下的比较次数

假设 $n=2k$ 为偶数.

按照算法, 所有的元素实际上被分为 k 组, 每组两个元素, 对于前 $k-1$ 组, 如果特殊元素在第 i 组, 则需要比较 i 次才能发现不同, 之后还需要比较 1 次来确定具体是哪一个元素因此一共比较了 $(i+1)$ 次. 对于第 k 组, 是在比较了 $k-1$ 次后均相同, 反推出最后一组存在特殊元素, 还需要一次来确定具体是哪一个, 因此比较了 k 次.

$$\text{记 } X = \text{比较次数}, E(X) = \sum_{i=1}^{k-1} \frac{i+1}{k} + \frac{k}{k} = \frac{k^2 + 3k - 2}{2k} = \frac{n^2 + 6n - 8}{4n}.$$

注 : A smart guess

$$E(X) = \frac{n^2 + 6n - 8}{4n} \rightarrow \frac{n}{4}, \text{ 差不多是最坏情况的一半.}$$

3) 最坏情况的下界

证明. 记 $k = \lfloor \frac{n}{2} \rfloor$. 用 B 表示一个元素没有经过比较, N 表示比较结果是相同, 因此不可能是特殊元素, C 表示比较结果不同, 此时必然是这两个元素为 C, 其余元素为 N, 显然, 此时一定只需要 1 次就可以找出特殊元素, 用 Y 表示该元素为特殊元素, 算法终止.

初始情况下, n 个元素都是 B, 确定特殊元素, 则至少需要将 $(n-1)$ 个 B 排除, 即将这些 B 变为 Y/N/C(如果发现了 C, 则此时必然有 2 个 C, 且其余所有的元素都变为 N, 此时一定可以再比 1 次得到结果, 因此是确定的). 因此, 算法的对手总是希望 B 的减少速度尽可能慢.

下面考虑各种比较的情况:

- (B, B) 两个未知元素进行比较, 如果相等, 则得到两个 N, 否则得到两个 C(即将其余所有 B 均变为 N). 显然, 当剩余的 B 的数目大于 2 时, 对手会选择结果相等, 这样减少的 B 的数量更少.
- (B, N) 如果结果相等, 则减少 1 个 B, 否则说明找到了 Y, 算法终止 (即所有的 B 都变为 N), 显然, 对手总是希望结果相等.

因此, 算法的设计者首先尽可能将 (B,B) 比较, 因为这样消除 B 的效率更高, 而对手总是优先把结果设为相同 (否则将产生 C, 只需要再比较 1 次就找到答案). 对于 n 为偶数的情况, 在第 $k-1$ 次比较时, 对手使得结果相同或不同, 都是减少 2 个 B, 因此这一次结果无所谓. 对于 n 为奇数的情况, 在执行了 $k-1$ 次比较结果均相同时, 第 k 次如果结果相同, 则算法直接结束, 如果不同, 则还需要再比较一次, 因此对手会选择使结果不同. 至此, 当 n 为偶数时, 产生了 $n-2$ 个 N 和 2 个 C, 还需要比较 1 次, 一共比较了 $k = \lfloor \frac{n+1}{2} \rfloor$ 次. 当 n 为奇数时, 产生了 $n-2$ 个 N 和 2 个 C, 还需要比较 1 次, 一共比较了 $k + 1 = \lfloor \frac{n+1}{2} \rfloor$ 次.

综上所述, 该问题的最坏情况的下界为 $\lfloor \frac{n+1}{2} \rfloor$.

□

Chapt. 20 问题与归约

P 20.1

1) CLIQUE

优化问题

求图 G 中最大团的大小. 其中, 最大团被定义为图 G 中最大完全子图的阶.

判定问题

判定图 G 中是否存在阶为 k 的团.

优化问题与判定问题转化的证明

最大团问题的优化问题是多项式时间可解的, 当且仅当它的判定问题是多项式时间可解的.

证明.

- 必要性 (\Rightarrow). 如果优化问题是多项式时间可解的, 则意味着可以在 $O(\text{ploy}(n))$ 时间内求出图 G 的最大团的大小 k_0 . 因此可以立即回答任意判定问题: 对于大小为 k 的团的询问, 如果 $k \leq k_0$, 输出 True, 否则输出 False.
- 充分性 (\Leftarrow). 如果判定问题是多项式时间可解的, 则意味着可以在 $O(\text{poly}(n))$ 时间内回答是否存在大小为 k 的团的询问, 由于图 G 的最大团的阶不超过 $|G| = n$, 因此可以遍历 $k = n \dots 1$, 第一个回答 True 的 k 值记为优化问题的解, 该算法的时间复杂度为 $O(n \cdot \text{ploy}(n)) = O(\text{poly}(n))$, 因此优化问题也是多项式时间可解的.

□

NP 的证明

对于任意猜测的 CLIQUE 判定问题的一个解, 假设输入是 k 个顶点 (否则直接输出 False), 只需要 $O(k)$ 时间遍历每个顶点, 对于每个顶点, 以 $O(k)$ 时间判定该点与其余顶点是否关联. 如果均关联, 输出 True, 否则输出 False. 显然, 算法的时间复杂度为 $O(n^2)$. 因此该问题的解可以在多项式时间内验证, CLIQUE 问题是 NP 问题.

2) KNAPSACK

输入: n 个物品, 其大小分别为 s_1, s_2, \dots, s_n , 每个物品的价值为 c_1, c_2, \dots, c_n , 背包的总容量为 S .

优化问题

在不超过背包总容量 S 的前提下, 任意选择物品装入背包, 问装入物品的总价值最大为多少?

判定问题

在不超过背包总容量 S 的前提下, 任意选择物品装入背包, 问装入物品的总价值能否不低于 k ?

优化问题与判定问题转化的证明

01 背包的优化问题是多项式时间可解的, 当且仅当它的判定问题是多项式时间可解的.

- 证明.
- 必要性 (\Rightarrow). 如果优化问题是多项式时间可解的, 则意味着可以在 $O(\text{poly}(n))$ 时间内求出总容量不超过 S 的最大总价值 C_0 , 因此可以立即回答该问题的判定问题: 对于是否存在总价值不低于 k 的方案, 如果 $k \leq C_0$, 输入 True, 否则输出 False.
 - 充分性 (\Leftarrow). 如果判定问题是多项式时间可解的, 则意味着可以在 $O(\text{poly}(n))$ 时间内判定是否存在装入总价值不低于 k 的方案, 因此只要遍历所有 k 的取值 (不超过所有物品的总价值), 最后取输出 True 的最大 k 值即为优化问题的解. 因此, 优化问题是多项式时间可解的.

□

NP 的证明

考虑 KNAPSACK 的判定问题, 考虑随机生成的一个方案, 首先假设输入是若干选择的物品, 否则可以直接返回 False. 首先以 $O(n)$ 的时间统计所选物品的总容量 $S' = \sum s_i$, 如果 $S' > S$ 则直接输出 False, 否则继续以 $O(n)$ 的时间统计总价值 $K = \sum c_i$, 如果 $K \geq k$, 输出 True, 否则输入 False. 因此可以在 $O(n)$ 时间内验证某个方案是否为 True. 因此 KNAPSACK 问题是 NP 的.

3) INDEPENDENT-SET

输入: 无向图 G .

优化问题

输出: 图 G 中最大点独立集的大小.

判定问题

输出: 图 G 中是否存在大小为 k 的点独立集.

优化问题与判定问题转化的证明

独立集的优化问题是多项式时间可解的, 当且仅当它的判定问题是多项式时间可解的.

- 证明.
- 必要性 (\Rightarrow). 如果优化问题是多项式时间可解的, 则意味着可以在 $O(\text{poly}(n))$ 时间内求出图 G 中的最大点独立集的大小 S , 因此可以立即回答任意判定问题: 对于图 G , 询问是否存在大小为 k 的点独立集, 如果 $k \leq S$, 则输出 True(因为点独立集的子集也是点独立集), 否则, 输出 False(否则与最大点独立集大小为 S 矛盾).
 - 充分性 (\Leftarrow). 如果判定问题是多项式时间可解的, 则意味着可以在 $O(\text{poly}(n))$ 时间内判定图 G 中是否存在大小为 k 的点独立集, 因此只需要遍历 $k = n \dots 1$, 选取输出 True 的最大 k 作为优化问题的解. 显然, 优化问题同样是多项式时间可解的.

□

NP 的证明

考虑独立集问题的判定问题. 对于任意输入的 k 的顶点的方案, 首先假设输入恰为 k 个顶点, 否则直接输出 False. 然后以 $O(k^2) = O(n^2)$ 的时间代价判断每个顶点是否与其余顶点均不相邻, 若满足, 则输出 True, 否则输出 False. 显然, 算法的时间复杂度为 $O(n^2)$, 即独立集问题是 NP 的.

4) VERTEX-COVER

输入: 无向图 G

优化问题

输出: 图 G 的最小点覆盖集的大小.

判定问题

输出: 图 G 是否存在大小为 k 的点覆盖?

优化问题与判定问题转化的证明

点覆盖集的优化问题是多项式时间可解的, 当且仅当它的判定问题是多项式时间可解的.

- 证明.
- 必要性 (\Rightarrow). 如果优化问题是多项式时间可解的, 则意味着可以在 $O(\text{poly}(n))$ 时间内求出图 G 的点覆盖数 K , 因此可以立即回答任意的判定问题: 如果询问是否存在大小为 k 的点覆盖集, 如果 $k \geq K$, 则输出 True(向最小点覆盖集中添加点得到的仍然为点覆盖集), 否则输出 False(否则与最小点覆盖集的大小为 K 矛盾). 因此可以多项式时间求解判定问题.

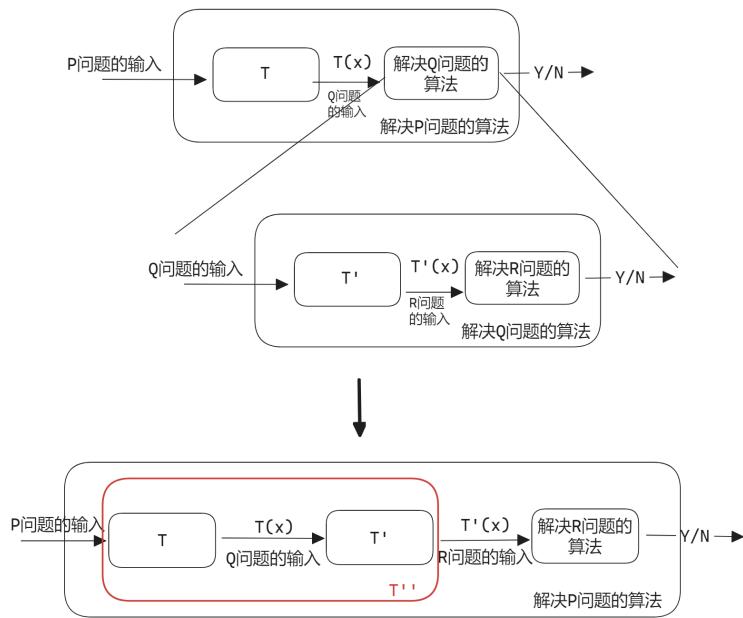
- 充分性 (\Leftarrow). 如果判定问题是多项式时间可解的, 则意味着可以在 $O(\text{poly}(n))$ 时间内回答是否存在大小为 k 的点覆盖集, 因此只要遍历 $k = 1 \dots n$, 最后取输出 True 的最小 k 最为优化问题的解即可. 显然, 优化问题是多项式时间可解的.

□

NP 的证明

考虑点覆盖集的判定问题. 假设输入为 k 个顶点的方案, 否则输出 False. 首先以 $O(n)$ 的时间遍历每个顶点, 每个顶点以 $O(n)$ 的时间遍历其关联的边, 将关联的边标记为 T , 最后遍历每条边, 如果所有边均被标记为 T , 则输出 True, 否则输出 False. 算法的时间复杂度为 $O(n^2)$, 因此点覆盖问题是 NP 的.

P 20.2 多项式时间规约关系是传递关系



如上图所示. 因为 $P \leq Q, Q \leq R$, 因此 $P = O(\text{poly}(Q)), Q = O(\text{poly}(R)), T(x) = O(\text{poly}(Q)) = O(\text{poly}(R)), T'(x) = O(\text{poly}(R))$, 因此 $T''(x) = O(\text{poly}(R))$, 因此可以以多项式时间代价将问题 P 的输入转换为问题 R 的输入, 从而将问题 P 归约到问题 R.

P 20.3

转换的代价为 $O(n^2)$, 解决 B 问题的时间复杂度为 $O(n^4)$, 因此总的时间代价为 $O(n^4)$ 即为 A 问题所需的时间.

P 20.4

- 排序 \leq 选择: 选择 n 次, 每次选择第 k 小.
- 选择 \leq 排序: 将待选的元素标记为 0, 其余标记为 1, 排序后第一个元素就是待选元素.

P 20.5

- 问题 1 \leq 问题 2: 当 $n = |S|$ 为奇数时, 即中位数即 S 的所有数中阶为 $\lceil \frac{n}{2} \rceil$ 的元素, 否则, 中位数即 $n/2, n/2 + 1$ 的元素的平均值.
- 问题 2 \leq 问题 1: 当 n 为奇数时, 比较 k 与 $\lceil \frac{n}{2} \rceil$ 的大小关系: 如果相等, 则第一轮就找到, 如果大于, 则删除中位数, 并使 $k := k-1$, 否则小于, k 不变. 此后 n 为偶数. 当 n 为偶数时, 比较 k 与 $n/2, n/2 + 1$, 如果等于其中一个, 则当前一轮找到, 否则删除这两个数, $n := n-2$. 如果 $k < n/2$ 则 k 不变, 否则 $k := k-2$. 注意到每一轮 n 至少减小 1, 因此最多执行 $O(n)$ 次问题 1 的算法即可解决问题 2.

Chapt. 21 NP 完全性理论初步

P 21.1

1)

如果任意一个 NPC 问题可以在多项式时间内解决, 则所有 NP 问题均可以在多项式时间内解决. 即 $P = NP$.

证明. 不妨记 $P^* \in NPC$, 则 $\forall Q \in NP, Q \leq P^*$, 即 NP 中任意问题的难度不超过 P^* , 又因为 P^* 是多项式时间可解的, 因此 $\forall Q \in NP, Q$ 是多项式时间可解的. 即 $NP = P$. \square

2)

如果任意一个 NPC 问题不存在多项式时间解, 则所有的 NPC 问题均不可能再多项式时间内解决.

证明. 不妨记该 NPC 问题为 P^* , 则 P^* 不存在多项式时间解. 反证法, 假设存在另一个 NPC 问题 Q 在多项式时间内可解, 因为 $P^* \in NP, Q \in NP-hard$, 因此 $P^* \leq Q$, 于是 P^* 可以多项式时间规约到 Q , 因此 P^* 同样存在多项式时间解, 与前提矛盾. 故任意 NPC 问题不存在多项式时间解. \square

P 21.3 伪最大团问题

1)

基本原理

首先枚举所有可能的 k 个顶点的组合, 一共有 $\binom{n}{k} = O(n^k)$ 个. 对于每个 k 个顶点的方案, 可以在 $O(k^2) = O(n^2)$ 的时间内判断这 k 个顶点的导出子图是否为完全图. 如果是, 则输出 True, 否则如果均不是, 则输出 False.

时间复杂度分析

时间复杂度为 $O(n^k \times n^2 = n^{k+2})$, 当 k 是常数时, 可以认为该算法是多项式时间的.

2)

不是.

理由 1: 如果是的话, P vs. NP 问题就不是至今未解之谜了.

理由 2: 这里的 k 不能认为是常数, 因为 k 的任意性, k 能够取到 $n/2 \dots n$, 此时的时间复杂度不是关于 n 的多项式.

P 21.4

1)

证明. 可以设计如下算法:

遍历每个子句, 由于每个子句由若干文字的交组成, 因此该子句的成真指派 iff. 每个文字均为真. 因此只需要维护一个表, 记录每个元素的当前指派是 NULL(未指派, 任意)/T/F, 如果遍历到某个文字时, 发现该文字的反已经出现过, 则此时该子句必然不能被满足, 结束该子句的遍历; 否则, 说明该子句存在成真指派, 由于 DNF 范式由若干子句的并组成, 因此如果存在一个子句为真, 则整个 DNF 存在成真指派. 因此终止算法并输出 True, 否则, 若所有子句均不存在成真指派, 则输出 False.

显然, 该算法是 $O(n)$ 的, 即 DNF-SAT 问题是 P 问题. □

2)

问题出在, 不存在一个多项式时间的算法, 将任意逻辑表达式的 DNF 范式转换为 CNF 范式, 因此将 CNF-SAT 问题规约到 DNF-SAT 问题是无意义的. 不能得出 $CNF - SAT \leq DNF - SAT$ 的结论.

P 21.5

1)

设已知子集和问题是 NPC 问题, 证明背包问题是 NPC 问题.

证明. 不妨记子集和问题为 Q 问题, 背包问题是 R 问题. 因为 Q 是 NPC, 因此 $\forall P \in P, P \leq Q$, 因此只需证明 $Q \leq R \wedge R \in NP$, 就有 $\forall P \in P, P \leq Q \leq R$ 且 $R \in NP$, 于是 R 是 NPC 的.

1. 证明 $Q \leq R$. 只需将每个元素对应成一个物品, 该物品的价值和体积均为该元素的值, 然后将问题的参数 C 和 V 均设为 S, 于是问题转换为是否存在一个方案, 使得所选物品的总体积不高于 S, 且所选物品的总价值不低于 S, 在当前的情形下, 该问题输出 True 当且仅当存在集合 S 存在一个子集使得子集的和恰为 S. 易见, 规约的代价是多项式时间的, 从而能够将问题 Q 规约到问题 R.

2. 证明 R 是 NP 的. 这个证明同 P20.1, 此处略.

综上, $R \in NP-hard \wedge R \in NP$, 因此 R 是 NPC 的.

□

2)

Algorithm 41: 01-KNAPSACK

Input : 每个物品的体积 $w[1..n]$, 每个物品的价值 $v[1..n]$, V

Output: the highest value with limit volume of V

```
1 初始化: int f[n][n] 数组, f[i][j] 表示前 i 个物品在体积不超过 j 的前提下的最
      大价值, 初始均为 0;
2 for i := n to 1 do
3   for j := V to 0 do
4     if j ≥ w[i] then
5       f[i][j] := max(f[i-1][j], f[i-1][j - w[i]] + v[i]);
6     else
7       f[i][j] := f[i-1][j];
8 return f[n][V];
```

3)

算法的状态数为 $O(nV)$, 每个状态的转移的代价为 $O(1)$, 因此总的时间复杂度为 $O(nV)$. 空间复杂度为 $O(nV)$, 但是考虑到 $f[i][.]$ 仅依赖 $f[i-1][.]$, 因此可以采用滚动数组压缩空间至 $O(V)$.

不能, 因此时间复杂度中的 V 不是一个关于 n 的多项式. 当我们说一个算法是多项式时间可解的, 是指该算法的运行时间是关于其输入大小的多项式. 具体来说, 输入的每个物品的容量 $S_i, 0 < S_i \leq S$ (否则可以直接舍弃该物品). 一共输入 n 个物品, 实际的输入大小为 $O(n(\log V + \log S))$ (因为每个数字以二进制 bit 串存储). 记 $v = \log V, s = \log S$, 则输入的大小为 $O(n(v + s))$. 但是算法的运行时间为 $O(nS) = O(n \cdot 2^s)$, 因此该算法是指数级别时间复杂度的.

注 : 更详细的说明

关于为什么该算法不是多项式时间的, 参见这篇 recitation: Polynomial Time vs Pseudo-Polynomial Time, 以及这个回答.

P 21.6 支配集和集合覆盖问题

已知 DOMINATION-SET 是 NPC 问题, 证明 SET-COVER 是 NPC 问题.

证明. 不妨记支配集问题为 Q 问题, 集合覆盖问题为 R 问题. 即证明:

1. $Q \leq R$. 对于 Q 问题的输入图 $G = \langle V, E \rangle$, 令 $U = V$, 令 S 中每个顶点为 $S_i = \{v_i\} \cup N(v_i)$. 于是图 G 的任意一个点支配集等价于全集 U 的若干子集的一个集合覆盖. 易见转换的代价是多项式时间的, 从而将 Q 多项式时间规约到 R.
2. R 是 NP 的. 考虑 R 的判定问题, 给定任意一个输入, 假设输入是 k 个选择的若干子集, 否则直接输出 False. 遍历每个子集, 将其中的元素标记为 T, 最后遍历每个元素, 如果所有元素均为 T, 则输出 True, 否则输出 False. 这个验证算法的时间复杂度是 $O(n^2)$. 因此 R 是 NP 的.

综上, $R \in NP-hard \wedge R \in NP$, 因此 R 是 NPC 的. □