

ac 模板

1 快速排序算法模板 — 模板题 AcWing 785. 快速排序

```
2 void quick_sort(int q[], int l, int r)
3 {
4     if (l >= r) return;
5
6     int i = l - 1, j = r + 1, x = q[l + r >> 1];
7     while (i < j)
8     {
9         do i++; while (q[i] < x);
10        do j--; while (q[j] > x);
11        if (i < j) swap(q[i], q[j]);
12    }
13    quick_sort(q, l, j), quick_sort(q, j + 1, r);
14 }
```

15 归并排序算法模板 — 模板题 AcWing 787. 归并排序

```
16 void merge_sort(int q[], int l, int r)
17 {
18     if (l >= r) return;
19
20     int mid = l + r >> 1;
21     merge_sort(q, l, mid);
22     merge_sort(q, mid + 1, r);
23
24     int k = 0, i = l, j = mid + 1;
25     while (i <= mid && j <= r)
26         if (q[i] <= q[j]) tmp[k++] = q[i++];
27         else tmp[k++] = q[j++];
28
29     while (i <= mid) tmp[k++] = q[i++];
30     while (j <= r) tmp[k++] = q[j++];
31
32     for (i = l, j = 0; i <= r; i++, j++) q[i] = tmp[j];
33 }
```

34 整数二分算法模板 — 模板题 AcWing 789. 数的范围

```
35 bool check(int x) { /* ... */ } // 检查x是否满足某种性质
36
37 // 区间[l, r]被划分成[l, mid]和[mid + 1, r]时使用：
```

```

38 int bsearch_1(int l, int r)
39 {
40     while (l < r)
41     {
42         int mid = l + r >> 1;
43         if (check(mid)) r = mid;    // check()判断mid是否满足性质
44         else l = mid + 1;
45     }
46     return l;
47 }
48 // 区间[l, r]被划分成[l, mid - 1]和[mid, r]时使用：
49 int bsearch_2(int l, int r)
50 {
51     while (l < r)
52     {
53         int mid = l + r + 1 >> 1;
54         if (check(mid)) l = mid;
55         else r = mid - 1;
56     }
57     return l;
58 }
59 浮点数二分算法模板 — 模板题 AcWing 790. 数的三次方根
60 bool check(double x) { /* ... */ } // 检查x是否满足某种性质
61
62 double bsearch_3(double l, double r)
63 {
64     const double eps = 1e-6;    // eps 表示精度，取决于题目对精度的要求
65     while (r - l > eps)
66     {
67         double mid = (l + r) / 2;
68         if (check(mid)) r = mid;
69         else l = mid;
70     }
71     return l;
72 }
73 高精度加法 — 模板题 AcWing 791. 高精度加法
74 // C = A + B, A >= 0, B >= 0
75 vector<int> add(vector<int> &A, vector<int> &B)
76 {
77     if (A.size() < B.size()) return add(B, A);

```

```

78
79     vector<int> C;
80     int t = 0;
81     for (int i = 0; i < A.size(); i++)
82     {
83         t += A[i];
84         if (i < B.size()) t += B[i];
85         C.push_back(t % 10);
86         t /= 10;
87     }
88
89     if (t) C.push_back(t);
90     return C;
91 }
92 高精度减法 — 模板题 AcWing 792. 高精度减法
93 // C = A - B, 满足A >= B, A >= 0, B >= 0
94 vector<int> sub(vector<int> &A, vector<int> &B)
95 {
96     vector<int> C;
97     for (int i = 0, t = 0; i < A.size(); i++)
98     {
99         t = A[i] - t;
100         if (i < B.size()) t -= B[i];
101         C.push_back((t + 10) % 10);
102         if (t < 0) t = 1;
103         else t = 0;
104     }
105
106     while (C.size() > 1 && C.back() == 0) C.pop_back();
107     return C;
108 }
109 高精度乘低精度 — 模板题 AcWing 793. 高精度乘法
110 // C = A * b, A >= 0, b >= 0
111 vector<int> mul(vector<int> &A, int b)
112 {
113     vector<int> C;
114
115     int t = 0;
116     for (int i = 0; i < A.size() || t; i++)
117     {

```

```

118         if (i < A.size()) t += A[i] * b;
119         C.push_back(t % 10);
120         t /= 10;
121     }
122
123     while (C.size() > 1 && C.back() == 0) C.pop_back();
124
125     return C;
126 }

```

127 高精度除以低精度 — 模板题 AcWing 794. 高精度除法

```

128 // A / b = C ... r, A >= 0, b > 0
129 vector<int> div(vector<int> &A, int b, int& r)
130 {
131     vector<int> C;
132     r = 0;
133     for (int i = A.size() - 1; i >= 0; i--)
134     {
135         r = r * 10 + A[i];
136         C.push_back(r / b);
137         r %= b;
138     }
139     reverse(C.begin(), C.end());
140     while (C.size() > 1 && C.back() == 0) C.pop_back();
141     return C;
142 }

```

143 一维前缀和 — 模板题 AcWing 795. 前缀和

```

144 S[i] = a[1] + a[2] + ... a[i]
145 a[1] + ... + a[r] = S[r] - S[1 - 1]

```

146 二维前缀和 — 模板题 AcWing 796. 子矩阵的和

147 $S[i, j]$ = 第*i*行*j*列格子左上部分所有元素的和

148 以 $(x1, y1)$ 为左上角, $(x2, y2)$ 为右下角的子矩阵的和为:

```

149  $S[x2, y2] - S[x1 - 1, y2] - S[x2, y1 - 1] + S[x1 - 1, y1 - 1]$ 

```

150 一维差分 — 模板题 AcWing 797. 差分

151 给区间 $[l, r]$ 中的每个数加上 c : $B[l] += c, B[r + 1] -= c$

152 二维差分 — 模板题 AcWing 798. 差分矩阵

153 给以 $(x1, y1)$ 为左上角, $(x2, y2)$ 为右下角的子矩阵中的所有元素加上 c :

```

154  $S[x1, y1] += c, S[x2 + 1, y1] -= c, S[x1, y2 + 1] -= c, S[x2 + 1, y2$ 
     $+ 1] += c$ 

```

155 位运算 — 模板题 AcWing 801. 二进制中1的个数

156 求*n*的第*k*位数字: $n \gg k \& 1$

```

157 返回n的最后一位1: lowbit(n) = n & -n
158 双指针算法 — 模板题 AcWing 799. 最长连续不重复子序列, AcWing 800. 数组元素的目标和
159 for (int i = 0, j = 0; i < n; i++)
160 {
161     while (j < i && check(i, j)) j++;
162
163     // 具体问题的逻辑
164 }
165 常见问题分类:
166 (1) 对于一个序列, 用两个指针维护一段区间
167 (2) 对于两个序列, 维护某种次序, 比如归并排序中合并两个有序序列的操作
168 离散化 — 模板题 AcWing 802. 区间和
169 vector<int> alls; // 存储所有待离散化的值
170 sort(alls.begin(), alls.end()); // 将所有值排序
171 alls.erase(unique(alls.begin(), alls.end()), alls.end()); // 去掉重复元素
172
173 // 二分求出x对应的离散化的值
174 int find(int x) // 找到第一个大于等于x的位置
175 {
176     int l = 0, r = alls.size() - 1;
177     while (l < r)
178     {
179         int mid = l + r >> 1;
180         if (alls[mid] >= x) r = mid;
181         else l = mid + 1;
182     }
183     return r + 1; // 映射到1, 2, ...n
184 }
185 区间合并 — 模板题 AcWing 803. 区间合并
186 // 将所有存在交集的区间合并
187 void merge(vector<PII> &segs)
188 {
189     vector<PII> res;
190
191     sort(segs.begin(), segs.end());
192
193     int st = -2e9, ed = -2e9;
194     for (auto seg : segs)

```

```

195         if (ed < seg.first)
196         {
197             if (st != -2e9) res.push_back({ st, ed });
198             st = seg.first, ed = seg.second;
199         }
200         else ed = max(ed, seg.second);
201
202         if (st != -2e9) res.push_back({ st, ed });
203
204         segs = res;
205     }
206
207 单链表 — 模板题 AcWing 826. 单链表
208 // head存储链表头, e[]存储节点的值, ne[]存储节点的next指针, idx表示当前用到了哪个
    节点
209 int head, e[N], ne[N], idx;
210
211 // 初始化
212 void init()
213 {
214     head = -1;
215     idx = 0;
216 }
217
218 // 在链表头插入一个数a
219 void insert(int a)
220 {
221     e[idx] = a, ne[idx] = head, head = idx++;
222 }
223
224 // 将头结点删除, 需要保证头结点存在
225 void remove()
226 {
227     head = ne[head];
228 }
229 双链表 — 模板题 AcWing 827. 双链表
230 // e[]表示节点的值, l[]表示节点的左指针, r[]表示节点的右指针, idx表示当前用到了哪个
    节点
231 int e[N], l[N], r[N], idx;
232

```

```
233 // 初始化
234 void init()
235 {
236     //0是左端点, 1是右端点
237     r[0] = 1, l[1] = 0;
238     idx = 2;
239 }
240
241 // 在节点a的右边插入一个数x
242 void insert(int a, int x)
243 {
244     e[idx] = x;
245     l[idx] = a, r[idx] = r[a];
246     l[r[a]] = idx, r[a] = idx++;
247 }
248
249 // 删除节点a
250 void remove(int a)
251 {
252     l[r[a]] = l[a];
253     r[l[a]] = r[a];
254 }
255 栈 — 模板题 AcWing 828. 模拟栈
256 // tt表示栈顶
257 int stk[N], tt = 0;
258
259 // 向栈顶插入一个数
260 stk[++tt] = x;
261
262 // 从栈顶弹出一个数
263 tt--;
264
265 // 栈顶的值
266 stk[tt];
267
268 // 判断栈是否为空, 如果 tt > 0, 则表示不为空
269 if (tt > 0)
270 {
271
272 }
```

273 队列 — 模板题 AcWing 829. 模拟队列

274 1. 普通队列:

275 // hh 表示队头, tt表示队尾

276 `int q[N], hh = 0, tt = -1;`

277

278 // 向队尾插入一个数

279 `q[++tt] = x;`

280

281 // 从队头弹出一个数

282 `hh++;`

283

284 // 队头的值

285 `q[hh];`

286

287 // 判断队列是否为空, 如果 `hh <= tt`, 则表示不为空

288 `if (hh <= tt)`

289 {

290

291 }

292 2. 循环队列

293 // hh 表示队头, tt表示队尾的后一个位置

294 `int q[N], hh = 0, tt = 0;`

295

296 // 向队尾插入一个数

297 `q[tt++] = x;`

298 `if (tt == N) tt = 0;`

299

300 // 从队头弹出一个数

301 `hh++;`

302 `if (hh == N) hh = 0;`

303

304 // 队头的值

305 `q[hh];`

306

307 // 判断队列是否为空, 如果`hh != tt`, 则表示不为空

308 `if (hh != tt)`

309 {

310

311 }

312 单调栈 — 模板题 AcWing 830. 单调栈

313 常见模型：找出每个数左边离它最近的比它大 / 小的数

```
314 int tt = 0;
315 for (int i = 1; i <= n; i++)
316 {
317     while (tt && check(stk[tt], i)) tt--;
318     stk[++tt] = i;
319 }
```

320 单调队列 — 模板题 AcWing 154. 滑动窗口

321 常见模型：找出滑动窗口中的最大值 / 最小值

```
322 int hh = 0, tt = -1;
323 for (int i = 0; i < n; i++)
324 {
325     while (hh <= tt && check_out(q[hh])) hh++; // 判断队头是否滑出窗口
326     while (hh <= tt && check(q[tt], i)) tt--;
327     q[++tt] = i;
328 }
```

329 KMP — 模板题 AcWing 831. KMP字符串

330 // s[]是长文本, p[]是模式串, n是s的长度, m是p的长度

331 求模式串的Next数组：

```
332 for (int i = 2, j = 0; i <= m; i++)
333 {
334     while (j && p[i] != p[j + 1]) j = ne[j];
335     if (p[i] == p[j + 1]) j++;
336     ne[i] = j;
337 }
338
```

339 // 匹配

```
340 for (int i = 1, j = 0; i <= n; i++)
341 {
342     while (j && s[i] != p[j + 1]) j = ne[j];
343     if (s[i] == p[j + 1]) j++;
344     if (j == m)
345     {
346         j = ne[j];
347         // 匹配成功后的逻辑
348     }
349 }
```

350 Trie树 — 模板题 AcWing 835. Trie字符串统计

```
351 int son[N][26], cnt[N], idx;
352 // 0号点既是根节点, 又是空节点
```

```

353 // son[][]存储树中每个节点的子节点
354 // cnt[]存储以每个节点结尾的单词数量
355
356 // 插入一个字符串
357 void insert(char* str)
358 {
359     int p = 0;
360     for (int i = 0; str[i]; i++)
361     {
362         int u = str[i] - 'a';
363         if (!son[p][u]) son[p][u] = ++idx;
364         p = son[p][u];
365     }
366     cnt[p]++;
367 }
368
369 // 查询字符串出现的次数
370 int query(char* str)
371 {
372     int p = 0;
373     for (int i = 0; str[i]; i++)
374     {
375         int u = str[i] - 'a';
376         if (!son[p][u]) return 0;
377         p = son[p][u];
378     }
379     return cnt[p];
380 }
381 并查集 — 模板题 AcWing 836. 合并集合, AcWing 837. 连通块中点的数量
382 (1)朴素并查集：
383
384 int p[N]; //存储每个点的祖宗节点
385
386 // 返回x的祖宗节点
387 int find(int x)
388 {
389     if (p[x] != x) p[x] = find(p[x]);
390     return p[x];
391 }
392

```

```

393 // 初始化，假定节点编号是1~n
394 for (int i = 1; i <= n; i++) p[i] = i;
395
396 // 合并a和b所在的两个集合：
397 p[find(a)] = find(b);
398
399
400 (2)维护size的并查集：
401
402 int p[N], size[N];
403 //p[]存储每个点的祖宗节点， size[]只有祖宗节点的有意义，表示祖宗节点所在集合中的点的
    数量
404
405 // 返回x的祖宗节点
406 int find(int x)
407 {
408     if (p[x] != x) p[x] = find(p[x]);
409     return p[x];
410 }
411
412 // 初始化，假定节点编号是1~n
413 for (int i = 1; i <= n; i++)
414 {
415     p[i] = i;
416     size[i] = 1;
417 }
418
419 // 合并a和b所在的两个集合：
420 size[find(b)] += size[find(a)];
421 p[find(a)] = find(b);
422
423
424 (3)维护到祖宗节点距离的并查集：
425
426 int p[N], d[N];
427 //p[]存储每个点的祖宗节点， d[x]存储x到p[x]的距离
428
429 // 返回x的祖宗节点
430 int find(int x)
431 {

```

```

432     if (p[x] != x)
433     {
434         int u = find(p[x]);
435         d[x] += d[p[x]];
436         p[x] = u;
437     }
438     return p[x];
439 }
440
441 // 初始化，假定节点编号是1~n
442 for (int i = 1; i <= n; i++)
443 {
444     p[i] = i;
445     d[i] = 0;
446 }
447
448 // 合并a和b所在的两个集合：
449 p[find(a)] = find(b);
450 d[find(a)] = distance; // 根据具体问题，初始化find(a)的偏移量
451 堆 — 模板题 AcWing 838. 堆排序, AcWing 839. 模拟堆
452 // h[N]存储堆中的值, h[1]是堆顶, x的左儿子是2x, 右儿子是2x + 1
453 // ph[k]存储第k个插入的点在堆中的位置
454 // hp[k]存储堆中下标是k的点是第几个插入的
455 int h[N], ph[N], hp[N], size;
456
457 // 交换两个点，及其映射关系
458 void heap_swap(int a, int b)
459 {
460     swap(ph[hp[a]], ph[hp[b]]);
461     swap(h[a], h[b]);
462     swap(h[a], h[b]);
463 }
464
465 void down(int u)
466 {
467     int t = u;
468     if (u * 2 <= size && h[u * 2] < h[t]) t = u * 2;
469     if (u * 2 + 1 <= size && h[u * 2 + 1] < h[t]) t = u * 2 + 1;
470     if (u != t)
471     {

```

```

472         heap_swap(u, t);
473         down(t);
474     }
475 }
476
477 void up(int u)
478 {
479     while (u / 2 && h[u] < h[u / 2])
480     {
481         heap_swap(u, u / 2);
482         u >>= 1;
483     }
484 }
485
486 // O(n)建堆
487 for (int i = n / 2; i; i--) down(i);
488 一般哈希 — 模板题 AcWing 840. 模拟散列表
489 (1) 拉链法
490 int h[N], e[N], ne[N], idx;
491
492 // 向哈希表中插入一个数
493 void insert(int x)
494 {
495     int k = (x % N + N) % N;
496     e[idx] = x;
497     ne[idx] = h[k];
498     h[k] = idx++;
499 }
500
501 // 在哈希表中查询某个数是否存在
502 bool find(int x)
503 {
504     int k = (x % N + N) % N;
505     for (int i = h[k]; i != -1; i = ne[i])
506         if (e[i] == x)
507             return true;
508
509     return false;
510 }
511

```

512 (2) 开放寻址法

513 `int h[N];`

514

515 // 如果x在哈希表中，返回x的下标；如果x不在哈希表中，返回x应该插入的位置

516 `int find(int x)`

517 {

518 `int t = (x % N + N) % N;`

519 `while (h[t] != null && h[t] != x)`

520 {

521 `t++;`

522 `if (t == N) t = 0;`

523 }

524 `return t;`

525 }

526 字符串哈希 — 模板题 AcWing 841. 字符串哈希

527 核心思想：将字符串看成P进制数，P的经验值是131或13331，取这两个值的冲突概率低

528 小技巧：取模的数用 2^{64} ，这样直接用unsigned long long存储，溢出的结果就是取模的结果

529

530 `typedef unsigned long long ULL;`

531 `ULL h[N], p[N];` // h[k]存储字符串前k个字母的哈希值，p[k]存储 $P^k \bmod 2^{64}$

532

533 // 初始化

534 `p[0] = 1;`

535 `for (int i = 1; i <= n; i++)`

536 {

537 `h[i] = h[i - 1] * P + str[i];`

538 `p[i] = p[i - 1] * P;`

539 }

540

541 // 计算子串 str[l ~ r] 的哈希值

542 `ULL get(int l, int r)`

543 {

544 `return h[r] - h[l - 1] * p[r - l + 1];`

545 }

546 C++ STL简介

547 vector, 变长数组，倍增的思想

548 `size()` 返回元素个数

549 `empty()` 返回是否为空

550 `clear()` 清空

```
551 front() / back()
552 push_back() / pop_back()
553 begin() / end()
554 []
555 支持比较运算，按字典序
556
557 pair<int, int>
558 first, 第一个元素
559 second, 第二个元素
560 支持比较运算，以first为第一关键字，以second为第二关键字（字典序）
561
562 string, 字符串
563 size() / length() 返回字符串长度
564 empty()
565 clear()
566 substr(起始下标, (子串长度)) 返回子串
567 c_str() 返回字符串所在字符数组的起始地址
568
569 queue, 队列
570 size()
571 empty()
572 push() 向队尾插入一个元素
573 front() 返回队头元素
574 back() 返回队尾元素
575 pop() 弹出队头元素
576
577 priority_queue, 优先队列，默认是大根堆
578 size()
579 empty()
580 push() 插入一个元素
581 top() 返回堆顶元素
582 pop() 弹出堆顶元素
583 定义成小根堆的方式: priority_queue<int, vector<int>, greater<int>> q;
584
585 stack, 栈
586 size()
587 empty()
588 push() 向栈顶插入一个元素
589 top() 返回栈顶元素
590 pop() 弹出栈顶元素
```

```
591
592 deque, 双端队列
593 size()
594 empty()
595 clear()
596 front() / back()
597 push_back() / pop_back()
598 push_front() / pop_front()
599 begin() / end()
600 []
601
602 set, map, multiset, multimap, 基于平衡二叉树（红黑树），动态维护有序序列
603 size()
604 empty()
605 clear()
606 begin() / end()
607 ++, --返回前驱和后继，时间复杂度  $O(\log n)$ 
608
609 set / multiset
610 insert() 插入一个数
611 find() 查找一个数
612 count() 返回某一个数的个数
613 erase()
614 (1) 输入是一个数x，删除所有x  $O(k + \log n)$ 
615 (2) 输入一个迭代器，删除这个迭代器
616 lower_bound() / upper_bound()
617 lower_bound(x) 返回大于等于x的最小的数的迭代器
618 upper_bound(x) 返回大于x的最小的数的迭代器
619 map / multimap
620 insert() 插入的数是一个pair
621 erase() 输入的参数是pair或者迭代器
622 find()
623 [] 注意multimap不支持此操作。 时间复杂度是  $O(\log n)$ 
624 lower_bound() / upper_bound()
625
626 unordered_set, unordered_map, unordered_multiset, unordered_multimap, 哈希表
627 和上面类似，增删改查的时间复杂度是  $O(1)$ 
628 不支持 lower_bound() / upper_bound(), 迭代器的++, --
629
```

```

630 bitset, 压位
631 bitset<10000> s;
632 ~, &, |, ^
633 >>, <<
634 ==, !=
635 []
636
637 count() 返回有多少个1
638
639 any() 判断是否至少有一个1
640 none() 判断是否全为0
641
642 set() 把所有位置成1
643 set(k, v) 将第k位变成v
644 reset() 把所有位变成0
645 flip() 等价于~
646 flip(k) 把第k位取反
647
648 树与图的存储
649 树是一种特殊的图，与图的存储方式相同。
650 对于无向图中的边ab，存储两条有向边a->b, b->a。
651 因此我们可以只考虑有向图的存储。
652
653 (1) 邻接矩阵：g[a][b] 存储边a->b
654
655 (2) 邻接表：
656
657 // 对于每个点k，开一个单链表，存储k所有可以走到的点。h[k]存储这个单链表的头结点
658 int h[N], e[N], ne[N], idx;
659
660 // 添加一条边a->b
661 void add(int a, int b)
662 {
663     e[idx] = b, ne[idx] = h[a], h[a] = idx++;
664 }
665
666 // 初始化
667 idx = 0;
668 memset(h, -1, sizeof h);
669 树与图的遍历

```

670 时间复杂度 $O(n + m)$ $O(n + m)$, nn 表示点数, mm 表示边数

671 (1) 深度优先遍历 — 模板题 AcWing 846. 树的重心

672

```
673 int dfs(int u)
```

```
674 {
```

```
675     st[u] = true; // st[u] 表示点u已经被遍历过
```

```
676
```

```
677     for (int i = h[u]; i != -1; i = ne[i])
```

```
678     {
```

```
679         int j = e[i];
```

```
680         if (!st[j]) dfs(j);
```

```
681     }
```

```
682 }
```

683 (2) 宽度优先遍历 — 模板题 AcWing 847. 图中点的层次

684

```
685 queue<int> q;
```

```
686 st[1] = true; // 表示1号点已经被遍历过
```

```
687 q.push(1);
```

```
688
```

```
689 while (q.size())
```

```
690 {
```

```
691     int t = q.front();
```

```
692     q.pop();
```

```
693
```

```
694     for (int i = h[t]; i != -1; i = ne[i])
```

```
695     {
```

```
696         int j = e[i];
```

```
697         if (!st[j])
```

```
698         {
```

```
699             st[j] = true; // 表示点j已经被遍历过
```

```
700             q.push(j);
```

```
701         }
```

```
702     }
```

```
703 }
```

704 拓扑排序 — 模板题 AcWing 848. 有向图的拓扑序列

705 时间复杂度 $O(n + m)$ $O(n + m)$, nn 表示点数, mm 表示边数

```
706 bool topsort()
```

```
707 {
```

```
708     int hh = 0, tt = -1;
```

```
709
```

```

710 // d[i] 存储点i的入度
711 for (int i = 1; i <= n; i++)
712     if (!d[i])
713         q[++tt] = i;
714
715 while (hh <= tt)
716 {
717     int t = q[hh++];
718
719     for (int i = h[t]; i != -1; i = ne[i])
720     {
721         int j = e[i];
722         if (--d[j] == 0)
723             q[++tt] = j;
724     }
725 }
726
727 // 如果所有点都入队了,说明存在拓扑序列;否则不存在拓扑序列。
728 return tt == n - 1;
729 }
730 朴素dijkstra算法 — 模板题 AcWing 849. Dijkstra求最短路 I
731 时间复杂度是  $O(n^2 + m)$   $O(n^2 + m)$ , nn 表示点数, mm 表示边数
732 int g[N][N]; // 存储每条边
733 int dist[N]; // 存储1号点到每个点的最短距离
734 bool st[N]; // 存储每个点的最短路是否已经确定
735
736 // 求1号点到n号点的最短路,如果不存在则返回-1
737 int dijkstra()
738 {
739     memset(dist, 0x3f, sizeof dist);
740     dist[1] = 0;
741
742     for (int i = 0; i < n - 1; i++)
743     {
744         int t = -1; // 在还未确定最短路的点中,寻找距离最小的点
745         for (int j = 1; j <= n; j++)
746             if (!st[j] && (t == -1 || dist[t] > dist[j]))
747                 t = j;
748
749         // 用t更新其他点的距离

```

```

750     for (int j = 1; j <= n; j++)
751         dist[j] = min(dist[j], dist[t] + g[t][j]);
752
753     st[t] = true;
754 }
755
756 if (dist[n] == 0x3f3f3f3f) return -1;
757 return dist[n];
758 }
759 堆优化版dijkstra — 模板题 AcWing 850. Dijkstra求最短路 II
760 时间复杂度  $O(m\log n)$   $O(m\log n)$ , nn 表示点数, mm 表示边数
761 typedef pair<int, int> PII;
762
763 int n;          // 点的数量
764 int h[N], w[N], e[N], ne[N], idx;          // 邻接表存储所有边
765 int dist[N];    // 存储所有点到1号点的距离
766 bool st[N];     // 存储每个点的最短距离是否已确定
767
768 // 求1号点到n号点的最短距离, 如果不存在, 则返回-1
769 int dijkstra()
770 {
771     memset(dist, 0x3f, sizeof dist);
772     dist[1] = 0;
773     priority_queue<PII, vector<PII>, greater<PII>> heap;
774     heap.push({ 0, 1 });          // first存储距离, second存储节点编号
775
776     while (heap.size())
777     {
778         auto t = heap.top();
779         heap.pop();
780
781         int ver = t.second, distance = t.first;
782
783         if (st[ver]) continue;
784         st[ver] = true;
785
786         for (int i = h[ver]; i != -1; i = ne[i])
787         {
788             int j = e[i];
789             if (dist[j] > distance + w[i])

```

```

790         {
791             dist[j] = distance + w[i];
792             heap.push({ dist[j], j });
793         }
794     }
795 }
796
797 if (dist[n] == 0x3f3f3f3f) return -1;
798 return dist[n];
799 }

```

800 Bellman - Ford算法 — 模板题 AcWing 853. 有边数限制的最短路

801 时间复杂度 $O(nm)$ $O(nm)$, nn 表示点数, mm 表示边数

802 注意在模板题中需要对下面的模板稍作修改, 加上备份数组, 详情见模板题。

```

803
804 int n, m;          // n表示点数, m表示边数
805 int dist[N];        // dist[x] 存储1到x的最短路距离
806
807 struct Edge         // 边, a表示出点, b表示入点, w表示边的权重
808 {
809     int a, b, w;
810 } edges[M];
811
812 // 求1到n的最短路距离, 如果无法从1走到n, 则返回-1。
813 int bellman_ford()
814 {
815     memset(dist, 0x3f, sizeof dist);
816     dist[1] = 0;
817
818     // 如果第n次迭代仍然会松弛三角不等式, 就说明存在一条长度是n+1的最短路径, 由抽屉
      原理, 路径中至少存在两个相同的点, 说明图中存在负权回路。
819     for (int i = 0; i < n; i++)
820     {
821         for (int j = 0; j < m; j++)
822         {
823             int a = edges[j].a, b = edges[j].b, w = edges[j].w;
824             if (dist[b] > dist[a] + w)
825                 dist[b] = dist[a] + w;
826         }
827     }
828

```

```

829     if (dist[n] > 0x3f3f3f3f / 2) return -1;
830     return dist[n];
831 }
832 spfa 算法 (队列优化的Bellman - Ford算法) — 模板题 AcWing 851. spfa求最短路
833 时间复杂度 平均情况下  $O(m)O(m)$  , 最坏情况下  $O(nm)O(nm)$  , nn 表示点数 , mm 表示边数
834 int n;          // 总点数
835 int h[N], w[N], e[N], ne[N], idx;          // 邻接表存储所有边
836 int dist[N];      // 存储每个点到1号点的最短距离
837 bool st[N];       // 存储每个点是否在队列中
838
839 // 求1号点到n号点的最短路距离, 如果从1号点无法走到n号点则返回-1
840 int spfa()
841 {
842     memset(dist, 0x3f, sizeof dist);
843     dist[1] = 0;
844
845     queue<int> q;
846     q.push(1);
847     st[1] = true;
848
849     while (q.size())
850     {
851         auto t = q.front();
852         q.pop();
853
854         st[t] = false;
855
856         for (int i = h[t]; i != -1; i = ne[i])
857         {
858             int j = e[i];
859             if (dist[j] > dist[t] + w[i])
860             {
861                 dist[j] = dist[t] + w[i];
862                 if (!st[j]) // 如果队列中已存在j, 则不需要将j重复插入
863                 {
864                     q.push(j);
865                     st[j] = true;
866                 }
867             }
868         }
869     }
870 }

```

```

869     }
870
871     if (dist[n] == 0x3f3f3f3f) return -1;
872     return dist[n];
873 }
874 spfa判断图中是否存在负环 — 模板题 AcWing 852. spfa判断负环
875 时间复杂度是  $O(nm)$   $O(nm)$ , nn 表示点数, mm 表示边数
876 int n;          // 总点数
877 int h[N], w[N], e[N], ne[N], idx;          // 邻接表存储所有边
878 int dist[N], cnt[N];          // dist[x]存储1号点到x的最短距离, cnt[x]存储1到x
    的最短路中经过的点数
879 bool st[N];          // 存储每个点是否在队列中
880
881 // 如果存在负环, 则返回true, 否则返回false。
882 bool spfa()
883 {
884     // 不需要初始化dist数组
885     // 原理: 如果某条最短路径上有n个点 (除了自己), 那么加上自己之后一共有n+1个点,
    由抽屉原理一定有两个点相同, 所以存在环。
886
887     queue<int> q;
888     for (int i = 1; i <= n; i++)
889     {
890         q.push(i);
891         st[i] = true;
892     }
893
894     while (q.size())
895     {
896         auto t = q.front();
897         q.pop();
898
899         st[t] = false;
900
901         for (int i = h[t]; i != -1; i = ne[i])
902         {
903             int j = e[i];
904             if (dist[j] > dist[t] + w[i])
905             {
906                 dist[j] = dist[t] + w[i];

```

```

907         cnt[j] = cnt[t] + 1;
908         if (cnt[j] >= n) return true;           // 如果从1号点到x的最
        短路中包含至少n个点（不包括自己），则说明存在环
909         if (!st[j])
910         {
911             q.push(j);
912             st[j] = true;
913         }
914     }
915 }
916 }
917
918 return false;
919 }
920 floyd算法 — 模板题 AcWing 854. Floyd求最短路
921 时间复杂度是  $O(n^3)$   $O(n^3)$ , nn 表示点数
922 初始化：
923 for (int i = 1; i <= n; i++)
924 for (int j = 1; j <= n; j++)
925 if (i == j) d[i][j] = 0;
926 else d[i][j] = INF;
927
928 // 算法结束后，d[a][b]表示a到b的最短距离
929 void floyd()
930 {
931     for (int k = 1; k <= n; k++)
932         for (int i = 1; i <= n; i++)
933             for (int j = 1; j <= n; j++)
934                 d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
935 }
936 朴素版prim算法 — 模板题 AcWing 858. Prim算法求最小生成树
937 时间复杂度是  $O(n^2 + m)$   $O(n^2 + m)$ , nn 表示点数，mm 表示边数
938 int n;           // n表示点数
939 int g[N][N];      // 邻接矩阵，存储所有边
940 int dist[N];       // 存储其他点到当前最小生成树的距离
941 bool st[N];        // 存储每个点是否已经在生成树中
942
943
944 // 如果图不连通，则返回INF(值是0x3f3f3f3f)，否则返回最小生成树的树边权重之和
945 int prim()

```

```

946 {
947     memset(dist, 0x3f, sizeof dist);
948
949     int res = 0;
950     for (int i = 0; i < n; i++)
951     {
952         int t = -1;
953         for (int j = 1; j <= n; j++)
954             if (!st[j] && (t == -1 || dist[t] > dist[j]))
955                 t = j;
956
957         if (i && dist[t] == INF) return INF;
958
959         if (i) res += dist[t];
960         st[t] = true;
961
962         for (int j = 1; j <= n; j++) dist[j] = min(dist[j], g[t][j]);
963     }
964
965     return res;
966 }
967 Kruskal算法 — 模板题 AcWing 859. Kruskal算法求最小生成树
968 时间复杂度是  $O(m \log m)$   $O(m \log m)$ , nn 表示点数, mm 表示边数
969 int n, m;          // n是点数, m是边数
970 int p[N];          // 并查集的父节点数组
971
972 struct Edge        // 存储边
973 {
974     int a, b, w;
975
976     bool operator< (const Edge& W) const
977     {
978         return w < W.w;
979     }
980 }edges[M];
981
982 int find(int x)      // 并查集核心操作
983 {
984     if (p[x] != x) p[x] = find(p[x]);
985     return p[x];

```

```

986 }
987
988 int kruskal()
989 {
990     sort(edges, edges + m);
991
992     for (int i = 1; i <= n; i++) p[i] = i;    // 初始化并查集
993
994     int res = 0, cnt = 0;
995     for (int i = 0; i < m; i++)
996     {
997         int a = edges[i].a, b = edges[i].b, w = edges[i].w;
998
999         a = find(a), b = find(b);
1000         if (a != b)    // 如果两个连通块不连通，则将这两个连通块合并
1001         {
1002             p[a] = b;
1003             res += w;
1004             cnt++;
1005         }
1006     }
1007
1008     if (cnt < n - 1) return INF;
1009     return res;
1010 }
1011 染色法判别二分图 — 模板题 AcWing 860. 染色法判定二分图
1012 时间复杂度是  $O(n + m)$   $O(n + m)$ , nn 表示点数, mm 表示边数
1013 int n;    // n表示点数
1014 int h[N], e[M], ne[M], idx;    // 邻接表存储图
1015 int color[N];    // 表示每个点的颜色, -1表示未染色, 0表示白色, 1表示黑色
1016
1017 // 参数: u表示当前节点, c表示当前点的颜色
1018 bool dfs(int u, int c)
1019 {
1020     color[u] = c;
1021     for (int i = h[u]; i != -1; i = ne[i])
1022     {
1023         int j = e[i];
1024         if (color[j] == -1)
1025         {

```

```

1026         if (!dfs(j, !c)) return false;
1027     }
1028     else if (color[j] == c) return false;
1029 }
1030
1031 return true;
1032 }
1033
1034 bool check()
1035 {
1036     memset(color, -1, sizeof color);
1037     bool flag = true;
1038     for (int i = 1; i <= n; i++)
1039         if (color[i] == -1)
1040             if (!dfs(i, 0))
1041             {
1042                 flag = false;
1043                 break;
1044             }
1045     return flag;
1046 }
1047 匈牙利算法 — 模板题 AcWing 861. 二分图的最大匹配
1048 时间复杂度是  $O(nm)$   $O(nm)$ , nn 表示点数, mm 表示边数
1049 int n1, n2;        // n1表示第一个集合中的点数, n2表示第二个集合中的点数
1050 int h[N], e[M], ne[M], idx;    // 邻接表存储所有边, 匈牙利算法中只会用到从第
    一个集合指向第二个集合的边, 所以这里只用存一个方向的边
1051 int match[N];        // 存储第二个集合中的每个点当前匹配的第一个集合中的点是哪个
1052 bool st[N];        // 表示第二个集合中的每个点是否已经被遍历过
1053
1054 bool find(int x)
1055 {
1056     for (int i = h[x]; i != -1; i = ne[i])
1057     {
1058         int j = e[i];
1059         if (!st[j])
1060         {
1061             st[j] = true;
1062             if (match[j] == 0 || find(match[j]))
1063             {
1064                 match[j] = x;

```

```

1065         return true;
1066     }
1067 }
1068 }
1069
1070     return false;
1071 }
1072
1073 // 求最大匹配数，依次枚举第一个集合中的每个点能否匹配第二个集合中的点
1074 int res = 0;
1075 for (int i = 1; i <= n1; i++)
1076 {
1077     memset(st, false, sizeof st);
1078     if (find(i)) res++;
1079 }
1080
1081 试除法判定质数 — 模板题 AcWing 866. 试除法判定质数
1082 bool is_prime(int x)
1083 {
1084     if (x < 2) return false;
1085     for (int i = 2; i <= x / i; i++)
1086         if (x % i == 0)
1087             return false;
1088     return true;
1089 }
1090 试除法分解质因数 — 模板题 AcWing 867. 分解质因数
1091 void divide(int x)
1092 {
1093     for (int i = 2; i <= x / i; i++)
1094         if (x % i == 0)
1095         {
1096             int s = 0;
1097             while (x % i == 0) x /= i, s++;
1098             cout << i << ' ' << s << endl;
1099         }
1100     if (x > 1) cout << x << ' ' << 1 << endl;
1101     cout << endl;
1102 }
1103 朴素筛法求素数 — 模板题 AcWing 868. 筛质数
1104 int primes[N], cnt; // primes[]存储所有素数

```

```

1105 bool st[N];          // st[x]存储x是否被筛掉
1106
1107 void get_primes(int n)
1108 {
1109     for (int i = 2; i <= n; i++)
1110     {
1111         if (st[i]) continue;
1112         primes[cnt++] = i;
1113         for (int j = i + i; j <= n; j += i)
1114             st[j] = true;
1115     }
1116 }
1117 线性筛法求素数 — 模板题 AcWing 868. 筛质数
1118 int primes[N], cnt;    // primes[]存储所有素数
1119 bool st[N];           // st[x]存储x是否被筛掉
1120
1121 void get_primes(int n)
1122 {
1123     for (int i = 2; i <= n; i++)
1124     {
1125         if (!st[i]) primes[cnt++] = i;
1126         for (int j = 0; primes[j] <= n / i; j++)
1127         {
1128             st[primes[j] * i] = true;
1129             if (i % primes[j] == 0) break;
1130         }
1131     }
1132 }
1133 试除法求所有约数 — 模板题 AcWing 869. 试除法求约数
1134 vector<int> get_divisors(int x)
1135 {
1136     vector<int> res;
1137     for (int i = 1; i <= x / i; i++)
1138         if (x % i == 0)
1139         {
1140             res.push_back(i);
1141             if (i != x / i) res.push_back(x / i);
1142         }
1143     sort(res.begin(), res.end());
1144     return res;

```

```

1145 }
1146 约数个数和约数之和 — 模板题 AcWing 870. 约数个数, AcWing 871. 约数之和
1147 如果  $N = p_1^{c_1} * p_2^{c_2} * \dots * p_k^{c_k}$ 
1148 约数个数:  $(c_1 + 1) * (c_2 + 1) * \dots * (c_k + 1)$ 
1149 约数之和:  $(p_1^0 + p_1^1 + \dots + p_1^{c_1}) * \dots * (p_k^0 + p_k^1 + \dots + p_k^{c_k})$ 
1150 欧几里得算法 — 模板题 AcWing 872. 最大公约数
1151 int gcd(int a, int b)
1152 {
1153     return b ? gcd(b, a % b) : a;
1154 }
1155 求欧拉函数 — 模板题 AcWing 873. 欧拉函数
1156 int phi(int x)
1157 {
1158     int res = x;
1159     for (int i = 2; i <= x / i; i++)
1160         if (x % i == 0)
1161         {
1162             res = res / i * (i - 1);
1163             while (x % i == 0) x /= i;
1164         }
1165     if (x > 1) res = res / x * (x - 1);
1166
1167     return res;
1168 }
1169 筛法求欧拉函数 — 模板题 AcWing 874. 筛法求欧拉函数
1170 int primes[N], cnt; // primes[]存储所有素数
1171 int euler[N]; // 存储每个数的欧拉函数
1172 bool st[N]; // st[x]存储x是否被筛掉
1173
1174
1175 void get_eulers(int n)
1176 {
1177     euler[1] = 1;
1178     for (int i = 2; i <= n; i++)
1179     {
1180         if (!st[i])
1181         {
1182             primes[cnt++] = i;
1183             euler[i] = i - 1;

```

```

1184     }
1185     for (int j = 0; primes[j] <= n / i; j++)
1186     {
1187         int t = primes[j] * i;
1188         st[t] = true;
1189         if (i % primes[j] == 0)
1190         {
1191             euler[t] = euler[i] * primes[j];
1192             break;
1193         }
1194         euler[t] = euler[i] * (primes[j] - 1);
1195     }
1196 }
1197 }

```

1198 快速幂 — 模板题 AcWing 875. 快速幂

1199 求 $m^k \bmod p$, 时间复杂度 $O(\log k)$ 。

```

1200
1201 int qmi(int m, int k, int p)
1202 {
1203     int res = 1 % p, t = m;
1204     while (k)
1205     {
1206         if (k & 1) res = res * t % p;
1207         t = t * t % p;
1208         k >>= 1;
1209     }
1210     return res;
1211 }

```

1212 扩展欧几里得算法 — 模板题 AcWing 877. 扩展欧几里得算法

1213 // 求 x, y , 使得 $ax + by = \gcd(a, b)$

```

1214 int exgcd(int a, int b, int& x, int& y)
1215 {
1216     if (!b)
1217     {
1218         x = 1; y = 0;
1219         return a;
1220     }
1221     int d = exgcd(b, a % b, y, x);
1222     y -= (a / b) * x;
1223     return d;

```

```

1224 }
1225 高斯消元 — 模板题 AcWing 883. 高斯消元解线性方程组
1226 // a[N][N]是增广矩阵
1227 int gauss()
1228 {
1229     int c, r;
1230     for (c = 0, r = 0; c < n; c++)
1231     {
1232         int t = r;
1233         for (int i = r; i < n; i++) // 找到绝对值最大的行
1234             if (fabs(a[i][c]) > fabs(a[t][c]))
1235                 t = i;
1236
1237         if (fabs(a[t][c]) < eps) continue;
1238
1239         for (int i = c; i <= n; i++) swap(a[t][i], a[r][i]); // 将
绝对值最大的行换到最顶端
1240         for (int i = n; i >= c; i--) a[r][i] /= a[r][c]; // 将当前
行的首位变成1
1241         for (int i = r + 1; i < n; i++) // 用当前行将下面所有的列消成
0
1242             if (fabs(a[i][c]) > eps)
1243                 for (int j = n; j >= c; j--)
1244                     a[i][j] -= a[r][j] * a[i][c];
1245
1246         r++;
1247     }
1248
1249     if (r < n)
1250     {
1251         for (int i = r; i < n; i++)
1252             if (fabs(a[i][n]) > eps)
1253                 return 2; // 无解
1254         return 1; // 有无穷多组解
1255     }
1256
1257     for (int i = n - 1; i >= 0; i--)
1258         for (int j = i + 1; j < n; j++)
1259             a[i][n] -= a[i][j] * a[j][n];
1260

```

```

1261     return 0; // 有唯一解
1262 }
1263 递推法求组合数 — 模板题 AcWing 885. 求组合数 I
1264 // c[a][b] 表示从a个苹果中选b个的方案数
1265 for (int i = 0; i < N; i++)
1266     for (int j = 0; j <= i; j++)
1267         if (!j) c[i][j] = 1;
1268         else c[i][j] = (c[i - 1][j] + c[i - 1][j - 1]) % mod;
1269 通过预处理逆元的方式求组合数 — 模板题 AcWing 886. 求组合数 II
1270 首先预处理出所有阶乘取模的余数fact[N]，以及所有阶乘取模的逆元infact[N]
1271 如果取模的数是质数，可以用费马小定理求逆元
1272 int qmi(int a, int k, int p) // 快速幂模板
1273 {
1274     int res = 1;
1275     while (k)
1276     {
1277         if (k & 1) res = (LL)res * a % p;
1278         a = (LL)a * a % p;
1279         k >>= 1;
1280     }
1281     return res;
1282 }
1283
1284 // 预处理阶乘的余数和阶乘逆元的余数
1285 fact[0] = infact[0] = 1;
1286 for (int i = 1; i < N; i++)
1287 {
1288     fact[i] = (LL)fact[i - 1] * i % mod;
1289     infact[i] = (LL)infact[i - 1] * qmi(i, mod - 2, mod) % mod;
1290 }
1291 Lucas定理 — 模板题 AcWing 887. 求组合数 III
1292 若p是质数，则对于任意整数  $1 \leq m \leq n$ ，有：
1293  $C(n, m) = C(n \% p, m \% p) * C(n / p, m / p) \pmod{p}$ 
1294
1295 int qmi(int a, int k, int p) // 快速幂模板
1296 {
1297     int res = 1 % p;
1298     while (k)
1299     {
1300         if (k & 1) res = (LL)res * a % p;

```

```

1301     a = (LL)a * a % p;
1302     k >>= 1;
1303 }
1304 return res;
1305 }
1306
1307 int C(int a, int b, int p) // 通过定理求组合数C(a, b)
1308 {
1309     if (a < b) return 0;
1310
1311     LL x = 1, y = 1; // x是分子, y是分母
1312     for (int i = a, j = 1; j <= b; i--, j++)
1313     {
1314         x = (LL)x * i % p;
1315         y = (LL)y * j % p;
1316     }
1317
1318     return x * (LL)qmi(y, p - 2, p) % p;
1319 }
1320
1321 int lucas(LL a, LL b, int p)
1322 {
1323     if (a < p && b < p) return C(a, b, p);
1324     return (LL)C(a % p, b % p, p) * lucas(a / p, b / p, p) % p;
1325 }
1326 分解质因数法求组合数 — 模板题 AcWing 888. 求组合数 IV
1327 当我们需要求出组合数的真实值, 而非对某个数的余数时, 分解质因数的方式比较好用:
1328 1. 筛法求出范围内的所有质数
1329 2. 通过  $C(a, b) = a! / b! / (a - b)!$  这个公式求出每个质因子的次数。  $n!$  中  $p$  的次数
    是  $n / p + n / p^2 + n / p^3 + \dots$ 
1330 3. 用高精度乘法将所有质因子相乘
1331
1332 int primes[N], cnt; // 存储所有质数
1333 int sum[N]; // 存储每个质数的次数
1334 bool st[N]; // 存储每个数是否已被筛掉
1335
1336
1337 void get_primes(int n) // 线性筛法求素数
1338 {
1339     for (int i = 2; i <= n; i++)

```

```

1340     {
1341         if (!st[i]) primes[cnt++] = i;
1342         for (int j = 0; primes[j] <= n / i; j++)
1343         {
1344             st[primes[j] * i] = true;
1345             if (i % primes[j] == 0) break;
1346         }
1347     }
1348 }
1349
1350
1351 int get(int n, int p)          // 求n!中的次数
1352 {
1353     int res = 0;
1354     while (n)
1355     {
1356         res += n / p;
1357         n /= p;
1358     }
1359     return res;
1360 }
1361
1362
1363 vector<int> mul(vector<int> a, int b)          // 高精度乘低精度模板
1364 {
1365     vector<int> c;
1366     int t = 0;
1367     for (int i = 0; i < a.size(); i++)
1368     {
1369         t += a[i] * b;
1370         c.push_back(t % 10);
1371         t /= 10;
1372     }
1373
1374     while (t)
1375     {
1376         c.push_back(t % 10);
1377         t /= 10;
1378     }
1379 }

```

```

1380     return c;
1381 }
1382
1383 get_primes(a); // 预处理范围内的所有质数
1384
1385 for (int i = 0; i < cnt; i++) // 求每个质因数的次数
1386 {
1387     int p = primes[i];
1388     sum[i] = get(a, p) - get(b, p) - get(a - b, p);
1389 }
1390
1391 vector<int> res;
1392 res.push_back(1);
1393
1394 for (int i = 0; i < cnt; i++) // 用高精度乘法将所有质因子相乘
1395     for (int j = 0; j < sum[i]; j++)
1396         res = mul(res, primes[i]);
1397
1398 卡特兰数 — 模板题 AcWing 889. 满足条件的01序列
1399 给定n个0和n个1，它们按照某种顺序排成长度为2n的序列，满足任意前缀中0的个数都不少于1
    的个数的序列的数量为：  $Cat(n) = C(2n, n) / (n + 1)$ 
1400 NIM游戏 — 模板题 AcWing 891. Nim游戏
1401 给定N堆物品，第i堆物品有Ai个。两名玩家轮流行动，每次可以任选一堆，取走任意多个物品，
    可把一堆取光，但不能不取。取走最后一件物品者获胜。两人都采取最优策略，问先手是否必胜。
1402
1403 我们把这种游戏称为NIM博弈。把游戏过程中面临的状态称为局面。整局游戏第一个行动的称为先
    手，第二个行动的称为后手。若在某一局面下无论采取何种行动，都会输掉游戏，则称该局面必
    败。
1404 所谓采取最优策略是指，若在某一局面下存在某种行动，使得行动后对面面临必败局面，则优先采
    取该行动。同时，这样的局面被称为必胜。我们讨论的博弈问题一般都只考虑理想情况，即两人都
    无失误，都采取最优策略行动时游戏的结果。
1405 NIM博弈不存在平局，只有先手必胜和先手必败两种情况。
1406
1407 定理： NIM博弈先手必胜，当且仅当  $A_1 \oplus A_2 \oplus \dots \oplus A_n \neq 0$ 
1408
1409 公平组合游戏ICG
1410 若一个游戏满足：
1411
1412 由两名玩家交替行动；
1413 在游戏进程的任意时刻，可以执行的合法行动与轮到哪名玩家无关；

```

1414 不能行动的玩家判负；

1415 则称该游戏为一个公平组合游戏。

1416 NIM博弈属于公平组合游戏，但城建的棋类游戏，比如围棋，就不是公平组合游戏。因为围棋交战双方分别只能落黑子和白子，胜负判定也比较复杂，不满足条件2和条件3。

1417

1418 有向图游戏

1419 给定一个有向无环图，图中有一个唯一的起点，在起点上放有一枚棋子。两名玩家交替地把这枚棋子沿有向边进行移动，每次可以移动一步，无法移动者判负。该游戏被称为有向图游戏。

1420 任何一个公平组合游戏都可以转化为有向图游戏。具体方法是，把每个局面看成图中的一个节点，并且从每个局面向沿着合法行动能够到达的下一个局面连有向边。

1421

1422 Mex运算

1423 设 S 表示一个非负整数集合。定义 $\text{mex}(S)$ 为求出不属于集合 S 的最小非负整数的运算，即：

1424 $\text{mex}(S) = \min\{x\}$ ， x 属于自然数，且 x 不属于 S

1425

1426 SG函数

1427 在有向图游戏中，对于每个节点 x ，设从 x 出发共有 k 条有向边，分别到达节点 $y_1, y_2, \dots, y_k$ ，定义 $\text{SG}(x)$ 为 x 的后继节点 $y_1, y_2, \dots, y_k$ 的SG函数值构成的集合再执行 $\text{mex}(S)$ 运算的结果，即：

1428 $\text{SG}(x) = \text{mex}(\{\text{SG}(y_1), \text{SG}(y_2), \dots, \text{SG}(y_k)\})$

1429 特别地，整个有向图游戏 G 的SG函数值被定义为有向图游戏起点 s 的SG函数值，即 $\text{SG}(G) = \text{SG}(s)$ 。

1430

1431 有向图游戏的和 — 模板题 AcWing 893. 集合 - Nim游戏

1432 设 $G_1, G_2, \dots, G_m$ 是 m 个有向图游戏。定义有向图游戏 G ，它的行动规则是任选某个有向图游戏 G_i ，并在 G_i 上行动一步。 G 被称为有向图游戏 $G_1, G_2, \dots, G_m$ 的和。

1433 有向图游戏的和的SG函数值等于它包含的各个子游戏SG函数值的异或和，即：

1434 $\text{SG}(G) = \text{SG}(G_1) \oplus \text{SG}(G_2) \oplus \dots \oplus \text{SG}(G_m)$

1435

1436 定理

1437 有向图游戏的某个局面必胜，当且仅当该局面对应节点的SG函数值大于0。

1438 有向图游戏的某个局面必败，当且仅当该局面对应节点的SG函数值等于0。

1439

1440 //多重背包不能和完全背包一样优化

1441 //完全背包的优化方式是：

1442 /*

1443 $f[i][j]$ 的集合是只考虑前 i 种物品的，体积总不超过 j 的所有方案的集合

1444 这个集合可以分为，第 i 种物品选了0个，选了1个，选了2个...选了 k 个这些子集， k 表示满足 $k \times v[i] \leq j$ 的最大值

1445 那么，选了 k 个 i 的那个子集，他们所遇的方案都是两部分， k 个 i ，占据 $k \times v[i]$ ，然后剩下 j

-k*v[i]的是从前i-1种里面随便挑

1446 那么就是, $f[i-1][j-k*v[i]]+k*w[i]$;

1447 $f[i][j]$ 就是所有这些子集的最大值

1448

1449 优化:

1450 $f[i][j]=\max(f[i-1][j], f[i-1][j-v]+w, f[i-1][j-2v]+2w, f[i-1][j-3v]+3w, \dots, f[i-1][j-kv]+kw)$

1451 $f[i][j-v]=\max(f[i-1][j-v], f[i-1][j-2v]+w, f[i-1][j-3v]+2w, \dots, f[i-1][j-kv]+(k-1)w)$

1452

1453 然后可以发现, $f[i][j]=\max(f[i-1][j], f[i][j-v]+w)$;

1454

1455 这里之所以可以这么优化,是因为k是我们定义的体积j最多可以放的下的v的个数,如果j最多可以放下k*v,

1456 那么j-v最多可以放下 (k-1) *v;

1457 因此 $f[i][j]$ 和 $f[i][j-v]$ 中的k的取值肯定是相差1的, $f[i][j-v]$ 中是0~k-1共k项,而 $f[i][j]$ 是0~k共k+1项,

1458 $f[i][j]$ 的后k项正好和 $f[i][j-v]$ 的k项每项差值为w

1459

1460 但是我们在多重背包里,每个物品最多取s个,有可能,j-v和j都可以装的下这s个,因此,会导致:

1461 $f[i][j]=\max(f[i-1][j], f[i-1][j-v]+w, f[i-1][j-2v]+2w, f[i-1][j-3v]+3w, \dots, f[i-1][j-sv]+sw)$

1462 $f[i][j-v]=\max(f[i-1][j-v], f[i-1][j-2v]+w, f[i-1][j-3v]+2w, \dots, f[i-1][j-sv]+(s-1)w+f[i-1][j-(s+1)v])$

1463

1464 那么,都是k+1项,错位就多出来一项,而我们没有办法在知道k+1个数的最大值和最后一个数的同时求出来前k个数的最大值,就没有办法优化

1465 */

1466 /*

1467 用二进制优化,也就是说,我么对于之前的暴力做法,遍历每种物品可以选0个,1个,2个,一直到不能装得下,实际上不用遍历那么多次,

1468 因为我们可以1个编成一组,2个编成一组,4个编成一组,8个编成一组...以此类推

1469 设k是满足 $1+2+4+8+\dots+2^k=2^{k+1}-1 \leq s$ 的最后一个k

1470 那么,所有的数字s一定可以写成

1471 $1+2+4+8+\dots+2^k+c=s$;

1472 c一定是小于 2^{k+1} 的,否则就应该把c再写成 $2^{k+1}+c'$

1473

1474 那么,已知 $1,2,4,8,\dots,2^k$ 可以组合出来 $[0, 2^{k+1}-1]$ 之间的所有的数(整数)

1475 那么加上c之后,就可以组合出来 c到 $2^{k+1}-1+c=s$,也就是 $[c, s]$ 之间的所有的数

```

1476
1477     那么,  $1, 2, 4, 8, \dots, 2^k$ ,  $c$  就可以表示出  $[0, 2^{(k+1)}-1]$  并上  $[c, s]$  的所有整数, 由
    于  $c$  一定时小于  $2^{(k+1)}$  的,
1478     因此,  $c$  一定小于等于  $2^{(k+1)}-1$ , 所以两个集合并起来就是  $[0, s]$ 
1479
1480     因此, 我们可以用  $0, i, 2i, 4i, \dots, 2^{ki}, ci$  把我们暴力算法中考虑的如果选了第  $i$  种物
    品, 考虑的选了  $0$  个,  $1$  个  $\dots$  最多  $s$  个的所有可能
1481     也就是说, 我们只需要把所有种类的物品, 都拆分成一个个捆绑包, 就可以每个捆绑包只取
    一次取出任何一种可能的方案
1482     那么就是对于这些新得物品进行 01 背包
1483 */
1484
1485 #include<iostream>
1486 using namespace std;
1487
1488 const int maxn = 2020;
1489 int N, V;
1490 int v[11 * maxn], w[11 * maxn];    //用于存储拆分出来的新的物品的体积和价值,
    由于旧的物品每种最多2000, 也就是说只需要11位就能够表示,
1491 //这里就开成maxn*11
1492 //对新物品进行01背包(空间优化后的一维的)
1493 int f[2020];    //f[i][j]表示的只考虑前i个物品的, 体积总不超过j的所有方
    案的集合中, 最大的价值
1494 //总共maxn*11新物品, 总体积不超过2000
1495 int main()
1496 {
1497     cin >> N >> V;
1498     int cnt = 0;    //用与记录分了多少个新物品出来, 也作为
    新物品的下标
1499     for (int i = 1; i <= N; i++)
1500     {
1501         int a, b, s;    //当前种类物品的体积, 价值, 个数
1502         cin >> a >> b >> s;
1503
1504         //对s进行拆分
1505         int k = 1;    //表示分成1, 2, 4, 8, \dots
1506         while (k <= s)    //k还小于等于s, 那就可以继续分
1507         {
1508             cnt++;    //表示整体的新物品的数量和下标, 从1开始
1509

```

```

1510         v[cnt] = a * k;           //单个为a，k个物品绑定在一起，体积就
    是a*k
1511         w[cnt] = b * k;           //价值
1512
1513         s -= k;                     //已经把k拆分出来了
1514         k *= 2;                     //k变成下一个2的幂
1515
1516     }
1517
1518     if (s > 0)                       //还剩下的s的值就是那个c
1519     {
1520         cnt++;
1521         v[cnt] = a * s;
1522         w[cnt] = b * s;
1523     }
1524
1525
1526 }
1527 //cnt表示新物品的总数量
1528 int n = cnt;
1529 for (int i = 1; i <= n; i++)
1530 {
1531     for (int j = V; j >= v[i]; j--)
1532     {
1533         f[j] = max(f[j], f[j - v[i]] + w[i]);
1534     }
1535 }
1536
1537
1538
1539 cout << f[V];
1540
1541 return 0;
1542 }
1543
1544 /*
1545     对于所有长度的上升子序列，记录最后一个数的最小值，因为后面的数肯定会加到最后一个
    数最小的序列上面
1546     q[i]表示长度为i的上升子序列的最后一个数的最小值，可以证明，q是递增的，q[i+1]一
    定要大于q[i] 否则的话，就是说，

```

```

1547     长度为i+1的上升子序列的第i位小于q[i]，那q[i]就不是长度为i的上升子序列的最后
        一位数的最小值了
1548
1549     遍历一个数x的话，就在q中找到最大的小于x的，把x接在这个长度的子序列后面，子序列长
        度加一，那么就要更新q的下一位的值
1550 */
1551
1552 #include<iostream>
1553 #include<algorithm>
1554 using namespace std;
1555
1556 const int N = 100020;
1557
1558 int n;
1559
1560 int a[N], q[N];
1561
1562 int main()
1563 {
1564     cin >> n;
1565
1566     for (int i = 1; i <= n; i++)
1567     {
1568         cin >> a[i];
1569     }
1570
1571     int len = 0; //记录当前最大的长度
1572
1573     for (int i = 1; i <= n; i++) //遍历所有的元素，选择
        可以添加上去的长度最大的上升子序列添加
1574     {
1575         int l = 0, r = len; //这里和i无关，主要是
        说，长度可以为0,1,2,3,...
1576
1577         while (l < r)
1578         {
1579             int mid = (l + r + 1) / 2;
1580
1581             if (q[mid] < a[i])
1582             {

```

```

1583         l = mid;
1584     }
1585     else
1586     {
1587         r = mid - 1;
1588     }
1589 }
1590 len = max(len, r + 1);
1591 q[r + 1] = a[i];
1592 }
1593 cout << len << endl;
1594
1595 return 0;
1596 }
1597 //最长公共子序列
1598
1599 #include<iostream>
1600 using namespace std;
1601 #include<string>
1602 const int maxn = 10010;
1603
1604 int d[maxn][maxn];
1605 char a[maxn], b[maxn];
1606
1607 int n, m;
1608
1609
1610 //涉及两个字符串的动态规划，一般都是第一个字符串前i个字符和第二个字符串前j个字符
1611 //这一题的话，我们用d[i][j]表示第一个字符串前i个字符和第二个字符串前j个字符的所有最
    长公共子序列的最大值
1612
1613 //而第一个字符串前i个字符和第二个字符串前j个字符的所有最长公共子序列可以划分成，a
    [i], b[j]都不取，取a[i]不取b[j],取b[j]不取a[i],都不
1614 //显然答案肯定不是第一种里面的，而应该在后面三种，中间那两种看起来和d[i][j-1]和d[i]
    [j-1]很像，但实际上不是的，因为d[i][j-1]和d[i][j-1]不一定一定含有a[i] (b[j])，但
    是我们求的是最大值，所以只要能覆盖就行
1615 //那么就可以用这两个量来做，如果a[i]==b[j]，那么再引入d[i-1][j-1]+1
1616
1617
1618 int main()

```

```

1619 {
1620     cin >> n >> m;
1621
1622     for (int i = 1; i <= n; i++)
1623     {
1624         cin >> a[i];
1625     }
1626     for (int i = 1; i <= m; i++)
1627     {
1628         cin >> b[i];
1629     }
1630     for (int i = 1; i <= n; i++)
1631     {
1632         for (int j = 1; j <= m; j++)
1633         {
1634             d[i][j] = max(d[i][j - 1], d[i - 1][j]);
1635             if (a[i] == b[j])
1636             {
1637                 d[i][j] = max(d[i][j], d[i - 1][j - 1] + 1);
1638             }
1639         }
1640     }
1641
1642     cout << d[n][m];
1643
1644     return 0;
1645 }
1646
1647
1648 //最短编辑距离
1649 /*
1650     两个字符串的问题，状态表示一般都 $f[i, j]$ ，表示第一个字符串的前 $i$ 个字符，第二个字符串的前 $j$ 个字符
1651     这里用 $f[i, j]$ 表示，所有的把第一个字符串的前 $i$ 个字符变为第二个字符串的前 $j$ 个字符的方案，操作次数的最小值
1652
1653     那么接下来就是要把 $f[i, j]$ 表示的集合进行划分，通常划分依据都是看最后一个元素（操作）
1654
1655     本题里，根据对于 $a[i]$ 的操作， $f[i, j]$ 表示的集合可以分为4种

```

```

1656
1657     删掉a[i]           那么f[i][j]=f[i-1][j]+1，就是前a[1~i-1]变成b[1~j]的方案
                        的最小值再加一
1658     在a[i]后面增加   f[i][j]=f[i][j-1]+1           //就是说a[1~i]变为b[1~j-1]，再
                        增加一个b[j]
1659     修改a[i]变成b[j] f[i][j]=f[i-1][j-1]+1;       //a[1~i-1]变为b[1~j-1]，再把
                        a[i]变为b[j]
1660     对a[i]不做操作   f[i][j]=f[i-1][j-1]           //不对a[i]操作，那么a[i]==b
                        [j]，这一类就相当于只考虑f[i-1][j-1]
1661
1662 */
1663
1664 #include<iostream>
1665 #include<cmath>
1666 using namespace std;
1667
1668 const int maxn = 1020;
1669
1670 char a[maxn], b[maxn];
1671
1672 int f[maxn][maxn];
1673
1674 int n, m;
1675
1676 int main()
1677 {
1678     cin >> n;
1679
1680     for (int i = 1; i <= n; i++)
1681     {
1682         cin >> a[i];
1683     }
1684     cin >> m;
1685     for (int i = 1; i <= m; i++)
1686     {
1687         cin >> b[i];
1688     }
1689     //初始化边界
1690     for (int i = 0; i <= m; i++)
1691     {

```

```

1692         f[0][i] = i;                                //用a的前0的字母变成b[0~m]，只能一直加
1693     }
1694     for (int i = 0; i <= n; i++)                        //a的前i个字符变为bd的前0个字母，
    只能一直减
1695     {
1696         f[i][0] = i;
1697     }
1698     for (int i = 1; i <= n; i++)
1699     {
1700         for (int j = 1; j <= m; j++)
1701         {
1702             f[i][j] = min(f[i - 1][j] + 1, f[i][j - 1] + 1);
1703             if (a[i] == b[j])
1704             {
1705                 f[i][j] = min(f[i][j], f[i - 1][j - 1]);
1706             }
1707             else
1708             {
1709                 f[i][j] = min(f[i][j], f[i - 1][j - 1] + 1);
1710             }
1711         }
1712     }
1713     cout << f[n][m] << endl;
1714
1715     return 0;
1716
1717 }
1718
1719 //区间dp
1720 /*
1721     用f[i, j]表示i到j这个区间的所有合并方法的最小值
1722     合并f[i, j]的最后一步一定是合并左右两个区间，那么就可以用最后一次的分界线的位置作
    为划分条件
1723     用k表示将[i, j]分为[i, k]和[k+1, j]，所以f[i, j]表示的所有方案可以分类为
1724     k=i, i+1, i+2, ..., j-1
1725
1726     每一种情况的结果是f[i, k]+f[k+1, j]+s[i, j]                                //最后一次合
    并的代价就是整个区间所有的元素之和
1727
1728     那么f[i, j]就是k的所有取值情况导致的结果的最小值

```

```

1729
1730     状态数量是 $n^2$ , 计算次数因为要枚举 $k$ , 也就是 $n$ 次, 那么总的复杂度是 $n^3$ 
1731
1732     由于我们计算时要使用的状态是 $f[i, k] + f[k+1, j] + s[i, j]$ , 因此我们要用长度从小到大来枚举
1733
1734 */
1735
1736 #include<iostream>
1737 using namespace std;
1738 #include<cstring>
1739 const int maxn = 500;
1740
1741 int a[maxn], s[maxn], f[maxn][maxn];
1742 int n;
1743
1744
1745 int main()
1746 {
1747     cin >> n;
1748     for (int i = 1; i <= n; i++)
1749     {
1750         cin >> a[i];
1751         s[i] = a[i] + s[i - 1];
1752     }
1753
1754     //由于我们计算时要使用的状态是 $f[i, k] + f[k+1, j] + s[i, j]$ , 因此我们要用长度从小到大来枚举
1755     for (int len = 2; len <= n; len++) //长度从2到n, 因为长度为1没有代价
1756     {
1757         //枚举起点
1758         for (int i = 1; i + len - 1 <= n; i++)
1759         {
1760             int l = i, r = i + len - 1;
1761             f[l][r] = 1e9; //计算之前再初
1762             for (int k = l; k < r; k++)
1763             {
1764                 f[l][r] = min(f[l][r], f[l][k] + f[k + 1][r] + s[r] - s
[1 - 1]);

```

```

1765         }
1766     }
1767 }
1768
1769     cout << f[1][n] << endl;
1770
1771     return 0;
1772 }
1773
1774 //整数划分：
1775 #include<iostream>
1776 using namespace std;
1777 const int mod = 1e9 + 7;
1778 const int maxn = 1020;
1779
1780 int f[maxn][maxn];           //f[i][j]表示前i个数加成j的方案集合
                               的元素个数，那么就可以根据包不包含第i个数来划分
1781 //不包含第i个数的部分就是f[i-1][j]，
1782 //包含的话就是得考虑包含几个i，设j最多包含k个i
1783 //那么就应该是f[i-1][j-i]+f[i-1][j-2i]+f[i-1][j-3i]+...+f[i-1][j-ki]
1784 //那么f[i][j]=f[i-1][j]+f[i-1][j-i]+f[i-1][j-2i]+...+f[i-1][j-ki]
1785 //考虑j-i，如果j最多包含k个i，那么j-i就应该是最多k-1个i
1786 //那么f[i][j-1]=f[i-1][j-i]+f[i-1][j-2i]+...+f[i-1][j-i-(k-1)i]
1787 //于是，不难发现，f[i][j]=f[i-1][j]+f[i][j-i];
1788 //优化思路和完全背包类似
1789 int n;
1790
1791 int main()
1792 {
1793     cin >> n;
1794
1795     for (int i = 1; i <= n; i++)
1796     {
1797
1798         f[i][0] = 1;           //前任何数的到0的方案都只有一个，这
                               样可以避免f[i][i]少考虑i=i，也就是会变成i=i+0
1799     }
1800     for (int i = 1; i <= n; i++)
1801     {
1802         for (int j = 1; j <= n; j++)

```

```
1803     {
1804         f[i][j] = f[i - 1][j];
1805         if (j >= i)
1806         {
1807             f[i][j] += f[i][j - i] % mod;
1808             f[i][j] %= mod;
1809         }
1810     }
1811 }
1812 cout << f[n][n] << endl;
1813
1814 return 0;
1815 }
```

1816 优化为一维的：

```
1817
1818 #include<iostream>
1819 using namespace std;
1820 const int mod = 1e9 + 7;
1821 const int maxn = 1020;
1822
1823 int f[maxn];
1824
1825 int n;
1826
1827 int main()
1828 {
1829     cin >> n;
1830
1831     f[0] = 1;
1832     for (int i = 1; i <= n; i++)
1833     {
1834         for (int j = i; j <= n; j++)
1835         {
1836
1837             f[j] += f[j - i] % mod;
1838             f[j] %= mod;
1839
1840         }
1841     }
1842     cout << f[n] << endl;
```

```

1843
1844     return 0;
1845 }
1846
1847
1848 //数位dp
1849 /*小学奥数题
1850
1851     对于每个数，考虑这个数出现在第几位上的从0到n中的所有数字的个数，加在一起就是这个
    数出现的个数了
1852     比如考虑1在第四位的所有数字可以直接计算出来，那么把1出现在各个位置上的数字个数加
    在一起就是1出现的次数
1853
1854     同理处理2,3,4,5,6,6,7....
1855
1856 */
1857
1858 #include<iostream>
1859 #include<algorithm>
1860 #include<vector>
1861 using namespace std;
1862
1863 int get(vector<int>num, int l, int r)                                //数字是逆序存储
    的
1864 {
1865     int res = 0;
1866
1867     for (int i = l; i >= r; i--)
1868     {
1869         res = res * 10 + num[i];
1870     }
1871
1872     return res;
1873
1874 }
1875
1876 int power10(int x)                                                //求10的x次方
1877 {
1878     int res = 1;
1879

```

```

1880     while (x--)
1881     {
1882         res *= 10;
1883     }
1884
1885     return res;
1886 }
1887
1888 int count(int n, int x)
1889 {
1890     if (n == 0)
1891     {
1892         return 0;                //n==0,返回0
1893     }
1894
1895
1896     vector<int>num;
1897
1898     while (n)                    //取出n的数位
1899     {
1900         num.push_back(n % 10);
1901
1902         n /= 10;
1903     }
1904
1905     n = num.size();              //让n再表示位数
1906
1907     int res = 0;
1908
1909     for (int i = n - 1 - !x; i >= 0; i--)                //从最高位开始枚举,枚举
x出现在这一位上的个数      x==0时从第而位开始枚举
1910     {
1911         if (i < n - 1)                    //abcdefg                当枚举不是最
高位的时候,我们可以有第一种情况就是,
1912         //比如我们要求1出现在第4位上的数字个数,那么第一种情况就是,前面三位可
以取000~abc-1
1913         //这种情况的个数就是abc*10^3
1914         {
1915             res += get(num, n - 1, i + 1) * power10(i);
1916             if (x == 0)                    //x=0的时候,就不可以从000开始,而应

```

该从001开始

```
1917         {
1918             res -= power10(i);
1919         }
1920     }
1921
1922     //第二种情况就是前面的就固定为abc, 然后的话, 在第四位有这么几种可能,  $d < x$ ,
    那么x不能在第四位出现, 为0,  $d == x$ , 则只有000~efg个,  $d > x$ 时, 后面几位就可以随便选
1923
1924     if (num[i] == x)
1925     {
1926         res += get(num, i - 1, 0) + 1;
1927     }
1928     else if (num[i] > x)
1929     {
1930         res += power10(i);
1931     }
1932
1933 }
1934
1935 return res;
1936 }
1937
1938
1939
1940 int main()
1941 {
1942
1943     int a, b;
1944     while (cin >> a >> b, a || b) //直到输入的全是0才停止
1945     {
1946         if (a > b) swap(a, b);
1947
1948         for (int i = 0; i < 10; i++)
1949         {
1950             cout << count(b, i) - count(a - 1, i) << " ";
            //count(n,x)表示从0到n所有的数中, i出现的个数
1951         }
1952         cout << endl;
1953     }
```

```
1954 }
1955 //状态压缩dp
1956 //AcWing 291. 蒙德里安的梦想
1957
1958 //朴素写法, 1000ms
1959 #include <cstring>
1960 #include <iostream>
1961 #include <algorithm>
1962
1963 using namespace std;
1964
1965 const int N = 12, M = 1 << N;
1966
1967 int n, m;
1968 long long f[N][M];
1969 bool st[M];
1970
1971 int main()
1972 {
1973     while (cin >> n >> m, n || m)
1974     {
1975         for (int i = 0; i < 1 << n; i++)
1976         {
1977             int cnt = 0;
1978             st[i] = true;
1979             for (int j = 0; j < n; j++)
1980                 if (i >> j & 1)
1981                 {
1982                     if (cnt & 1) st[i] = false;
1983                     cnt = 0;
1984                 }
1985             else cnt++;
1986             if (cnt & 1) st[i] = false;
1987         }
1988
1989         memset(f, 0, sizeof f);
1990         f[0][0] = 1;
1991         for (int i = 1; i <= m; i++)
1992             for (int j = 0; j < 1 << n; j++)
1993                 for (int k = 0; k < 1 << n; k++)
```



```

2034         cnt = 0;
2035     }
2036     else cnt++;
2037     if (cnt & 1) is_valid = false;
2038     st[i] = is_valid;
2039 }
2040
2041 for (int i = 0; i < 1 << n; i++)
2042 {
2043     state[i].clear();
2044     for (int j = 0; j < 1 << n; j++)
2045         if ((i & j) == 0 && st[i | j])
2046             state[i].push_back(j);
2047 }
2048
2049 memset(f, 0, sizeof f);
2050 f[0][0] = 1;
2051 for (int i = 1; i <= m; i++)
2052     for (int j = 0; j < 1 << n; j++)
2053         for (auto k : state[j])
2054             f[i][j] += f[i - 1][k];
2055
2056 cout << f[m][0] << endl;
2057 }
2058
2059 return 0;
2060 }
2061
2062 //AcWing 91. 最短Hamilton路径
2063
2064 #include <cstring>
2065 #include <iostream>
2066 #include <algorithm>
2067
2068 using namespace std;
2069
2070 const int N = 20, M = 1 << N;
2071
2072 int n;
2073 int w[N][N];

```

```

2074 int f[M][N];
2075
2076 int main()
2077 {
2078     cin >> n;
2079     for (int i = 0; i < n; i++)
2080         for (int j = 0; j < n; j++)
2081             cin >> w[i][j];
2082
2083     memset(f, 0x3f, sizeof f);
2084     f[1][0] = 0;
2085
2086     for (int i = 0; i < 1 << n; i++)
2087         for (int j = 0; j < n; j++)
2088             if (i >> j & 1)
2089                 for (int k = 0; k < n; k++)
2090                     if (i >> k & 1)
2091                         f[i][j] = min(f[i][j], f[i - (1 << j)][k] + w
[k][j]);
2092
2093     cout << f[(1 << n) - 1][n - 1];
2094
2095     return 0;
2096 }

```

2098 题目

2099 285. 没有上司的舞会

2100

2101 题目描述

2102 Ural 大学有 N 名职员，编号为 $1 \sim N$ 。

2103

2104 他们的关系就像一棵以校长为根的树，父节点就是子节点的直接上司。

2105

2106 每个职员有一个快乐指数，用整数 H_i 给出，其中 $1 \leq i \leq N$ 。

2107

2108 现在要召开一场周年庆宴会，不过，没有职员愿意和直接上司一起参会。

2109

2110 在满足这个条件的前提下，主办方希望邀请一部分职员参会，使得所有参会职员的快乐指数总和最大，求这个最大值。

2111

2112 输入格式

2113 第一行一个整数 N 。

2114

2115 接下来 N 行，第 i 行表示 i 号职员快乐指数 H_i 。

2116

2117 接下来 $N-1$ 行，每行输入一对整数 L, K ，表示 K 是 L 的直接上司。

2118 输出格式

2119 输出最大的快乐指数。

2120 数据范围

2121 $1 \leq N \leq 6000$

2122 $-128 \leq H_i \leq 127$

2123 输入样例

2124 7

2125 1

2126 1

2127 1

2128 1

2129 1

2130 1

2131 1

2132 1 3

2133 2 3

2134 6 4

2135 7 4

2136 4 5

2137 3 5

2138 0 0

2139 输出样例

2140 5

2141 时间复杂度

2142 $O(n)$

2143

2144 树形DP

2145 就是在树或图上的一种DP，一般是某个父节点或子节点有特殊要求的时候用的一种DP

2146

2147 首先是建图，在图上遍历的时候进行DP操作，对于这道题来说我们用 $f(i, j)$ 来表示 i 这个节点，状态为 j (用0来表示不选，用1来表示选) 值的最大值，对于每个节点我们有俩种操作：

2148

2149 1. 选当前这个节点， j 状态为1，它的子节点只能不选，所以 $f(i, 1) = f(i, 1) + f(u, 0)$ (u 表示 i 的子节点)

```
2150
2151 2.不选当前这个节点, j 的状态为0, 它的子节点可以选, 也可以不选, 取俩者的最大值
2152
2153 所以  $f(i, 0) = f(i, 0) + \max(f(u, 1), f(u, 0))$  (u表示 i 的子节点)
2154
2155 3.从任意一个跟节点开始搜索, 所以还需要一个数组来储存哪些节点有父节点
2156
2157 C++代码
2158 #include <iostream>
2159 #include <algorithm>
2160 #include <cstring>
2161 using namespace std;
2162 const int N = 6010;
2163 int n;
2164 int happy[N];
2165 int f[N][2];
2166 int e[N], ne[N], h[N], idx;
2167 bool has_father[N];
2168 void add(int a, int b) {
2169     e[idx] = b, ne[idx] = h[a], h[a] = idx++;
2170 }
2171 void dfs(int u) {
2172     f[u][1] = happy[u];
2173     for (int i = h[u]; ~i; i = ne[i]) {
2174         int j = e[i];
2175         dfs(j);
2176
2177         f[u][0] += max(f[j][1], f[j][0]);
2178         f[u][1] += f[j][0];
2179     }
2180 }
2181 int main() {
2182     scanf("%d", &n);
2183     for (int i = 1; i <= n; i++) scanf("%d", &happy[i]);
2184     memset(h, -1, sizeof h);
2185     for (int i = 1; i < n; i++) {
2186         int a, b;
2187         scanf("%d%d", &a, &b);
2188         has_father[a] = true;
2189         add(b, a);
2190     }
```

```

2190     }
2191     int root = 1;
2192     while (has_father[root]) root++;
2193     dfs(root);
2194     printf("%d\n", max(f[root][0], f[root][1]));
2195     return 0;
2196 }
2197
2198 //记忆化搜索
2199
2200 /*题目描述
2201 给定一个 R 行 C
2202
2203 列的矩阵，表示一个矩形网格滑雪场。
2204
2205 矩阵中第 i 行第 j 列的点表示滑雪场的第 i 行第 j 列区域的高度。
2206
2207 一个人从滑雪场中的某个区域内出发，每次可以向上下左右任意一个方向滑动一个单位距离。
2208
2209 当然，一个人能够滑动到某相邻区域的前提是该区域的高度低于自己目前所在区域的高度。
2210
2211 下面给出一个矩阵作为例子：
2212
2213 1   2   3   4  5
2214
2215 16  17  18  19  6
2216
2217 15  24  25  20  7
2218
2219 14  23  22  21  8
2220
2221 13  12  11  10  9
2222 在给定矩阵中，一条可行的滑行轨迹为 24-17-2-1。
2223 在给定矩阵中，最长的滑行轨迹为 25-24-23-...-3-2-1，沿途共经过 25 个区域。
2224 现在给你二维矩阵表示滑雪场各区域的高度，请你找出在该滑雪场中能够完成的最长滑雪
    轨，并输出其长度(可经过最大区域数)。
2225 输入格式
2226
2227 第一行包含两个整数 R 和 C。
2228

```

2229 接下来 R

2230 行，每行包含 C

2231

2232 个整数，表示完整的二维矩阵。

2233 输出格式

2234

2235 输出一个整数，表示可完成的最长滑雪长度。

2236 数据范围

2237

2238 $1 \leq R, C \leq 300, 0 \leq \text{矩阵中整数} \leq 10000$

2239

2240 样例

2241 输入样例：

2242

2243 5 5

2244 1 2 3 4 5

2245 16 17 18 19 6

2246 15 24 25 20 7

2247 14 23 22 21 8

2248 13 12 11 10 9

2249

2250 输出样例：

2251

2252 25

2253

2254

2255 题目大意题目大意

2256 给定一个矩阵，从矩阵的每个位置出发，问最长的递减的一条路径最长可以是多长。给定一个矩阵，从矩阵的每个位置出发，问最长的递减的一条路径最长可以是多长。

2257 思路思路

2258 有多条路径，可以直接搜索每一条路径。有多条路径，可以直接搜索每一条路径。

2259 但是时间太慢了，考虑优化。但是时间太慢了，考虑优化。

2260 定义数组 $f[N][N]$ 定义数组 $f[N][N]$

2261 设 $f[i][j]$ 表示从位置 (i, j) 出发可以得到的最长长度。设 $f[i][j]$ 表示从位置 (i, j) 出发可以得到的最长长度。

2262 那么考虑 f 数组对于当前求得 $\max(dp(i, j))$ 的 $dp(i, j)$ 又有什么关系。那么考虑 f 数组对于当前求得 $\max(dp(i, j))$ 的 $dp(i, j)$ 又有什么关系。

2263 若当前搜索的位置 (p, q) 是一条路径 (i, j) 的最优路径经过的一个位置。若当前搜索的位置 (p, q) 是一条路径 (i, j) 的最优路径经过的一个位置。

2264 那么其实不需要尝试四个方向，产生多余的复杂度。那么其实不需要尝试四个方向，产生多余的复

杂度。

2265 最开始把f数组初始化为-1, f[i][j] == -1表示该位置还没有被计算过, 当f[i][j] != -1时, 则不需要计算f[i][j], 这就是优化的地方。最开始把f数组初始化为-1, f[i][j] == -1表示该位置还没有被计算过, 当f[i][j] != -1时, 则不需要计算f[i][j], 这就是优化的地方。

2266 代码实现部分: 代码实现部分:

2267 第一步, 定义数组h[N][N]保存高度, f[N][N]保存每一个(i, j)的最优解, 也是记忆化数组。
第一步, 定义数组h[N][N]保存高度, f[N][N]保存每一个(i, j)的最优解, 也是记忆化数组。

2268 const int N = 3e2 + 10;

2269 int h[N][N], f[N][N], n, m;

2270 第二步, 定义函数。dp(x, y)返回以位置(i, j)为起点能延申的最长长度第二步, 定义函数。dp(x, y)返回以位置(i, j)为起点能延申的最长长度

2271 函数写什么: 函数写什么:

2272 ①判断当前位置的f[x][y]是否已被计算过, 如果计算过则不计算, 直接返回f[x][y]①判断当前位置的f[x][y]是否已被计算过, 如果计算过则不计算, 直接返回f[x][y]

2273 int& v = f[x][y]; if (~v) return v;

2274 ②否则尝试四个方向, 为了简便可以定义两个数组dx和dy控制方向②否则尝试四个方向, 为了简便可以定义两个数组dx和dy控制方向

2275 int dx[] = { -1, 0, 1, 0 }, dy[] = { 0, 1, 0, -1 }; for (int i = 0; i < 4; i++) { int xx = x + dx[i], yy = y + dy[i]; }

2276 ③判断移动后的位置是否合法(是否再场地里并且高度比h[i][j]低)③判断移动后的位置是否合法(是否再场地里并且高度比h[i][j]低)

2277 if (xx >= 1 && xx <= n && yy >= 1 && yy <= m && h[xx][yy] < h[x][y])

2278 ④如果位置合法, 那么计算它的最长路径再加上本身的一格, 取max④如果位置合法, 那么计算它的最长路径再加上本身的一格, 取max

2279 v = max(v, dp(xx, yy) + 1);

2280 ⑤返回计算的max值, 并记录再f[x][y]里, 一边下次调用所使用⑤返回计算的max值, 并记录再f[x][y]里, 一边下次调用所使用

2281 return v;

2282 第三步, 读入数据并且初始化f数组为-1第三步, 读入数据并且初始化f数组为-1

2283 cin >> n >> m;

2284 for (int i = 1; i <= n; i++)

2285 for (int j = 1; j <= m; j++)

2286 cin >> h[i][j];

2287 memset(f, -1, sizeof(f));

2288 第四步, 循环枚举起始位置(i, j)的所有可能, 最优解为dp(i, j), 那么答案就是Max(dp(i, j))第四步, 循环枚举起始位置(i, j)的所有可能, 最优解为dp(i, j), 那么答案就是Max(dp(i, j))

2289 int res = 0;

2290 for (int i = 1; i <= n; i++)

```

2291 for (int j = 1; j <= m; j++)
2292 res = max(res, dfs(i, j));
2293 cout << res;
2294 样例测试：样例测试：
2295 f数组：f数组：
2296 1234512345
2297 161718196161718196
2298 152425207152425207
2299 142322218142322218
2300 131211109131211109
2301 样例组样例组
2302 输出：输出：
2303 2525*/
2304 //AC!AC!
2305 //带注释代码：带注释代码：
2306 #include <iostream>
2307 #include <cstring>
2308 using namespace std;
2309 const int N = 310;
2310 int h[N][N], f[N][N], n, m;
2311 //定义数组h[N][N]存储高度，f[N][N]表示最长长度
2312 int dx[] = { -1 , 0 , 1 , 0 }, dy[] = { 0 , 1 , 0 , -1 };
2313 //定义数组dx,dy控制四个方向
2314 int dfs(int x, int y) {
2315     int& v = f[x][y]; //小技巧：用取地址符的变量v代替f[x][y], v = t等价于f
        [x][y] = t
2316     if (~v) //如果已经被搜过了，则不需要搜索了
2317         return v; //直接return答案
2318     v = 1; //f[x][y]最少为1，因为极端情况h (x, y)的四个方向都
        比h (x, y)高，那么长度也是(i, j)这一格
2319     for (int i = 0; i < 4; i++) { //分别尝试四个方向
2320         int to_x = dx[i] + x, to_y = dy[i] + y; //通过调用dx和d
            y获取x和y当前的位置
2321         if (to_x >= 1 && to_x <= n && to_y >= 1 && to_y <= m && h[to_
            x][to_y] < h[x][y]) //判断位置是否合法
2322             v = max(v, dfs(to_x, to_y) + 1); //计算最长长度
2323     }
2324     return v; //返回
2325 }
2326 int main() {

```

```

2327     cin >> n >> m;
2328     for (int i = 1; i <= n; i++)
2329         for (int j = 1; j <= m; j++)
2330             cin >> h[i][j];
2331     //读入数据
2332     memset(f, -1, sizeof(f));    //初始化f数组
2333     int res = 0;
2334     for (int i = 1; i <= n; i++)
2335         for (int j = 1; j <= m; j++)
2336             res = max(res, dfs(i, j));    //取Max (dp (i, j))
2337     cout << res;    //输出
2338 }
2339 //无注释代码：无注释代码：
2340 #include <iostream>
2341 #include <cstring>
2342 using namespace std;
2343 const int N = 310;
2344 int h[N][N], f[N][N], n, m;
2345 int dx[] = { -1 , 0 , 1 , 0 }, dy[] = { 0 , 1 , 0 , -1 };
2346 int dfs(int x, int y) {
2347     int& v = f[x][y];
2348     if (~v)
2349         return v;
2350     v = 1;
2351     for (int i = 0; i < 4; i++) {
2352         int to_x = dx[i] + x, to_y = dy[i] + y;
2353         if (to_x >= 1 && to_x <= n && to_y >= 1 && to_y <= m && h[to_
x][to_y] < h[x][y])
2354             v = max(v, dfs(to_x, to_y) + 1);
2355     }
2356     return v;
2357 }
2358 int main() {
2359     cin >> n >> m;
2360     for (int i = 1; i <= n; i++)
2361         for (int j = 1; j <= m; j++)
2362             cin >> h[i][j];
2363     memset(f, -1, sizeof(f));
2364     int res = 0;
2365     for (int i = 1; i <= n; i++)

```

```

2366         for (int j = 1; j <= m; j++)
2367             res = max(res, dfs(i, j));
2368     cout << res;
2369 }
2370
2371 //表达式求值
2372 #include<iostream>
2373 using namespace std;
2374 #include<string>
2375 #include<algorithm>
2376 #include<stack>
2377 #include<unordered_map>
2378
2379 const int maxn = 100020;
2380
2381 stack<int>num;           //数字栈
2382 stack<char>op;          //运算符栈
2383
2384 void eval()
2385 {
2386     auto b = num.top();           //考虑运算顺序，先取出来的是第二个运算数
2387     num.pop();
2388     auto a = num.top();
2389     num.pop();
2390
2391     auto c = op.top();
2392     op.pop();
2393     int x = 0;
2394     if (c == '+') x = a + b;
2395     else if (c == '-') x = a - b;
2396     else if (c == '*') x = a * b;
2397     else x = a / b;
2398     num.push(x);
2399 }
2400 int main()
2401 {
2402     unordered_map<char, int>pr{ {'+', 1}, {'-', 1}, {'*', 2}, {'/', 2} };
2403     //定义一个哈希表，存储优先级
2404
2405     string str;

```

```

2405     cin >> str;
2406
2407     for (int i = 0; i < str.size(); i++)
2408     {
2409         auto c = str[i];
2410         if (isdigit(c))
2411         {
2412             int x = c - '0', j = i + 1;
2413             while (j < str.size() && isdigit(str[j]))
2414             {
2415                 x = x * 10 + str[j++] - '0';           //这里的j是要自增
                的
2416             }
2417             i = j - 1;                                   //别忘了更新i
2418             num.push(x);
2419         } //取出数字
2420         else if (c == '(') //如果是左括号，入栈即可
2421         {
2422             op.push(c);
2423         }
2424         else if (c == ')') //如果是右括号，就意味着我们应该把目前栈里面的运
                算符都计算完
2425             //并且我们知道，遇到优先级比当前栈顶优先级低的运算符，我们就会先把栈顶
                运算符操作完
2426             //那么，还留在栈里面的操作符都是优先级递增的，所以要从右往左算
2427         {
2428             while (op.top() != '(') eval();           //一直计算完整个括号
2429             op.pop();                                   //弹出左括号
2430         }
2431         else //一般情况
2432         {
2433             while (op.size() && pr[op.top()] >= pr[c]) //符号栈没空，并且
                栈顶的运算符优先级大于当前运算符
2434             {
2435                 eval(); //操作栈顶的运算符
2436                 /*
2437                 while (op.size() && op.top() != '(' && pr[op.top()] >=
                pr[c]) eval();
2438             这里是while而不是if的原因
2439

```

```

2440 假如表达式为2-3*2+1,直到2-3*2都没有运算,遇到+的时候,先运算了3*2,还剩2-6
2441 如果是if的话,就会把+和1入栈,栈里成了2-6+1,会先运算栈顶的6+1,再运算2-7,最终运算
      错误。
2442 而如果是while,则会先运算2-6,再运算-4+1,运算正确。
2443
2444 思想就是当前树的子树都运算完了,再运算当前树的根节点
2445         */
2446     }
2447     op.push(c); //新的运算符压入栈
2448
2449 }
2450 }
2451
2452 while (op.size()) //把最后剩下的运算符操作完
2453 {
2454     eval();
2455 }
2456 cout << num.top() << endl;
2457
2458 return 0;
2459 }
2460
2461
2462 //滑动窗口
2463
2464 #include<iostream>
2465 using namespace std;
2466 const int maxn = 1000020;
2467 int q[maxn], hh = 0, tt = -1; //用队列维护窗口
2468 int a[maxn];
2469 int main()
2470 {
2471     int n, k;
2472     cin >> n >> k;
2473     for (int i = 0; i < n; i++)
2474     {
2475         cin >> a[i];
2476     }
2477     //最小值
2478     for (int i = 0; i < n; i++)

```

//队列存的是下标

```

2479     {
2480         //每一次循环，窗口都更新为以【i】位置结尾的窗口，但是我们的队列中
2481         if (q[hh] <= i - k && hh <= tt) //判断队头存的下
标是否已经滑出了窗口以及队列是不是已经空了。
2482     {
2483         hh++;
2484     }
2485     while (hh <= tt && a[q[tt]] >= a[i]) //队列不空，且队尾
所表示的下标处元素比a[i]大，那就说明，这个队尾位置的元素不会被用到，所以，删掉
2486     {
2487         tt--;
2488     }
2489     q[++tt] = i;
2490     //如果队列被删光，i会存到队头（队头队尾重合），只有一个元素
2491     //如果队列没有被删光，那就说明，队头指的元素就是还在目前窗口中，并且最小的
元素（也小于我当前循环到的元素）。
2492     if (i >= k - 1) //符合窗口长度要求才
输出。
2493     {
2494         cout << a[q[hh]] << " ";
2495     }
2496 }
2497 cout << endl;
2498
2499 //最大值：
2500 //重新初始化队列
2501 hh = 0, tt = -1;
2502 for (int i = 0; i < n; i++) //队列存的是下标
2503 {
2504     //每一次循环，窗口都更新为以【i】位置结尾的窗口，但是我们的队列中
2505     if (q[hh] <= i - k && hh <= tt) //判断队头存的下
标是否已经滑出了窗口以及队列是不是已经空了。
2506     {
2507         hh++;
2508     }
2509     while (hh <= tt && a[q[tt]] <= a[i]) //队列不空，且队尾
所表示的下标处元素比a[i]小，那就说明，这个队尾位置的元素不会被用到，所以，删掉
2510     {
2511         tt--;
2512     }

```

```

2513         q[++tt] = i;
2514         //如果队列被删光，i会存到队头（队头队尾重合），只有一个元素
2515         //如果队列没有被删光，那就说明，队头指的元素就是还在目前窗口中，并且最大的
        元素（也大于我当前循环到的元素）。
2516         if (i >= k - 1)                                     //符合窗口长度要求才
        输出。
2517     {
2518         cout << a[q[hh]] << " ";
2519     }
2520 }
2521
2522 return 0;
2523
2524 }
2525
2526 //kmp
2527 #include<iostream>
2528 using namespace std;
2529 const int N = 100020, M = 1000020;
2530 char p[N], s[M];
2531 int n, m;
2532 int ne[N];
2533 int main()
2534 {
2535     //整体的思想就是，匹配不成功之后需要移动模板串，而由于已经匹配了那么多了，所以可
        以根据已经匹配的这部分
2536     //的，最大的相同前后缀（前缀不包含最后一个，后缀不包含第一个），从而可以直到最少
        移动多少，就可以再次匹配到i这个位置
2537     //因为，一次匹配不成功的话，你移动之后再匹配还是要匹配到这个之前没成功的地方的
2538     //也就是说，你移动之后想要继续匹配的话，一定是p[1~(新的j)]=p[(n-新的j+1)~n]
        相等，这就是前后缀，那么这个东西，你可以算出来最大时多少，那么就知道了最小可以移动多少
2539     cin >> n >> p + 1 >> m >> s + 1;
2540     for (int i = 2, j = 0; i <= n; i++)
2541     {
2542         while (j && p[i] != p[j + 1])
2543         {
2544             j = ne[j];
2545         }
2546         if (p[i] == p[j + 1])
2547     {

```

```

2548         j++;
2549     }
2550     ne[i] = j;
2551 }
2552
2553     for (int i = 1, j = 0; i <= m; i++)
2554     {
2555         while (j && s[i] != p[j + 1])
2556         {
2557             j = ne[j];
2558         }
2559         if (s[i] == p[j + 1])
2560         {
2561             j++;
2562         }
2563         if (j == n)
2564         {
2565             cout << i - n << " ";
2566             j = ne[j];
2567         }
2568     }
2569 }
2570 }
2571
2572 //最大异或对
2573
2574 #include<iostream>
2575 using namespace std;
2576
2577 const int N = 100020, M = 31 * N;
2578
2579 int n;
2580 int a[N];
2581 int son[M][2], idx;
2582
2583 //存储输入的数
2584 //最多maxn个数，每个数存成31
2585 //位，那么也就是说，会用到31*N个位置
2586 void input(int x)
2587 {
2588     int p = 0;

```

```

2587
2588     for (int i = 30; i >= 0; i--)
2589     {
2590         int u = (x >> i) & 1;           //取出来的是第i+1位 (二
进制中) 指的是权值为 $2^i$ 的那一位
2591
2592         if (!son[p][u])
2593         {
2594             son[p][u] = ++idx;
2595         }
2596         p = son[p][u];
2597     }
2598
2599 }
2600
2601 int query(int x)
2602 {
2603     int p = 0, num = 0;                //num用来存储query的结果
2604     for (int i = 30; i >= 0; i--)      //在存储的时候保证了一
定会存成31位的, 所以没有必要担心循环停止的问题
2605     {
2606         int u = (x >> i) & 1;
2607
2608         if (son[p][!u] != 0)
2609         {
2610             p = son[p][!u];           //我们想要的是与这一位相反的, 好异或成1
2611             num = num * 2 + !u;
2612         }
2613         //没有的话, 只能将就一下了
2614         else
2615         {
2616             p = son[p][u];
2617             num = num * 2 + u;
2618         }
2619     }
2620     return num;
2621 }
2622
2623 int main()
2624 {

```

```

2625     cin >> n;
2626     for (int i = 0; i < n; i++)
2627     {
2628         cin >> a[i];
2629     }
2630
2631     int res = 0; //存储结果
2632
2633     for (int i = 0; i < n; i++) //n别写成a[i]了
2634     {
2635         input(a[i]); //插入a[i]这个数
2636
2637         int t = query(a[i]); //求在目前的数中与a[i]异或
        最大的数，因为异或是可交换的，所以
2638         //每个数只需要与之前的数遍历就可以保证没有遗漏
2639         res = max(res, a[i] ^ t);
2640
2641     }
2642
2643     cout << res;
2644
2645     return 0;
2646 }
2647
2648
2649 //联通块中点的数量
2650
2651 /*
2652     给定一个包含 n 个点（编号为 1~n）的无向图，初始时图中没有边。
2653
2654 现在要进行 m 个操作，操作共有三种：
2655
2656 C a b，在点 a 和点 b 之间连一条边，a 和 b 可能相等；
2657 Q1 a b，询问点 a 和点 b 是否在同一个连通块中，a 和 b 可能相等；
2658 Q2 a，询问点 a 所在连通块中点的数量；
2659 输入格式
2660 第一行输入整数 n 和 m。
2661
2662 接下来 m 行，每行包含一个操作指令，指令为 C a b，Q1 a b 或 Q2 a 中的一种。
2663

```

```
2664 输出格式
2665 对于每个询问指令 Q1 a b，如果 a 和 b 在同一个连通块中，则输出 Yes，否则输出 No。
2666
2667 对于每个询问指令 Q2 a，输出一个整数表示点 a 所在连通块中点的数量
2668
2669 每个结果占一行。
2670
2671 数据范围
2672  $1 \leq n, m \leq 105$ 
2673 输入样例：
2674 5 5
2675 C 1 2
2676 Q1 1 2
2677 Q2 1
2678 C 2 5
2679 Q2 5
2680 输出样例：
2681 Yes
2682 2
2683 3
2684
2685 */
2686
2687 #include<iostream>
2688 using namespace std;
2689 #include<string>
2690 const int maxn = 100020;
2691 int p[maxn], cnt[maxn];
2692 int find(int x)
2693 {
2694     if (p[x] != x) p[x] = find(p[x]); //查找祖宗节点，并且实现了
    路径压缩，即，从祖宗节点往回递归的过程中，会把所有子孙节点都定位到祖宗这
2695
2696
2697     return p[x];
2698 }
2699
2700
2701 int main()
2702 {
```

```

2703     int n, m;
2704     cin >> n >> m;
2705     for (int i = 1; i <= n; i++)
2706     {
2707         p[i] = i;
2708         cnt[i] = 1;          //初始化为1;
2709     }
2710     for (int i = 1; i <= m; i++)
2711     {
2712         string op;
2713         cin >> op;
2714         if (op == "C")
2715         {
2716             int a, b;
2717             cin >> a >> b;
2718             if (find(a) == find(b)) continue;
2719             cnt[find(b)] += cnt[find(a)];          //两个点连边，相当于只要操作
                                                    祖宗节点就可以了，让祖宗节点的cnt有意义即可，
2720
2721             p[find(a)] = find(b);          //注意这里要调用的find函数，因为调用的
                                                    同时就能够做到压缩路径
2722         }
2723         else if (op == "Q1")
2724         {
2725             int a, b;
2726             cin >> a >> b;
2727             if (find(a) == find(b)) cout << "Yes" << endl;
2728             else cout << "No" << endl;
2729         }
2730         else
2731         {
2732             int a;
2733             cin >> a;
2734             cout << cnt[find(a)] << endl;
2735         }
2736     }
2737
2738     return 0;
2739 }
2740

```

2741
2742 //食物链
2743 /*
2744 动物王国中有三类动物 A,B,C，这三类动物的食物链构成了有趣的环形。
2745
2746 A 吃 B，B 吃 C，C 吃 A。
2747
2748 现有 N 个动物，以 1~N 编号。
2749
2750 每个动物都是 A,B,C 中的一种，但是我们并不知道它到底是哪一种。
2751
2752 有人用两种说法对这 N 个动物所构成的食物链关系进行描述：
2753
2754 第一种说法是 1 X Y，表示 X 和 Y 是同类。
2755
2756 第二种说法是 2 X Y，表示 X 吃 Y。
2757
2758 此人对 N 个动物，用上述两种说法，一句接一句地说出 K 句话，这 K 句话有的是真的，有的是假的。
2759
2760 当一句话满足下列三条之一时，这句话就是假话，否则就是真话。
2761
2762 当前的话与前面的某些真的话冲突，就是假话；
2763 当前的话中 X 或 Y 比 N 大，就是假话；
2764 当前的话表示 X 吃 X，就是假话。
2765 你的任务是根据给定的 N 和 K 句话，输出假话的总数。
2766
2767 输入格式
2768 第一行是两个整数 N 和 K，以一个空格分隔。
2769
2770 以下 K 行每行是三个正整数 D,X,Y，两数之间用一个空格隔开，其中 D 表示说法的种类。
2771
2772 若 D=1，则表示 X 和 Y 是同类。
2773
2774 若 D=2，则表示 X 吃 Y。
2775
2776 输出格式
2777 只有一个整数，表示假话的数目。
2778
2779 数据范围

```

2780 1≤N≤50000,
2781 0≤K≤100000
2782 输入样例：
2783 100 7
2784 1 101 1
2785 2 1 2
2786 2 2 3
2787 2 3 3
2788 1 1 3
2789 2 3 1
2790 1 5 5
2791 输出样例：
2792 3
2793 难度：中等
2794 时/空限制：1s / 64MB
2795 总通过数：49259
2796 总尝试数：106977
2797 来源：《算法竞赛进阶指南》，模板题，NOI2001，POJ1182，kuangbin专题
2798 算法标签
2799
2800 */
2801
2802 #include<iostream>
2803
2804 using namespace std;
2805
2806 const int maxn = 50010;
2807
2808 int p[maxn], d[maxn]; //p[]存储父节点，d表示此时到自己的
                        根节点的距离，距离初始化为0
2809
2810 int find(int x)
2811 {
2812     if (p[x] != x)
2813     {
2814         int t = find(p[x]); //因为我们要先保留p[x]此时的值，
                        用与更新d，所以先用t存储
2815         d[x] += d[p[x]]; //根节点改变，因此距离等于之前的d
                        加上父节点的d
2816

```

```

2817         p[x] = t;                                //将这个节点指向新的根节点
2818     }
2819     return p[x];
2820 }
2821
2822
2823 int main()
2824 {
2825     int n, m;
2826     cin >> n >> m;
2827
2828     //初始化
2829     for (int i = 1; i <= n; i++)
2830     {
2831         p[i] = i;
2832     }
2833
2834     int res = 0;                                    //存储答案，即假话的数量
2835
2836     while (m--)
2837     {
2838         int t, x, y;
2839         cin >> t >> x >> y;
2840
2841         if (x > n || y > n) res++;
2842         else
2843         {
2844             int px = find(x), py = find(y);        //把x和y的根节点找出来
2845
2846             if (t == 1)                             //这句话说明x和y是同类
2847             {
2848                 if (px == py)                       //说明x, y是在一个集合里
2849                 {
2850                     if ((d[x] - d[y]) % 3 != 0) res++; //由分析可得，在同
一个集合里面，以三为周期循环，%3同余数是同一类
2851
2852                     /*
2853                     这里不可以用 (d[x] % 3) != (d[y] % 3) 作为判定条件
2854                     因为我们的条件实际上是%3意义上相等则同类，而不是模的结果相
同，因为要考虑到我们后面为了把两个结合组合在一起，会导致路径为负数

```

```

2855          C++的负数取模还是负数，但实际上 $-1\%3=-1$ 与 $2\%3=2$ 是同一个意
            义，因为 $-1$ 可以看成 $-1*3+2$ ， $2$ 可以是 $0*3+2$ 
2856          */
2857
2858      }
2859
2860      else          //此时还不是一个集合里的
2861      {
2862          p[px] = py;          //把x的根节点只想y的根节点
2863          d[px] = d[y] - d[x];          //把px到py的边设置为d[y]-d
            [x]
2864          //因为x,y应该是同类，那么x,y到根节点的距离应该是模3同余的，
2865          //x到根节点的距离应该为(d[x]+d[px])，y到根节点的距离是d
            [y]
2866          //那么 (d[x]+d[px]-d[y]) %3==0;    //同余的性质
2867          //也就是说d[px]可以取：d[y]-d[x];
2868      }
2869  }
2870  else          //这句话是第2种说法，表示的是 x吃y
2871  {
2872      if (px == py)          //说明x,y是在一个集合里
2873      {          //x吃y,那么模3的意义下，d[x]%3比
            d[y]%3多1
2874          if ((d[x] - d[y] - 1) % 3 != 0) res++;
            //不满足上述，说明是假话
2875      }
2876      else
2877      {
2878          p[px] = py;          //把x的根节点只想y的根节点
2879          //x吃y,那么模3的意义下，d[x]%3比d[y]%3多1
2880          //新的d[x]为d[x]+d[px]
2881          //那么，(d[x]+d[px]-d[y]-1) %3==0，所以，d[px]=d[y]+1
            -d[x]
2882
2883          d[px] = d[y] + 1 - d[x];
2884      }
2885  }
2886  }
2887  }
2888

```

```
2889
2890     cout << res << endl;
2891
2892     return 0;
2893 }
2894
2895 /*字符串哈希：
2896
2897 给定一个长度为 n 的字符串，再给定 m 个询问，每个询问包含四个整数 l1, r1, l2, r2，
    请你判断[l1, r1] 和[l2, r2] 这两个区间所包含的字符串子串是否完全相同。
2898
2899 字符串中只包含大小写英文字母和数字。
2900
2901 输入格式
2902 第一行包含整数 n 和 m，表示字符串长度和询问次数。
2903
2904 第二行包含一个长度为 n 的字符串，字符串中只包含大小写英文字母和数字。
2905
2906 接下来 m 行，每行包含四个整数 l1, r1, l2, r2，表示一次询问所涉及的两个区间。
2907
2908 注意，字符串的位置从 1 开始编号。
2909
2910 输出格式
2911 对于每个询问输出一个结果，如果两个字符串子串完全相同则输出 Yes，否则输出 No。
2912
2913 每个结果占一行。
2914
2915 数据范围
2916  $1 \leq n, m \leq 105$ 
2917 输入样例：
2918 8 3
2919 aabbaabb
2920 1 3 5 7
2921 1 3 6 8
2922 1 2 1 2
2923 输出样例：
2924 Yes
2925 No
2926 Yes
2927 难度：简单
```

```

2928 时 / 空限制：1s / 64MB
2929 总通过数：53618
2930 总尝试数：81230
2931 来源：模板题
2932 算法标签
2933 */
2934
2935 #include<iostream>
2936
2937 using namespace std;
2938
2939 const int maxn = 1000020;
2940
2941 typedef unsigned long long ULL; //无符号整数，0~2^64-1,利用它的溢出相当于对2^
    64
2942
2943 int n, m, P = 131; //一般P取131或者13331，然后对2^64取模，基
    本上不会冲突
2944
2945 char str[maxn];
2946
2947 ULL h[maxn], p[maxn];
2948 ULL get(int l, int r)
2949 {
2950     return h[r] - h[l - 1] * p[r - l + 1]; //任意段hash值的求法
2951
2952 }
2953 int main()
2954 {
2955     cin >> n >> m >> str + 1;
2956     p[0] = 1;
2957     for (int i = 1; i <= n; i++)
2958     {
2959         p[i] = p[i - 1] * P; //p[i]表示的是p[i]的i次方
2960         h[i] = h[i - 1] * P + str[i] + 1; //h[i]表示的是前i个数的hash值，
            加上str[i],是为了做到区分每一个字母
2961     } //+1保证不会映射到0，从而导致无论多少位都是0
2962
2963     for (int i = 0; i < m; i++)

```

```

2964     {
2965         int l1, r1, l2, r2;
2966         cin >> l1 >> r1 >> l2 >> r2;
2967         if (get(l1, r1) == get(l2, r2))
2968         {
2969             cout << "Yes" << endl;
2970         }
2971         else
2972             cout << "No" << endl;
2973     }
2974 }
2975
2976     return 0;
2977 }

```

2978 //哈夫曼

2979 算法

2980 (贪心, 哈夫曼树, 堆, 优先队列) $O(n\log n)O(n\log n)$

2981 经典哈夫曼树的模型, 每次合并重量最小的两堆果子即可。

2982

2983 时间复杂度

2984 使用小根堆维护所有果子, 每次弹出堆顶的两堆果子, 并将其合并, 合并之后将两堆重量之和再次插入小根堆中。

2985

2986 每次操作会将果子的堆数减一, 一共操作 $n-1$ 次即可将所有果子合并成1堆。每次操作涉及到2次堆的删除操作和1次堆的插入操作, 计算量是 $O(\log n)O(\log n)$ 。因此总时间复杂度是 $O(n\log n)O(n\log n)$ 。

2987

2988 C++ 代码

2989 #include <iostream>

2990 #include <algorithm>

2991 #include <queue>

2992

2993 using namespace std;

2994

2995 int main()

2996 {

2997 int n;

2998 scanf("%d", &n);

3000

```
3001     priority_queue<int, vector<int>, greater<int>> heap;
3002     while (n--)
3003     {
3004         int x;
3005         scanf ("%d", &x);
3006         heap.push(x);
3007     }
3008
3009     int res = 0;
3010     while (heap.size() > 1)
3011     {
3012         int a = heap.top(); heap.pop();
3013         int b = heap.top(); heap.pop();
3014         res += a + b;
3015         heap.push(a + b);
3016     }
3017
3018     printf ("%d\n", res);
3019     return 0;
3020 }
```

3021 题目描述

3022 在一个果园里，达达已经将所有的果子打了下来，而且按果子的不同种类分成了不同的堆。

3023

3024 达达决定把所有的果子合成一堆。

3025

3026 每一次合并，达达可以把两堆果子合并到一起，消耗的体力等于两堆果子的重量之和。

3027

3028 可以看出，所有的果子经过 $n - 1$ 次合并之后，就只剩下一堆了。

3029

3030 达达在合并果子时总共消耗的体力等于每次合并所耗体力之和。

3031

3032 因为还要花大力气把这些果子搬回家，所以达达在合并果子时要尽可能地节省体力。

3033

3034 假定每个果子重量都为1，并且已知果子的种类数和每种果子的数目，你的任务是设计出合并的次序方案，使达达耗费的体力最少，并输出这个最小的体力耗费值。

3035

3036 例如有3种果子，数目依次为1，2，9。

3037

3038 可以先将1、2堆合并，新堆数目为3，耗费体力为3。

3039

3040 接着，将新堆与原先的第三堆合并，又得到新的堆，数目为12，耗费体力为12。

3041

3042 所以达达总共耗费体力 = 3 + 12 = 15。

3043

3044 可以证明15为最小的体力耗费值。

3045

3046 输入格式

3047 输入包括两行，第一行是一个整数n，表示果子的种类数。

3048

3049 第二行包含n个整数，用空格分隔，第i个整数ai是第i种果子的数目。

3050

3051 输出格式

3052 输出包括一行，这一行只包含一个整数，也就是最小的体力耗费值。

3053

3054 输入数据保证这个值小于231。

3055

3056 数据范围

3057 $1 \leq n \leq 100001$ $1 \leq n \leq 10000$

3058 $1 \leq a_i \leq 200001$ $1 \leq a_i \leq 20000$

3059 样例

3060 输入样例：

3061 3

3062 1 2 9

3063 输出样例：

3064 15

3065 哈夫曼树 + 优先队列

3066 这道题目是哈夫曼树的典型模板，也就是每次选择最小的两个果堆，然后将他们合并起来，再次压入堆中。

3067 C++ 代码

3068 `#include <bits/stdc++.h>`

3069 `using namespace std;`

3070 `const int N = 10100;`

3071 `int n, m, i, j, a[N], ans;`

3072 `priority_queue<int, vector<int>, greater<int> > p;`

3073 `int main()`

3074 `{`

3075 `cin >> n;`

3076 `for (int i = 1; i <= n; i++)`

3077 `cin >> a[i], p.push(a[i]);`

3078 `for (int i = 1; i < n; i++)`

```
3079     {
3080         int x, y;
3081         x = p.top();
3082         p.pop();
3083         y = p.top();
3084         p.pop();
3085         ans += x + y;
3086         p.push(x + y);
3087     }
3088     cout << ans;
3089     return 0;
3090 }
```

3091

3092 书架上有若干本书排成一排。

3093

3094 每本书要么是大型书（用 L 表示），要么是中型书（用 M 表示），要么是小型书（用 S 表示）。

3095

3096 我们希望所有书能够从大到小有序排列，也就是说，所有大型书都在左侧，所有中型书都在中间，所有小型书都在右侧。

3097

3098 为此，你可以进行任意次交换操作，每次可以任选两本书并交换它们的位置。

3099

3100 请你计算，为了让所有书按要求有序排列，至少需要进行多少次交换操作。

3101

3102 输入格式

3103 共一行，包含一个由 L 、 M 、 S 构成的字符串，表示初始时每个位置上的书的类型。

3104

3105 输出格式

3106 一个整数，表示所需要的最少交换操作次数。

3107

3108 数据范围

3109 输入字符串的长度范围 $[1, 5 \times 10^5]$ 。

3110

3111 输入样例1：

3112 LMMMS

3113 输出样例1：

3114 0

3115 样例1解释

3116 无需任何操作，初始排列已经符合要求。

```
3117
3118 输入样例2：
3119 LLSLM
3120 输出样例2：
3121 2
3122 样例2解释
3123 一种最佳方案如下：
3124
3125 第一步，交换 S 和 M，序列变为 LLMLS。
3126 第二步，交换 M 和它右边的 L，序列变为 LLLMS。
3127
3128 //环图做法
3129
3130 #include<iostream>
3131 using namespace std;
3132 #include<cstring>
3133 #include<algorithm>
3134 #include<string>
3135 const int N = 500020;
3136 int a[N], b[N];                //a表示原始序列，b表示目标序列，建图：a[i]
    指向b[i]
3137
3138 int s[3];                      //L=0,M=1,S=2
3139
3140 string str;
3141 int main()
3142 {
3143     cin >> str;
3144     int n = str.size();
3145
3146
3147     for (int i = 0; i < str.size(); i++)
3148     {
3149         if (str[i] == 'L')
3150         {
3151             a[i] = 0;
3152         }
3153         else if (str[i] == 'M')
3154         {
3155             a[i] = 1;
```

```

3156     }
3157     else if (str[i] == 'S')
3158     {
3159         a[i] = 2;
3160
3161     }
3162     s[a[i]]++;
3163 }
3164
3165     for (int i = 0, k = 0; i < 3; i++) //把所有的0,
1,2按照要求放到b序列当中
3166     {
3167         for (int j = 1; j <= s[i]; j++, k++)
3168         {
3169             b[k] = i;
3170         }
3171     }
3172     int e[3][3] = { 0 }; //统计每条边的
数量
3173
3174     for (int i = 0; i < n; i++)
3175     {
3176         e[a[i]][b[i]]++;
3177     }
3178
3179     int m = 0; //求自环的数量
3180
3181     for (int i = 0; i < 3; i++)
3182     {
3183         m += e[i][i];
3184     }
3185
3186     //求小环的数量
3187
3188
3189     for (int i = 0; i < 3; i++)
3190     {
3191         for (int j = i + 1; j < 3; j++)
3192         {
3193             int t = min(e[i][j], e[j][i]); //i到j和j到i的边数,

```

小的那个就是环数

```
3194
3195         m += t;
3196
3197         e[i][j] -= t;
3198         e[j][i] -= t;
3199     }
3200 }
3201
3202 m += e[0][1] + e[1][0];
3203
3204
3205 cout << n - m << endl;
3206
3207 return 0;
3208
3209 }
```

3210 线段树

3211 //传入节点编号，用子节点信息来算父节点信息

3212 push up (int u)

3213

3214 //将一段区间初始化为一颗线段树

3215 build()

3216

3217 //修改操作，修改某一个点或者某一个区间（懒标记）

3218 modify()

3219

3220 //查询某一段区间的信息

3221 query()

3222 定义：

3223 线段树是一颗满二叉树；以一棵长度为10的序列为例：

3224

3225

3226

3227 对于图中的段从上到下，从左到右依次编号为1，2，3

3228

3229 因此某一段 u 的左儿子是 $u \ll 1$ ，右儿子是 $u \ll 1 | 1$ 。

3230

3231 线段树的结构

3232 //一般使用结构体来存储线段树，空间大小开四倍

```

3233 struct Node{
3234     int l,r;    //维护的区间
3235     int v;      //维护的信息...
3236 } tree[N*4];
3237 线段树的建树：
3238 //build
3239 void build(int u,int l,int r){ //构建节点u，其维护的是区间[l,r]
3240     tr[u]={l,r};
3241     if(l==r) return ; //已经是叶子节点
3242     int mid=l+r>>1;
3243     build(u<<1,l,mid),build(u<<1|1,mid+1,r);
3244 }
3245 push_up操作
3246 //push_up操作,用子节点信息来更新父节点信息,以维护最大值为例
3247 void push_up(int u){
3248     tree[u].v=max(tree[u<<1].v,tree[u<<1|1].v);
3249 }
3250 查询操作
3251 //query操作，用来查询某一段区间内的信息,以最大值为例
3252 int query(int u,int l,int r){ //从u节点开始查询[l,r]区间内的某一信息
3253     if(tree[u].l>=l&&tree[u].r<=r) return tree[u].v; //说明这一段的信息已
    经被完全包含，因此不需要继续向下递归，直接返回即可
3254     int res=0;
3255     //否则需要判断该递归那一边
3256     int mid=tree[u].l+tree[u].r >> 1;
3257     if(l<=mid) res=max(res,query(u<<1,l,r)); //递归左边并更新信息
3258     if(mid<r) res=max(res,query(u<<1|1,l,r)); //递归右边并更新信息,切记是
    mid<r，无等号
3259     return res;
3260 }
3261 修改操作
3262 //modify操作，用来修改某一叶子节点并更新其所有父节点
3263 void modify(int u,int x,int v){ //从u节点开始递归查找，将编号为x的节点的值修
    改为v
3264     if(tree[u].l==x&&tree[u].r==x) tree[u].v=v;
3265     else{
3266         int mid=tree[u].l+tree[u].r>>1;
3267         if(x<=mid) modify(u<<1,x,v);
3268         else modify(u<<1|1,x,v);
3269         push_up(u);

```

```

3270     }
3271 }
3272 本题代码：
3273 #include<bits/stdc++.h>
3274 using namespace std;
3275 #define N 200010
3276
3277 int m,p;
3278
3279 struct Node{
3280     int l,r;
3281     int v;
3282 }tr[N*4];
3283 //用子节点信息来更新父节点
3284 void push_up(int u){
3285     tr[u].v=max(tr[u<<1].v,tr[u<<1|1].v);
3286 }
3287 //建树
3288 void build(int u,int l,int r){
3289     tr[u]={l,r};
3290     if(l==r) return ;
3291     int mid=(l+r)>>1;
3292     build(u<<1,l,mid),build(u<<1|1,mid+1,r);
3293 }
3294 //查询操作
3295 int query(int u,int l,int r){
3296     if(tr[u].l>=l&&tr[u].r<=r) return tr[u].v;
3297
3298     int mid=tr[u].l+tr[u].r>>1;
3299     int v=0;
3300     if(mid>=l) v=max(v,query(u<<1,l,r));
3301     if(mid<r) v=max(v,query(u<<1|1,l,r));
3302
3303     return v;
3304 }
3305 //修改操作
3306 void modify(int u,int x,int v){ //将点x修改成v
3307     if(tr[u].l==x&&tr[u].r==x) tr[u].v=v;
3308     else{
3309         int mid=tr[u].l+tr[u].r >> 1;

```

```
3310         if(x<=mid) modify(u<<1,x,v);
3311         else modify(u<<1|1,x,v);
3312         push_up(u);
3313     }
3314 }
3315
3316 int main() {
3317     scanf("%d%d",&m,&p);
3318     //建树
3319     build(1,1,m);
3320
3321     int n=0,last=0;
3322     for(int i=1;i<=m;i++){
3323         char op[2];
3324         int l;
3325         scanf("%s%d",op,&l);
3326         if(op[0]=='Q'){
3327             last=query(1,n-l+1,n);
3328             printf("%d\n",last);
3329         }
3330         else{
3331             int v=(l+last)%p;
3332             modify(1,++n,v);
3333         }
3334     }
3335     return 0;
3336 }
3337
```