

cldelve的算法笔记



算法目录

1	/*
2	=====知识点目录 =====
3	
4	一、基础工具与宏定义
5	- 调试与输入输出宏
6	- 循环控制宏
7	- 内存与容器操作宏
8	- 数据类型别名
9	- 基础主函数示例
10	
11	二、前缀和与差分
12	- 前缀和
13	- 差分
14	
15	三、查找算法
16	- 二分查找
17	
18	四、排序算法
19	- 归并排序（含逆序对统计）
20	- 计数排序
21	- 快速排序
22	- 快速幂
23	
24	五、动态规划
25	- 最长不下降子序列（LIS）
26	- 最长公共子序列（LCS）
27	
28	六、数据结构
29	- 树状数组（BIT）
30	- 单点修改 + 区间查询
31	- 区间修改 + 单点查询
32	- 区间修改 + 区间查询
33	- 线段树
34	- 单点修改 + 区间查询
35	- 区间修改 + 区间查询（带延迟标记）
36	
37	七、图论
38	- 图的存储（邻接表）
39	- 最短路算法
40	- BFS最短路
41	- Floyd算法
42	- Dijkstra算法
43	- SPFA算法
44	- 最小生成树（Kruskal算法）
45	
46	八、数论
47	- 线性筛（欧拉筛）
48	- 质因数分解
49	- 中国剩余定理（CRT）
50	- 最大公约数（GCD）
51	- 欧拉函数（批量计算/单个计算）

```
52 - 矩阵乘法与快速幂
53 - 组合数计算
54 - Lucas定理
55 - 预处理阶乘+逆元
56
57 九、计算几何
58 - 基础元素定义
59 - 点 (Point) 结构体及运算 (加减、点积、叉积、距离等)
60 - 线段 (Seg) 结构体及运算 (方向向量、长度、交点判断等)
61 - 圆 (Cir) 结构体及运算 (圆心、半径、圆上点计算等)
62 - 点与基本图形关系
63 - 点到点距离 (含平方距离, 不开方优化)
64 - 点到直线 / 线段的距离计算
65 - 点到直线的垂足 (投影点) 计算
66 - 点是否在线段上的判断 (含端点)
67 - 线段与线段关系
68 - 两线段是否相交的判断 (含端点重合、点在线段上)
69 - 两线段所在直线的交点计算
70 - 圆相关计算
71 - 两圆位置关系判断 (返回公共切线数量: 内含 / 内切 / 相交 / 外切 / 外离)
72 - 两圆交点坐标计算
73 - 直线与圆的交点计算及数量判断 (相离 / 相切 / 相交)
74 - 点到圆的切线切点计算 (点在圆内 / 上 / 外的处理)
75 - 两圆公切线的切点计算 (针对第一个圆)
76 - 三角形内切圆与外接圆计算 (圆心与半径)
77 - 平面点集操作
78 - 平面最近点对距离 (分治法实现)
79 - 最小圆覆盖 (随机增量法, 覆盖所有点的最小圆)
80 - 多边形计算
81 - 多边形面积计算 (凹凸多边形通用)
82 - 多边形重心计算 (凹凸多边形通用)
83 - 点是否在多边形内的判断 (含边界、内部、外部)
84 - 凸包构造 (Andrew 算法, 返回凸包顶点)
85 - 凸多边形的凸性判断
86 - 凸多边形被直线切割的面积计算
87 - 特殊算法与定理
88 - 旋转卡壳求凸包直径 (凸多边形最远点对距离)
89 - Pick 定理 (格点多边形面积与内点、边上点数量关系)
90
91 十、字符串
92 - 马拉车算法 (Manacher): 线性时间求解最长回文子串, 统一处理奇/偶长度回文
93 - Z函数: 计算字符串每个位置与前缀的最长公共前缀长度, 用于模式匹配
94 - 后缀数组 (SuffixArray): 构建字符串所有后缀的排序数组, 支持 (LCP) 和 (LCS) 查询
95 - 后缀自动机 (SAM): 压缩存储字符串所有子串, 支持子串存在性、出现次数等查询
96 - 回文自动机 (PAM): 高效处理字符串中所有回文子串, 支持回文计数与最长回文查询
97 - AC自动机 (AhoCorasick): 多模式串匹配算法, 快速在文本中查找多个模式串
98 - 字符字典树 (CharTrie): 基于字符的前缀树, 用于前缀匹配、字符串计数等
99 - 二进制字典树 (BinTrie): 处理二进制整数的前缀树, 用于异或最值、区间异或查询等
100 - KMP算法 (前缀函数): 单模式串匹配, 通过前缀函数优化匹配效率
101 - 字符串哈希: 通过哈希值快速比较子串, 支持随机底模降低冲突概率
102
103
104 */
105
```

```
106 #include<bits/stdc++.h>
107 using namespace std;
108
109 // 基础宏定义
110 #define dbg(x) cout << #x << ":" << x << " "
111 #define arrout(a,n) for(int i=1;i<=n;i++) cout << a[i] << " "
112 #define arrin(a,n) for(int i=1;i<=n;i++) cin >> a[i]
113 #define rep(i,x,n) for(int i=x;i<=n;i++)
114 #define dep(i,x,n) for(int i=x;i>=n;i--)
115 #define mem(a,x) memset(a,x,sizeof(a))
116 #define all(x) x.begin(),x.end()
117 #define arrall(a,n) a+1,a+1+n
118 #define PII pair<int,int>
119 #define m_p make_pair
120 #define ff first
121 #define ss second
122 #define CD const double
123 #define CI const int
124 #define int long long
125 #define itn int
126
127 // 基础模板主函数示例
128 signed main_basic() {
129     int a[10];
130     arrin(a,5);
131     arrout(a,5);
132     return 0;
133 }
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
```

```
160 // ===== 二、前缀和与差分 =====
161 void prefix_sum_demo() {
162     const int N = 1e6 + 5;
163     int n, m;
164     int a[N], b[N];
165
166     ios::sync_with_stdio(false);
167     cin >> n >> m;
168     rep(i, 1, n) {
169         cin >> a[i];
170         b[i] = a[i] + b[i-1];
171     }
172     rep(i, 1, m) {
173         int x, y;
174         cin >> x >> y;
175         cout << b[y] - b[x-1] << endl;
176     }
177 }
178
179 void difference_demo() {
180     const int N = 1e6 + 5;
181     int a[N], b[N];
182     int n, m, l, r, c, x = 0;
183
184     cin >> n >> m;
185     rep(i, 1, n) {
186         cin >> a[i];
187         b[i] = a[i] - a[i-1];
188     }
189     while (m--) {
190         cin >> l >> r >> c;
191         b[l] += c;
192         b[r+1] -= c;
193     }
194     rep(i, 1, n) {
195         x += b[i];
196         cout << x << " ";
197     }
198 }
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
```

```
214 // ===== 三、查找算法 =====
215 void binary_search_demo() {
216     const int N = 1e5 + 5;
217     int n, a[N], x, ans = 0;
218
219     ios::sync_with_stdio(0);
220     cin >> n;
221     rep(i, 1, n) cin >> a[i];
222     cin >> x;
223
224     int l = 1, r = n;
225     while (l <= r) {
226         int mid = (l + r) >> 1;
227         if (a[mid] == x) {
228             ans = mid;
229             r = mid - 1;
230         } else if (a[mid] < x) {
231             l = mid + 1;
232         } else {
233             r = mid - 1;
234         }
235     }
236
237     cout << (ans ? ans : -1) << endl;
238 }
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
```

```
268 // ===== 四、排序算法 =====
269 // 1. 归并排序（附带逆序对统计）
270 // 全局变量暂存归并所需数据（替代lambda捕获）
271 int *merge_a, *merge_r;
272 int merge_sum;
273
274 // 归并排序核心函数（替代lambda）
275 void msort_merge(int left, int right) {
276     if (left == right) return;
277     int mid = (left + right) / 2;
278     msort_merge(left, mid);
279     msort_merge(mid + 1, right);
280
281     int i = left, j = mid + 1, k = left;
282     while (i <= mid && j <= right) {
283         if (merge_a[i] <= merge_a[j]) {
284             merge_r[k++] = merge_a[i++];
285         } else {
286             merge_r[k++] = merge_a[j++];
287             merge_sum += mid - i + 1;
288         }
289     }
290     while (i <= mid) merge_r[k++] = merge_a[i++];
291     while (j <= right) merge_r[k++] = merge_a[j++];
292     rep(i, left, right) merge_a[i] = merge_r[i];
293 }
294
295 void merge_sort_with_inversion() {
296     const int N = 1e5 + 5;
297     int a[N], r[N];
298     int n;
299     merge_sum = 0; // 初始化逆序对计数
300
301     cin >> n;
302     rep(i, 1, n) cin >> a[i];
303     // 绑定全局指针到当前数组
304     merge_a = a;
305     merge_r = r;
306     msort_merge(1, n); // 调用普通函数而非lambda
307
308     cout << "逆序对数量: " << merge_sum << endl;
309 }
310
311 // 2. 计数排序
312 void counting_sort_demo() {
313     const int N = 1e5 + 5;
314     int n, x, a[N] = {0};
315
316     cin >> n;
317     rep(i, 1, n) {
318         cin >> x;
319         a[x] = x;
320     }
321     rep(i, 0, x) {
```

```
322         if (a[i]) {
323             cout << i << " ";
324             a[i] = 0;
325         }
326     }
327 }
328
329 // 3. 快速排序（归并实现版）
330 int *qsort_a, *qsort_r;
331
332 void msort_quick(int left, int right) {
333     if (left == right) return;
334     int mid = (left + right) >> 1;
335     msort_quick(left, mid);
336     msort_quick(mid + 1, right);
337
338     int i = left, j = mid + 1, k = left;
339     while (i <= mid && j <= right) {
340         if (qsort_a[i] <= qsort_a[j]) qsort_r[k++] = qsort_a[i++];
341         else qsort_r[k++] = qsort_a[j++];
342     }
343     while (i <= mid) qsort_r[k++] = qsort_a[i++];
344     while (j <= right) qsort_r[k++] = qsort_a[j++];
345     rep(i, left, right) qsort_a[i] = qsort_r[i];
346 }
347
348 void quick_sort_demo() {
349     const int N = 1e5 + 5;
350     int n, a[N], r[N];
351
352     cin >> n;
353     rep(i, 1, n) cin >> a[i];
354     qsort_a = a;
355     qsort_r = r;
356     msort_quick(1, n);
357
358     arrout(a, n);
359 }
360
361 // 4. 快速幂
362 int qpow(long long a, long long b, long long q) {
363     if (b == 0) return 1 % q;
364     long long ans = 1;
365     while (b) {
366         if (b & 1) ans = ans * a % q;
367         a = a * a % q;
368         b >>= 1;
369     }
370     return ans;
371 }
372
373
374
375
```

```
376 // =====五、动态规划 =====
377 // 1. 最长不下降子序列 (LIS)
378 void lis_demo() {
379     const int N = 1e4 + 5;
380     int n, a[N], dp[N], ans = 0;
381
382     cin >> n;
383     rep(i,1,n) {
384         cin >> a[i];
385         dp[i] = 1;
386     }
387
388     rep(i,1,n) {
389         rep(j,1,i-1) {
390             if (a[i] >= a[j]) {
391                 dp[i] = max(dp[i], dp[j] + 1);
392             }
393         }
394         ans = max(ans, dp[i]);
395     }
396
397     cout << "LIS长度: " << ans << endl;
398 }
399
400 // 2. 最长公共子序列 (LCS)
401 void lcs_demo() {
402     const int N = 2e3 + 5;
403     string a, b;
404     int len1, len2, dp[N][N] = {0};
405
406     cin >> a >> b;
407     len1 = a.size(), len2 = b.size();
408
409     rep(i,1,len1) {
410         rep(j,1,len2) {
411             dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
412             if (a[i-1] == b[j-1]) {
413                 dp[i][j] = max(dp[i][j], dp[i-1][j-1] + 1);
414             }
415         }
416     }
417
418     cout << "LCS长度: " << dp[len1][len2] << endl;
419 }
420
421
422
423
424
425
426
427
428
429
```

```
430 // ===== 六、数据结构 =====
431 // 1. 树状数组 (BIT)
432 // 1.1 单点修改 + 区间查询
433 int bit_n; // 树状数组大小 (全局暂存)
434 int *bit_c;
435
436 int lowbit(int x) { return x & (-x); }
437
438 void bit_update(int x, int v) {
439     for (int i = x; i <= bit_n; i += lowbit(i)) {
440         bit_c[i] += v;
441     }
442 }
443
444 int bit_get_sum(int l, int r) {
445     int ans = 0;
446     for (int i = r; i; i -= lowbit(i)) ans += bit_c[i];
447     for (int i = l-1; i; i -= lowbit(i)) ans -= bit_c[i];
448     return ans;
449 }
450
451 void bit_point_update_range_query() {
452     const int N = 1e6 + 5;
453     int a[N], c[N];
454     int m, x;
455
456     ios::sync_with_stdio(false);
457     cin >> bit_n >> m; // 赋值给全局变量
458     bit_c = c; // 绑定全局指针
459     rep(i, 1, bit_n) {
460         cin >> a[i];
461         bit_update(i, a[i]);
462     }
463     while (m--) {
464         int op, k, v;
465         cin >> op >> k >> v;
466         if (op) bit_update(k, v);
467         else cout << bit_get_sum(k, v) << endl;
468     }
469 }
470
471 // 1.2 区间修改 + 单点查询
472 int *br_c, br_n;
473
474 void br_update(int k, int v) {
475     for (int i = k; i <= br_n; i += lowbit(i)) {
476         br_c[i] += v;
477     }
478 }
479
480 int br_get_val(int x) {
481     int ans = 0;
482     for (int i = x; i; i -= lowbit(i)) ans += br_c[i];
483     return ans;
```

```
484 }
485
486 void bit_range_update_point_query() {
487     const int N = 1e6 + 5;
488     int a[N], b[N], c[N];
489     int m;
490
491     ios::sync_with_stdio(false);
492     cin >> br_n >> m;
493     br_c = c;
494     rep(i, 1, br_n) {
495         cin >> a[i];
496         b[i] = a[i] - a[i-1];
497         br_update(i, b[i]);
498     }
499     while (m--) {
500         int op, x, y, k;
501         cin >> op;
502         if (op == 1) {
503             cin >> x >> y >> k;
504             br_update(x, k);
505             br_update(y+1, -k);
506         } else {
507             cin >> x;
508             cout << br_get_val(x) << endl;
509         }
510     }
511 }
512
513 // 1.3 区间修改 + 区间查询
514 int *rr_b, *rr_c, rr_n;
515
516 void rr_update(int x, int y) {
517     for (int i = x; i <= rr_n; i += lowbit(i)) {
518         rr_b[i] += y;
519         rr_c[i] += x * y;
520     }
521 }
522
523 int rr_query(int x) {
524     int ans = 0;
525     for (int i = x; i; i -= lowbit(i)) {
526         ans += (x + 1) * rr_b[i] - rr_c[i];
527     }
528     return ans;
529 }
530
531 int rr_range_query(int l, int r) {
532     return rr_query(r) - rr_query(l-1);
533 }
534
535 void rr_range_update(int l, int r, int x) {
536     rr_update(l, x);
537     rr_update(r+1, -x);
```

```
538 }
539
540 void bit_range_update_range_query() {
541     const int N = 1e6 + 5;
542     int a[N], b[N], c[N];
543     int m;
544
545     ios::sync_with_stdio(false);
546     cin >> rr_n >> m;
547     rr_b = b;
548     rr_c = c;
549     rep(i, 1, rr_n) {
550         cin >> a[i];
551         rr_update(i, a[i] - a[i-1]);
552     }
553     while (m--) {
554         int op, x, y, k;
555         cin >> op;
556         if (op == 1) {
557             cin >> x >> y >> k;
558             rr_range_update(x, y, k);
559         } else {
560             cin >> x >> y;
561             cout << rr_range_query(x, y) << endl;
562         }
563     }
564 }
565
566 // 2. 线段树
567 // 2.1 单点修改 + 区间查询
568 int *seg_a, *seg_s, seg_n;
569
570 void seg_build(int k, int l, int r) {
571     if (l == r) {
572         seg_s[k] = seg_a[l];
573         return;
574     }
575     int mid = (l + r) >> 1;
576     seg_build(2*k, l, mid);
577     seg_build(2*k+1, mid+1, r);
578     seg_s[k] = seg_s[2*k] + seg_s[2*k+1];
579 }
580
581 void seg_update(int k, int l, int r, int x, int v) {
582     if (x < l || x > r) return;
583     if (l == r) {
584         seg_s[k] += v;
585         return;
586     }
587     int mid = (l + r) >> 1;
588     seg_update(2*k, l, mid, x, v);
589     seg_update(2*k+1, mid+1, r, x, v);
590     seg_s[k] = seg_s[2*k] + seg_s[2*k+1];
591 }
```

```
592
593 int seg_query(int k, int l, int r, int x, int y) {
594     if (y < l || x > r) return 0;
595     if (x <= l && r <= y) return seg_s[k];
596     int mid = (l + r) >> 1;
597     return seg_query(2*k, l, mid, x, y) + seg_query(2*k+1, mid+1, r, x, y);
598 }
599
600 void segment_tree_point_update_range_query() {
601     const int N = 1e5 + 5;
602     int a[N], s[4*N];
603     int n;
604
605     ios::sync_with_stdio(false);
606     cin >> n;
607     seg_n = n;
608     seg_a = a;
609     seg_s = s;
610     rep(i, 1, n) cin >> a[i];
611     seg_build(1, 1, n);
612
613     string op;
614     int x, y;
615     while (cin >> op) {
616         cin >> x >> y;
617         if (op == "Add") seg_update(1, 1, n, x, y);
618         else if (op == "Query") cout << seg_query(1, 1, n, x, y) << endl;
619     }
620 }
621
622 // 2.2 区间修改 + 区间查询（延迟标记）
623 int *seg2_a, *seg2_s, *seg2_lazy, seg2_n;
624
625 void seg2_build(int k, int l, int r) {
626     if (l == r) {
627         seg2_s[k] = seg2_a[l];
628         return;
629     }
630     int mid = (l + r) >> 1;
631     seg2_build(2*k, l, mid);
632     seg2_build(2*k+1, mid+1, r);
633     seg2_s[k] = seg2_s[2*k] + seg2_s[2*k+1];
634 }
635
636 void seg2_pushdown(int k, int l, int r) {
637     if (seg2_lazy[k] == 0) return;
638     int mid = (l + r) >> 1;
639     seg2_s[2*k] += (mid - l + 1) * seg2_lazy[k];
640     seg2_lazy[2*k] += seg2_lazy[k];
641     seg2_s[2*k+1] += (r - mid) * seg2_lazy[k];
642     seg2_lazy[2*k+1] += seg2_lazy[k];
643     seg2_lazy[k] = 0;
644 }
645
```

```
646 void seg2_update(int k, int l, int r, int x, int y, int v) {
647     if (y < l || x > r) return;
648     if (x <= l && r <= y) {
649         seg2_s[k] += (r - l + 1) * v;
650         seg2_lazy[k] += v;
651         return;
652     }
653     seg2_pushdown(k, l, r);
654     int mid = (l + r) >> 1;
655     seg2_update(2*k, l, mid, x, y, v);
656     seg2_update(2*k+1, mid+1, r, x, y, v);
657     seg2_s[k] = seg2_s[2*k] + seg2_s[2*k+1];
658 }
659
660 int seg2_query(int k, int l, int r, int x, int y) {
661     if (y < l || x > r) return 0;
662     if (x <= l && r <= y) return seg2_s[k];
663     seg2_pushdown(k, l, r);
664     int mid = (l + r) >> 1;
665     return seg2_query(2*k, l, mid, x, y) + seg2_query(2*k+1, mid+1, r, x, y);
666 }
667
668 void segment_tree_range_update_range_query() {
669     const int N = 1e6 + 5;
670     int a[N], s[4*N], lazy[4*N] = {0};
671     int n, m;
672
673     ios::sync_with_stdio(false);
674     cin >> n >> m;
675     seg2_n = n;
676     seg2_a = a;
677     seg2_s = s;
678     seg2_lazy = lazy;
679     rep(i, 1, n) cin >> a[i];
680     seg2_build(1, 1, n);
681
682     char op;
683     int x, y, z;
684     while (m--) {
685         cin >> op;
686         if (op == 'Q') {
687             cin >> x >> y;
688             cout << seg2_query(1, 1, n, x, y) << endl;
689         } else {
690             cin >> x >> y >> z;
691             seg2_update(1, 1, n, x, y, z);
692         }
693     }
694 }
```

```
700 // ===== 七、图论 =====
701 // 1. 图的存储（邻接表）
702 struct Graph {
703     const static int N = 1e5 + 5;
704     struct Edge { int to, nex; };
705     Edge e[N];
706     int head[N], tot = 0;
707
708     void add_edge(int x, int y) {
709         e[++tot] = {y, head[x]};
710         head[x] = tot;
711         e[++tot] = {x, head[y]};
712         head[y] = tot;
713     }
714
715     void traverse(int u) {
716         for (int i = head[u]; i; i = e[i].nex) {
717             int v = e[i].to;
718             cout << v << " ";
719         }
720     }
721 };
722
723 // 2. 最短路算法
724 // 2.1 BFS最短路
725 int bfs_n, bfs_st, bfs_ed;
726 int bfs_vis[40], bfs_dis[40];
727 int bfs_e[40][40];
728
729 void bfs() {
730     mem(bfs_dis, 0x3f);
731     queue<int> q;
732     q.push(bfs_st);
733     bfs_vis[bfs_st] = 1;
734     bfs_dis[bfs_st] = 0;
735
736     while (!q.empty()) {
737         int x = q.front(); q.pop();
738         rep(i, 1, bfs_ed) {
739             if (bfs_e[x][i] && !bfs_vis[i]) {
740                 bfs_vis[i] = 1;
741                 bfs_dis[i] = bfs_dis[x] + 1;
742                 if (i == bfs_ed) {
743                     cout << bfs_dis[i] << endl;
744                     return;
745                 }
746                 q.push(i);
747             }
748         }
749     }
750 }
751
752 void bfs_shortest_path() {
753     int n;
```

```
754     cin >> bfs_st >> bfs_ed >> n;
755     mem(bfs_e, 0);
756     mem(bfs_vis, 0);
757     rep(i,1,n) {
758         int x, y; cin >> x >> y;
759         bfs_e[x][y] = bfs_e[y][x] = 1;
760     }
761     bfs(); // 调用普通函数而非lambda
762 }
763
764 // 2.2 Floyd算法
765 int floyd_n;
766 int floyd_dp[1003][1003];
767
768 void floyd() {
769     rep(k,1,floyd_n) {
770         rep(i,1,floyd_n) {
771             rep(j,1,floyd_n) {
772                 floyd_dp[i][j] = min(floyd_dp[i][j], floyd_dp[i][k] + floyd_dp[k][j]);
773             }
774         }
775     }
776 }
777
778 void floyd_shortest_path() {
779     int m, st, ed;
780     cin >> floyd_n >> m;
781     mem(floyd_dp, 0x3f);
782     rep(i,1,floyd_n) floyd_dp[i][i] = 0;
783
784     rep(i,1,m) {
785         int x, y; cin >> x >> y;
786         floyd_dp[x][y] = floyd_dp[y][x] = 1;
787     }
788     cin >> st >> ed;
789     floyd();
790     cout << floyd_dp[st][ed] << endl;
791 }
792
793 // 2.3 Dijkstra算法
794 struct DijkstraEdge { int nex, to, data; };
795 DijkstraEdge dijkstra_a[1000005];
796 int dijkstra_head[1000005], dijkstra_tot;
797 int dijkstra_dis[1000005];
798 int dijkstra_n, dijkstra_m, dijkstra_st, dijkstra_ed;
799
800 void dijkstra_add(int x, int y, int z) {
801     dijkstra_a[++dijkstra_tot] = {dijkstra_head[x], y, z};
802     dijkstra_head[x] = dijkstra_tot;
803     dijkstra_a[++dijkstra_tot] = {dijkstra_head[y], x, z};
804     dijkstra_head[y] = dijkstra_tot;
805 }
806
807 void dijkstra() {
```

```

808     mem(dijkstra_dis, 0x3f);
809     dijkstra_dis[dijkstra_st] = 0;
810     priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> q;
811     q.push(m_p(0, dijkstra_st));
812
813     while (!q.empty()) {
814         pair<int, int> curr = q.top(); q.pop();
815         int d = curr.first, u = curr.second;
816         if (d > dijkstra_dis[u]) continue;
817         for (int i = dijkstra_head[u]; i; i = dijkstra_a[i].nex) {
818             int v = dijkstra_a[i].to, w = dijkstra_a[i].data;
819             if (dijkstra_dis[v] > dijkstra_dis[u] + w) {
820                 dijkstra_dis[v] = dijkstra_dis[u] + w;
821                 q.push(m_p(dijkstra_dis[v], v));
822             }
823         }
824     }
825 }
826
827 void dijkstra_shortest_path() {
828     cin >> dijkstra_n >> dijkstra_m >> dijkstra_st >> dijkstra_ed;
829     dijkstra_tot = 0;
830     mem(dijkstra_head, 0);
831     rep(i, 1, dijkstra_m) {
832         int x, y, z; cin >> x >> y >> z;
833         dijkstra_add(x, y, z);
834     }
835     dijkstra();
836     cout << (dijkstra_dis[dijkstra_ed] == 0x3f3f3f3f ? -1 : dijkstra_dis[dijkstra_ed]) <<
endl;
837 }
838
839 // 2.4 SPFA算法
840 struct SpfaEdge { int to, data, nex; };
841 SpfaEdge spfa_e[1000005];
842 int spfa_head[1000005], spfa_tot;
843 int spfa_dis[1000005], spfa_cnt[1000005], spfa_vis[1000005];
844 int spfa_n, spfa_m, spfa_st, spfa_ed;
845
846 void spfa_add(int x, int y, int z) {
847     spfa_e[++spfa_tot] = {y, z, spfa_head[x]};
848     spfa_head[x] = spfa_tot;
849 }
850
851 void spfa() {
852     mem(spfa_dis, 0x3f);
853     spfa_dis[spfa_st] = 0;
854     queue<int> q;
855     q.push(spfa_st);
856     spfa_vis[spfa_st] = 1;
857
858     while (!q.empty()) {
859         int u = q.front(); q.pop();
860         spfa_vis[u] = 0;

```

```
861
862     for (int i = spfa_head[u]; i; i = spfa_e[i].nex) {
863         int v = spfa_e[i].to, w = spfa_e[i].data;
864         if (spfa_dis[v] > spfa_dis[u] + w) {
865             spfa_dis[v] = spfa_dis[u] + w;
866             spfa_cnt[v] = spfa_cnt[u] + 1;
867             if (spfa_cnt[v] > spfa_n) {
868                 cout << "有负环" << endl;
869                 return;
870             }
871             if (!spfa_vis[v]) {
872                 spfa_vis[v] = 1;
873                 q.push(v);
874             }
875         }
876     }
877 }
878 }
879
880 void spfa_shortest_path() {
881     cin >> spfa_n >> spfa_m >> spfa_st >> spfa_ed;
882     spfa_tot = 0;
883     mem(spfa_head, 0);
884     mem(spfa_cnt, 0);
885     mem(spfa_vis, 0);
886     rep(i, 1, spfa_m) {
887         int x, y, z; cin >> x >> y >> z;
888         spfa_add(x, y, z);
889     }
890     spfa();
891     cout << (spfa_dis[spfa_ed] == 0x3f3f3f3f ? -1 : spfa_dis[spfa_ed]) << endl;
892 }
893
894 // 3. 最小生成树 (Kruskal算法)
895 struct KruskalEdge {
896     int from, to, data;
897     bool operator<(const KruskalEdge& b) const {
898         return data < b.data;
899     }
900 } kruskal_e[100005];
901 int kruskal_f[100005];
902 int kruskal_n;
903
904 int kruskal_find(int x) {
905     return kruskal_f[x] == x ? x : kruskal_f[x] = kruskal_find(kruskal_f[x]);
906 }
907
908 void kruskal_mst() {
909     int ans = 0;
910     cin >> kruskal_n;
911     rep(i, 1, kruskal_n) kruskal_f[i] = i;
912
913     rep(i, 1, kruskal_n) {
914         rep(j, 1, kruskal_n) {
```

```
915         int x; cin >> x;
916         kruskal_e[++kruskal_f[0]] = {i, j, x};
917     }
918 }
919
920 sort(kruskal_e + 1, kruskal_e + 1 + kruskal_f[0]);
921 int cnt = 0;
922 rep(i, 1, kruskal_f[0]) {
923     int u = kruskal_e[i].from, v = kruskal_e[i].to;
924     if (kruskal_find(u) != kruskal_find(v)) {
925         kruskal_f[kruskal_find(u)] = kruskal_find(v);
926         ans += kruskal_e[i].data;
927         if (++cnt == kruskal_n - 1) break;
928     }
929 }
930
931 cout << "最小生成树总权值: " << ans << endl;
932 }
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
```

```
969 // ===== 八、数学 =====
970 // 1. 线性筛（欧拉筛）求素数
971 void linear_sieve() {
972     const int N = 1e6 + 5;
973     int p[N], vis[N] = {0};
974     int n, cnt = 0;
975
976     cin >> n;
977     rep(i, 2, n) {
978         if (!vis[i]) p[++cnt] = i;
979         rep(j, 1, cnt) {
980             if (p[j] * i > n) break;
981             vis[p[j] * i] = 1;
982             if (i % p[j] == 0) break;
983         }
984     }
985
986     rep(i, 1, cnt) cout << p[i] << " ";
987 }
988
989 // 2. 唯一分解定理（质因数分解）
990 void prime_factorization() {
991     int x;
992     cin >> x;
993     vector<pair<int, int>> factors;
994
995     for (int i = 2; i * i <= x; i++) {
996         if (x % i == 0) {
997             int cnt = 0;
998             while (x % i == 0) {
999                 x /= i;
1000                 cnt++;
1001             }
1002             factors.push_back(m_p(i, cnt));
1003         }
1004     }
1005     if (x > 1) factors.push_back(m_p(x, 1));
1006
1007     rep(i, 0, factors.size()-1) {
1008         PII p = factors[i];
1009         cout << p.first << "^" << p.second;
1010         if (i != factors.size()-1) cout << " * ";
1011     }
1012 }
1013
1014 // 3. 中国剩余定理（CRT）
1015 int crt_exgcd(int a, int b, int& x, int& y) {
1016     if (b == 0) { x = 1; y = 0; return a; }
1017     int ret = crt_exgcd(b, a%b, x, y);
1018     int t = x; x = y; y = t - (a/b)*y;
1019     return ret;
1020 }
1021
1022 int crt_demo() {
```

```
1023     int n;
1024     cin >> n;
1025     vector<int> a(n+1), m(n+1);
1026     rep(i,1,n) cin >> m[i] >> a[i];
1027
1028     int M = 1, ans = 0;
1029     rep(i,1,n) M *= m[i];
1030     rep(i,1,n) {
1031         int Mi = M / m[i];
1032         int x, y;
1033         crt_exgcd(Mi, m[i], x, y);
1034         ans = (ans + (long long)x * a[i] * Mi) % M;
1035     }
1036     if (ans < 0) ans += M;
1037
1038     return ans;
1039 }
1040
1041 // 4. 最大公约数 (GCD)
1042 int gcd(int a, int b) {
1043     return b == 0 ? a : gcd(b, a % b);
1044 }
1045
1046 // 5. 欧拉函数
1047 // 5.1 批量计算1~n的欧拉函数
1048 void euler_phi_range() {
1049     int n;
1050     cin >> n;
1051     vector<int> phi(n+1);
1052     rep(i,1,n) phi[i] = i;
1053     phi[1] = 1;
1054
1055     rep(i,2,n) {
1056         if (phi[i] == i) {
1057             for (int j = i; j <= n; j += i) {
1058                 phi[j] = phi[j] / i * (i - 1);
1059             }
1060         }
1061     }
1062
1063     rep(i,1,n) cout << "φ(" << i << ")=" << phi[i] << endl;
1064 }
1065
1066 // 5.2 计算单个数字的欧拉函数
1067 int euler_phi_single(int n) {
1068     int ans = n;
1069     for (int i = 2; i * i <= n; i++) {
1070         if (n % i == 0) {
1071             ans = ans / i * (i - 1);
1072             while (n % i == 0) n /= i;
1073         }
1074     }
1075     if (n > 1) ans = ans / n * (n - 1);
1076     return ans;
```

```
1077 }
1078
1079 // 6. 矩阵乘法与快速幂
1080 const int MOD = 1e9 + 7;
1081 const int MAT_SIZE = 105;
1082
1083 struct Matrix {
1084     int mat[MAT_SIZE][MAT_SIZE];
1085     Matrix() { mem(mat, 0); }
1086     Matrix operator*(const Matrix& b) const {
1087         Matrix ans;
1088         rep(i, 1, MAT_SIZE) {
1089             rep(j, 1, MAT_SIZE) {
1090                 rep(k, 1, MAT_SIZE) {
1091                     ans.mat[i][j] = (ans.mat[i][j] +
1092                                     (long long)mat[i][k] * b.mat[k][j] % MOD) % MOD;
1093                 }
1094             }
1095         }
1096         return ans;
1097     }
1098 };
1099
1100 Matrix matrix_qpow(Matrix a, int b) {
1101     Matrix ans;
1102     rep(i, 1, MAT_SIZE) ans.mat[i][i] = 1;
1103     while (b) {
1104         if (b & 1) ans = ans * a;
1105         a = a * a;
1106         b >>= 1;
1107     }
1108     return ans;
1109 }
1110
1111 // 7. 组合数计算
1112 // 7.1 Lucas定理
1113 int lucas_MOD = 1e9 + 7;
1114
1115 int lucas_C(int a, int b) {
1116     if (b > a) return 0;
1117     if (b == 0) return 1;
1118     long long up = 1, down = 1;
1119     rep(i, 0, b-1) {
1120         up = up * (a - i) % lucas_MOD;
1121         down = down * (i + 1) % lucas_MOD;
1122     }
1123     return up * qpow(down, lucas_MOD-2, lucas_MOD) % lucas_MOD;
1124 }
1125
1126 int lucas(int a, int b) {
1127     if (b == 0) return 1;
1128     return (long long)lucas(a / lucas_MOD, b / lucas_MOD) * lucas_C(a % lucas_MOD, b %
lucas_MOD) % lucas_MOD;
1129 }
```

```
1130
1131 int lucas_demo() {
1132     int n, m;
1133     cin >> n >> m;
1134     lucas_MOD = 1e9 + 7;
1135     return lucas(n, m);
1136 }
1137
1138 // 7.2 预处理阶乘+逆元
1139 // 7.2 预处理阶乘+逆元（移除lambda版本）
1140 // 全局变量用于传递组合数计算所需的阶乘、逆元和模数
1141 const int COMB_N = 1e6 + 5;
1142 int comb_fac[COMB_N]; // 阶乘数组
1143 int comb_inv[COMB_N]; // 逆元数组
1144 int comb_MOD;          // 模数
1145
1146 // 普通函数替代lambda，计算组合数C(n,m)
1147 int comb_C(int n, int m) {
1148     if (m < 0 || m > n) return 0;
1149     return (long long)comb_fac[n] * comb_inv[m] % comb_MOD * comb_inv[n - m] % comb_MOD;
1150 }
1151
1152 void comb_preprocess_demo() {
1153     // 初始化全局变量（替代lambda的[&]捕获）
1154     comb_MOD = 1e9 + 7;
1155     // 预处理阶乘
1156     comb_fac[0] = 1;
1157     for (int i = 1; i < COMB_N; i++) {
1158         comb_fac[i] = (long long)comb_fac[i-1] * i % comb_MOD;
1159     }
1160     // 预处理逆元（费马小定理）
1161     comb_inv[COMB_N - 1] = qpow(comb_fac[COMB_N - 1], comb_MOD - 2, comb_MOD);
1162     for (int i = COMB_N - 2; i >= 0; i--) {
1163         comb_inv[i] = (long long)comb_inv[i + 1] * (i + 1) % comb_MOD;
1164     }
1165
1166     // 处理查询（直接调用普通函数comb_C）
1167     int q;
1168     cin >> q;
1169     while (q--) {
1170         int n, m;
1171         cin >> n >> m;
1172         cout << comb_C(n, m) << endl;
1173     }
1174 }
1175
1176
1177 /*
1178     |大数模拟加法|
1179     |用string模拟|
1180     |16/11/05ztx, thanks to caojiji|
1181 */
1182
1183 string add1(string s1, string s2)
```

```

1184 {
1185     if (s1 == "" && s2 == "")    return "0";
1186     if (s1 == "")    return s2;
1187     if (s2 == "")    return s1;
1188     string maxx = s1, minn = s2;
1189     if (s1.length() < s2.length()){
1190         maxx = s2;
1191         minn = s1;
1192     }
1193     int a = maxx.length() - 1, b = minn.length() - 1;
1194     for (int i = b; i >= 0; --i){
1195         maxx[a--] += minn[i] - '0'; // a一直在减 , 额外还要减个'0'
1196     }
1197     for (int i = maxx.length()-1; i > 0; --i){
1198         if (maxx[i] > '9'){
1199             maxx[i] -= 10; //注意这个是减10
1200             maxx[i - 1]++;
1201         }
1202     }
1203     if (maxx[0] > '9'){
1204         maxx[0] -= 10;
1205         maxx = '1' + maxx;
1206     }
1207     return maxx;
1208 }
1209
1210 /*
1211 |大数模拟阶乘|
1212 |用数组模拟|
1213 |16/12/02ztx|
1214 */
1215
1216
1217 typedef long long LL;
1218
1219 const int maxn = 100010;
1220
1221 int num[maxn], len;
1222
1223 /*
1224     在mult函数中, 形参部分: len每次调用函数都会发生改变, n表示每次要乘以的数, 最终返回的是结
    果的长度
1225     tip: 阶乘都是先求之前的(n-1)!来求n!
1226     初始化Init函数很重要, 不要落下
1227 */
1228
1229 void Init() {
1230     len = 1;
1231     num[0] = 1;
1232 }
1233
1234 int mult(int num[], int len, int n) {
1235     LL tmp = 0;
1236     for(LL i = 0; i < len; ++i) {

```

```
1237         tmp = tmp + num[i] * n;    //从最低位开始，等号左边的tmp表示当前位，右边的tmp表示
进位（之前进的位）
1238         num[i] = tmp % 10; // 保存在对应的数组位置，即去掉进位后的一位数
1239         tmp = tmp / 10;    // 取整用于再次循环，与n和下一个位置的乘积相加
1240     }
1241     while(tmp) {    // 之后的进位处理
1242         num[len++] = tmp % 10;
1243         tmp = tmp / 10;
1244     }
1245     return len;
1246 }
1247
1248 int main() {
1249     Init();
1250     int n;
1251     n = 1977; // 求的阶乘数
1252     for(int i = 2; i <= n; ++i) {
1253         len = mult(num, len, i);
1254     }
1255     for(int i = len - 1; i >= 0; --i)
1256         printf("%d", num[i]);    // 从最高位依次输出，数据比较多采用printf输出
1257     printf("\n");
1258     return 0;
1259 }
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
```

```
1290 // ===== 九、计算几何 =====
1291 #include <bits/stdc++.h>
1292
1293 const double EPS = 1e-8;
1294
1295 int sign(double val) {
1296     if (std::abs(val) < EPS) return 0;
1297     if (val > 0) return 1;
1298     return -1;
1299 }
1300
1301 int dcmp(double val1, double val2) {
1302     return sign(val1 - val2);
1303 }
1304
1305 struct Point {
1306     double x, y;
1307
1308     Point(double a = 0, double b = 0) : x(a), y(b) {}
1309
1310     friend Point operator+(Point a, Point b) {
1311         return Point(a.x + b.x, a.y + b.y);
1312     }
1313
1314     friend Point operator-(Point a, Point b) {
1315         return Point(a.x - b.x, a.y - b.y);
1316     }
1317
1318     friend Point operator/(Point a, double b) {
1319         return Point(a.x / b, a.y / b);
1320     }
1321
1322     friend Point operator*(Point a, double b) {
1323         return Point(a.x * b, a.y * b);
1324     }
1325
1326     friend Point operator*(double a, Point b) {
1327         return Point(b.x * a, b.y * a);
1328     }
1329
1330     friend double operator*(Point a, Point b) {
1331         return a.x * b.x + a.y * b.y;
1332     }
1333
1334     friend bool operator==(Point a, Point b) {
1335         return (!sign(a.x - b.x) && !sign(a.y - b.y));
1336     }
1337
1338     friend bool operator!=(Point a, Point b) {
1339         return !(a == b);
1340     }
1341
1342     // 用%代表叉乘,因为 ^ 优先级太低了
1343     friend double operator%(Point a, Point b) {
```

```
1344         return a.x * b.y - a.y * b.x;
1345     }
1346
1347     double ddis() { return x * x + y * y; }
1348
1349     double dis() { return sqrt(x * x + y * y); }
1350 } INFP(1e20, 1e20);
1351
1352 // 点到点的距离
1353 double dis(Point a, Point b) {
1354     return sqrt((a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y));
1355 }
1356
1357 // 点到点的距离（不开方）
1358 double ddis(Point a, Point b) {
1359     return (a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y);
1360 }
1361
1362 struct Seg {
1363     Point p1, p2;
1364
1365     Seg() {}
1366
1367     Seg(Point a, Point b) : p1(a), p2(b) {}
1368
1369     friend double operator*(Seg a, Seg b) {
1370         return (a.p2 - a.p1) * (b.p2 - b.p1);
1371     }
1372
1373     friend double operator%(Seg a, Seg b) {
1374         return (a.p2 - a.p1) % (b.p2 - b.p1);
1375     }
1376
1377     friend double operator%(Seg a, Point b) {
1378         return (a.p2 - a.p1) % (b - a.p1);
1379     }
1380
1381     friend double operator%(Point a, Seg b) {
1382         return (a - b.p1) % (b.p2 - b.p1);
1383     }
1384
1385     friend bool operator==(Seg a, Seg b) {
1386         return (a.p1 == b.p1 && a.p2 == b.p2) || (a.p1 == b.p2 && a.p2 == b.p1);
1387     }
1388
1389     void swap() {
1390         std::swap(p1, p2);
1391     }
1392
1393     double len() { return dis(p1, p2); }
1394 };
1395
1396 // 点到直线距离
1397 double calPSDis1(Seg a, Point b) {
```

```
1398     return std::abs((a.p2 - a.p1) % (b - a.p1)) / a.len();
1399 }
1400
1401 // 点到线段距离
1402 double calPSDis2(Seg a, Point b) {
1403     int res1 = sign((b - a.p1) * (a.p2 - a.p1));
1404     int res2 = sign((b - a.p2) * (a.p1 - a.p2));
1405     if (res1 >= 0 && res2 >= 0) {
1406         return calPSDis1(a, b);
1407     }
1408     return std::min(dis(a.p1, b), dis(a.p2, b));
1409 }
1410
1411 // 求点到线段/直线的垂足
1412 Point calPSFeet(Seg a, Point b) {
1413     auto vec = a.p2 - a.p1;
1414     return a.p1 + vec * ((b - a.p1) * vec) / (vec.x * vec.x + vec.y * vec.y);
1415 }
1416
1417 // 计算线段及其延长线/直线的交点
1418 Point calSegCross(Seg a, Seg b) {
1419     auto res1 = (a.p2 - a.p1) % (b.p1 - a.p1);
1420     auto res2 = (a.p2 - a.p1) % (b.p2 - a.p1);
1421     auto x = (res2 * b.p1.x - res1 * b.p2.x) / (res2 - res1);
1422     auto y = (res2 * b.p1.y - res1 * b.p2.y) / (res2 - res1);
1423     return Point(x, y);
1424 }
1425
1426 // 判断点是否在线段上（包括端点）
1427 bool isPAAtS(Seg a, Point b) {
1428     Point p1 = a.p1 - b, p2 = a.p2 - b;
1429     if (sign(p1 % p2) == 0 && sign(p1 * p2) <= 0) return true;
1430     return false;
1431 }
1432
1433 // 判断线段是否相交
1434 bool isSegCross(Seg a, Seg b) {
1435     double c1 = (a.p2 - a.p1) % (b.p1 - a.p1), c2 = (a.p2 - a.p1) % (b.p2 - a.p1);
1436     double c3 = (b.p2 - b.p1) % (a.p1 - b.p1), c4 = (b.p2 - b.p1) % (a.p2 - b.p1);
1437
1438     // 允许线段在端点处相交则添加
1439     if (!sign(c1) || !sign(c2) || !sign(c3) || !sign(c4)) {
1440         bool f1 = isPAAtS(a, b.p1);
1441         bool f2 = isPAAtS(a, b.p2);
1442         bool f3 = isPAAtS(b, a.p1);
1443         bool f4 = isPAAtS(b, a.p2);
1444         bool f = (f1 | f2 | f3 | f4);
1445         return f;
1446     }
1447     return (sign(c1) * sign(c2) < 0 && sign(c3) * sign(c4) < 0);
1448 }
1449
1450 // 计算平面最近点对欧式距离
1451 double calPointMinDis(int l, int r, Point point[], int t[]) {
```

```

1452     /*
1453     开始递归前先按照 x 排序
1454     std::sort(point, point + n, [&](const Point &p1, const Point &p2) {
1455         return p1.x < p2.x;
1456     });
1457     */
1458     if (r - l == 0)
1459         return 1e15;
1460     if (r - l == 1) // 如果递归完后直接输出距离
1461         return dis(point[l], point[r]);
1462     int mid = (l + r) >> 1;
1463     double ans = std::min(calPointMinDis(l, mid, point, t),
1464                          calPointMinDis(mid + 1, r, point, t));
1465     int cnt = 0;
1466     for (int i = l; i <= r; i++)
1467         // 还有一种情况是距离最小的两点刚好分在 mid 两端 ans 距离内的点
1468         if (point[i].x >= point[mid].x - ans && point[i].x <= point[mid].x + ans)
1469             t[++cnt] = i;
1470     std::sort(t + 1, t + cnt + 1, [&](int i, int j) {
1471         return point[i].y < point[j].y;
1472     }); // 以 y 坐标大小排序
1473     for (int i = 1; i <= cnt; i++)
1474         for (int j = i + 1; j <= cnt; j++) {
1475             if (point[t[j]].y >= point[t[i]].y + ans) break;
1476             // 两个点的垂直距离超过 ans 就不必计算了, 显然不可能会成为新的 ans
1477             ans = std::min(ans, dis(point[t[i]], point[t[j]]));
1478         }
1479     return ans;
1480 }
1481 struct Cir {
1482     Point o;
1483     double r;
1484
1485     Cir(Point oo, double rr) : o(oo), r(rr) {}
1486
1487     Cir(double x = 0, double y = 0, double rr = 0) { o.x = x, o.y = y, r = rr; }
1488
1489     Point getPoint(double theta) { // 根据极角返回圆上一点的坐标
1490         double co = cos(theta) * r;
1491         double si = sin(theta) * r;
1492         if (sign(co) == 0) co = 0.0;
1493         if (sign(si) == 0) si = 0.0;
1494         return Point(o.x + co, o.y + si);
1495     }
1496
1497     Point getPoint(Point p) { // 根据一点返回圆上一点的坐标
1498         auto vec = p - o;
1499         return o + vec * r / vec.dis();
1500     }
1501 };
1502
1503 // 判断圆相交情况
1504 int isCirCross(Cir a, Cir b) {
1505     // 返回值恰好为两个圆的公共切线数量

```

```

1506     int res = dcmp(dis(a.o, b.o), a.r + b.r);
1507     if (res > 0) return 4; // 两个圆不相交
1508     if (res == 0) return 3; // 两个圆外接
1509     res = dcmp(dis(a.o, b.o), std::abs(a.r - b.r));
1510     if (res > 0) return 2; // 两个圆相交
1511     if (res == 0) return 1; // 有一个圆内切另外一个圆
1512     return 0; // 一个圆在另一个圆内部
1513 }
1514
1515 // 计算两个圆的交点
1516 std::array<Point, 2> calCirCross(Cir c1, Cir c2) {
1517     Point vec12 = c2.o - c1.o; // 两圆圆心的向量
1518     double d = vec12.dis(); // 圆心距
1519     double a = acos((c1.r * c1.r + d * d - c2.r * c2.r) / (2.0 * c1.r * d));
1520     // vec12 与 (c1 与一个交点) 的夹角
1521     double t = atan2(vec12.y, vec12.x); // vec12 与x轴的夹角
1522     return {c1.getPoint(t + a), c1.getPoint(t - a)};
1523 }
1524
1525 // 三角形内切圆
1526 Cir calTriangleInnerCir(Point p1, Point p2, Point p3) {
1527     Cir c;
1528     auto d1 = dis(p1, p2), d2 = dis(p1, p3), d3 = dis(p2, p3);
1529     c.r = std::abs((p2.x - p1.x) * (p3.y - p1.y) -
1530         (p2.y - p1.y) * (p3.x - p1.x)) / (d1 + d2 + d3);
1531     c.o.x = (d3 * p1.x + d1 * p3.x + d2 * p2.x) / (d1 + d2 + d3);
1532     c.o.y = (d3 * p1.y + d1 * p3.y + d2 * p2.y) / (d1 + d2 + d3);
1533     return c;
1534 }
1535
1536 // 三角形外接圆
1537 Cir calTriangleOuterCir(Point p1, Point p2, Point p3) {
1538     Cir c;
1539     c.r = dis(p1, p2) * dis(p2, p3) * dis(p1, p3) / std::abs((p2 - p1) % (p3 - p1)) /
1540     2.0;
1541     auto a11 = (p2.x - p1.x);
1542     auto a12 = (p2.y - p1.y);
1543     auto b1 = (p1.x * (p2.x - p1.x) + p1.y * (p2.y - p1.y) + 0.5 * ddis(p1, p2));
1544     auto a21 = (p3.x - p1.x);
1545     auto a22 = (p3.y - p1.y);
1546     auto b2 = (p1.x * (p3.x - p1.x) + p1.y * (p3.y - p1.y) + 0.5 * ddis(p1, p3));
1547     auto down = (a11 * a22 - a21 * a12);
1548     c.o.x = (b1 * a22 - b2 * a12) / down;
1549     c.o.y = (a11 * b2 - b1 * a21) / down;
1550     return c;
1551 }
1552 // 直线和圆交点个数
1553 int isSegCrossCir(Seg s, Cir c) {
1554     int res = dcmp(calPSDis1(s, c.o), c.r);
1555     if (res == 0) return 1; // 外切
1556     if (res == 1) return 0; // 不相交
1557     return 2;
1558 }

```

```
1559
1560 // 计算直线与圆交点
1561 std::array<Point, 2> calSegCrossCir(Cir c, Seg s) {
1562     auto feet = calPSFeet(s, c.o);
1563     int sta = isSegCrossCir(s, c);
1564     if (sta == 0) return {INFP, INFP};
1565     if (sta == 1) return {feet, feet};
1566     double base = sqrt(c.r * c.r - (feet - c.o).ddis());
1567     auto vec = s.p2 - s.p1;
1568     Point e = vec / vec.dis();
1569     return {feet + e * base, feet - e * base};
1570 }
1571
1572 std::array<Point, 2> calPointCirTangent(Cir C, Point p) {
1573     auto d = dis(p, C.o);
1574     int aa = dcmp(d, C.r);
1575     if (aa < 0) return {INFP, INFP}; // 点在圆内
1576     else if (aa == 0) return {p, p}; // 点在圆上, 该点就是切点
1577
1578     // 点在圆外, 有两个切点
1579     double base = atan2(p.y - C.o.y, p.x - C.o.x);
1580     double ang = acos(C.r / d);
1581     return {C.getPoint(base - ang), C.getPoint(base + ang)};
1582 }
1583
1584 // 计算 c1 与 c2 的所有切线中 c1 的所有切点
1585 std::vector<Point> calCirTangents(Cir c1, Cir c2) {
1586     int CrossRes = isCirCross(c1, c2);
1587     std::vector<Point> res;
1588     auto d = dis(c1.o, c2.o);
1589     if (CrossRes == 0) return res;
1590     double base = atan2(c2.o.y - c1.o.y, c2.o.x - c1.o.x);
1591     // AB 向量的极角, c1 为大圆
1592     if (CrossRes == 1) { // 内切
1593         res.push_back(c1.getPoint(base));
1594         return res;
1595     }
1596     double ang1 = acos((c1.r - c2.r) / d); // 外公切线的图
1597     res.push_back(c1.getPoint(base + ang1));
1598     res.push_back(c1.getPoint(base - ang1));
1599     if (CrossRes == 3)
1600         res.push_back(c1.getPoint(base));
1601     // 一条内公切线, 此时c2上的点为 c2.getPoint(base + pi)
1602     else if (CrossRes == 4) { // 两条内公切线, 对应内公切线的图
1603         double ang2 = acos((c1.r + c2.r) / d);
1604         res.push_back(c1.getPoint(base + ang2));
1605         res.push_back(c1.getPoint(base - ang2));
1606     }
1607     return res;
1608 }
1609
1610 // 最小圆覆盖
1611 Cir calMinCoverCir(Point p[], int n) {
1612     std::random_device rd;
```

```

1613     std::mt19937 g(rd());
1614     std::shuffle(p, p + 1 + n, g);
1615     Cir c(p[0], 0);
1616     for (int i = 1; i < n; i++) {
1617         if (dcmp(dis(c.o, p[i]), c.r) > 0) {
1618             c.r = 0;
1619             c.o = p[i];
1620             for (int j = 0; j < i; j++) {
1621                 if (dcmp(dis(c.o, p[j]), c.r) > 0) {
1622                     c.r = dis(p[j], p[i]) / 2;
1623                     c.o = {(p[j].x + p[i].x) / 2, (p[j].y + p[i].y) / 2};
1624                     for (int k = 0; k < j; k++) {
1625                         if (dcmp(dis(c.o, p[k]), c.r) > 0) {
1626                             c = calTrangleOuterCir(p[i], p[j], p[k]);
1627                         }
1628                     }
1629                 }
1630             }
1631         }
1632     }
1633     return c;
1634 }
1635
1636 // 多边形的重心(凹凸都可以)
1637 Point calPolygonCentre(Point point[], int n) {
1638     double sum = 0.0, sumx = 0, sumy = 0;
1639     Point p1 = point[0], p2 = point[1], p3;
1640     for (int i = 2; i <= n - 1; i++) {
1641         p3 = point[i];
1642         double area = (p2 - p1) % (p3 - p2) / 2.0;
1643         sum += area;
1644         sumx += (p1.x + p2.x + p3.x) * area;
1645         sumy += (p1.y + p2.y + p3.y) * area;
1646         p2 = p3;
1647     }
1648     return Point(sumx / (3.0 * sum), sumy / (3.0 * sum));
1649 }
1650
1651 // 判断点是否在多边形内部(凹凸都可以)
1652 int isPointInPolygon(int n, Point point[], Point P) {
1653     int cnt = 0;
1654     Point P1, P2;
1655     for (int i = 0; i < n; i++) {
1656         P1 = point[i];
1657         P2 = point[(i + 1) % n];
1658         if (isPAAtS(Seg(P1, P2), P)) return 1; // 如果点在边上返回 1
1659         if ((sign(P1.y - P.y) > 0 != sign(P2.y - P.y) > 0) &&
1660             sign(P.x - (P.y - P1.y) * (P1.x - P2.x) / (P1.y - P2.y) - P1.x) < 0)
1661             cnt++;
1662     }
1663     if (cnt & 1) return 2; // 在内部返回 2
1664     else return 0; // 在外部返回 0
1665 }
1666

```

```

1667 // 多边形面积(凹凸都可以)
1668 double calPolygonArea(Point point[], int n) {
1669     double area = 0;
1670     for (int i = 0; i < n; i++)
1671         area += point[i] % point[(i + 1) % n];
1672     return std::abs(area) / 2.0;
1673 }
1674
1675 // 计算凸多边形被直线 s 切割的面积
1676 double calPolygonAreaCut(Point point[], int n, Seg s) {
1677     Point dir = s.p2 - s.p1, tmp[110];
1678     // 默认点的顺序为逆时针
1679     int cnt = 0;
1680     for (int i = 0; i < n; i++) {
1681         int res1 = sign((point[i] - s.p1) % dir);
1682         int res2 = sign((point[(i + 1) % n] - s.p1) % dir);
1683         if (res1 <= 0) tmp[cnt++] = point[i];
1684         // 求左侧面积, 如果求右侧改成 >= 0
1685         if (res1 * res2 == -1) {
1686             tmp[cnt++] = calSegCross(s, Seg(point[i], point[(i + 1) % n]));
1687         }
1688     }
1689     return calPolygonArea(tmp, cnt);
1690 }
1691
1692 // Pick 定理: 给定顶点均为整点的简单多边形, 皮克定理说明了
1693 // 其面积 A 和内部格点数目 i, 边上格点数目 b 的关系:
1694 //  $A = i + b / 2 - 1$  (没有取整, 可能是小数)
1695 // 取格点的组成图形的面积为各单位. 在平行四边形格点, 皮克定理依然成立。
1696 // 套用于任意三角形格点, 皮克定理则是  $A = 2 * i + b - 2$ 
1697 std::array<int, 2> pick(int n, Point point[]) {
1698
1699     int sum = 0, num = 0;
1700     for (int i = 0; i < n; i++) {
1701         int x = std::abs(point[(i + 1) % n].x - point[i].x) + 0.5;
1702         int y = std::abs(point[(i + 1) % n].y - point[i].y) + 0.5;
1703         num += std::__gcd(abs(x), abs(y));
1704         sum += point[(i + 1) % n] % point[i];
1705     }
1706     sum = abs(sum); // sum 为两倍的面积
1707     return {(sum - num) / 2 + 1, num};
1708 }
1709
1710 // andrew 找凸包
1711 int andrew(int n, Point point[], Point res[]) {
1712     // n >= 2
1713     // point 和 res 下标从 0 开始, 方便取模
1714     std::sort(point, point + n, [&](const Point &a, const Point &b) {
1715         if (dcmp(a.x, b.x) == 0) return a.y < b.y;
1716         return a.x < b.x;
1717     });
1718     int m = 0;
1719     for (int i = 0; i < n; i++) {
1720         while (m > 1 && (res[m - 1] - res[m - 2]) % (point[i] - res[m - 2]) <= 0) --m;

```

```
1721 // 如果想保留凸包边上的点,可以把 <= 换成 <
1722 // 凸包为逆时针方向
1723 res[m++] = point[i];
1724 }
1725 int k = m;
1726 for (int i = n - 2; i >= 0; i--) {
1727     while (m > k && (res[m - 1] - res[m - 2]) % (point[i] - res[m - 2]) <= 0) --m;
1728     res[m++] = point[i];
1729 }
1730 --m;
1731 // point[0] 被重复记录了,删去
1732 return m;
1733 }
1734
1735 // 点c到线段ab的距离
1736 double segdis(Point a, Point b, Point c) {
1737     return std::abs((b - a) % (c - a)) / dis(a, b);
1738 }
1739
1740 // 旋转卡壳求凸包直径
1741 double rotate(int n, Point res[]) {
1742     // 虽然 n >= 2 时即认为凸包存在,但是旋转卡壳的时候要求 n >= 3
1743     if (n == 2) return dis(res[1], res[0]);
1744     int cur = 0;
1745     double ans = 0;
1746     for (int i = 0; i < n; i++) {
1747         while (segdis(res[i], res[(i + 1) % n], res[cur]) <=
1748             segdis(res[i], res[(i + 1) % n], res[(cur + 1) % n]))
1749             cur = (cur + 1) % n;
1750         ans = std::max(ans, dis(res[i], res[cur]));
1751         ans = std::max(ans, dis(res[(i + 1) % n], res[cur]));
1752     }
1753     return ans;
1754 }
1755
1756 // 判断凸包
1757 bool isConvex(int n, Point point[]) {
1758     // 默认为顺时针
1759     // 如果输入点的方式为逆时针输入,则将 < 改成 >
1760     // 认为凸包边上可以有点 则加上 =
1761     for (int i = 0; i < n; i++) {
1762         if ((point[i] - point[(i + 1) % n]) % (point[(i + 2) % n] - point[(i + 1) % n]) >
1763             0)
1764             return false;
1765     }
1766     return true;
1767 }
1768
1769
1770
1771
1772
1773
```

```

1774 // ===== 十、字符串 =====
1775 using i64 = long long;
1776 constexpr int INF = 1e9;
1777 constexpr int ALPHABET = 26;
1778
1779 // 1. 马拉车算法：求最长回文子串
1780 vector<int> manacher(string s) {
1781     string t = "#";
1782     for (char c : s) { t += c; t += '#'; }
1783     int n = t.size(), j = 0;
1784     vector<int> r(n);
1785     for (int i = 0; i < n; i++) {
1786         if (2 * j - i >= 0 && j + r[j] > i)
1787             r[i] = min(r[2 * j - i], j + r[j] - i);
1788         while (i - r[i] >= 0 && i + r[i] < n && t[i - r[i]] == t[i + r[i]])
1789             r[i]++;
1790         if (i + r[i] > j + r[j]) j = i;
1791     }
1792     return r;
1793 }
1794
1795 // 2. Z函数：求每个位置最长前缀匹配
1796 vector<int> z_function(string s) {
1797     int n = s.size(), j = 0;
1798     vector<int> z(n); z[0] = n;
1799     for (int i = 1; i < n; i++) {
1800         z[i] = max((long long)0, min(j + z[j] - i, z[i - j]));
1801         while (i + z[i] < n && s[z[i]] == s[i + z[i]])
1802             z[i]++;
1803         if (i + z[i] > j + z[j]) j = i;
1804     }
1805     return z;
1806 }
1807
1808 // 3. 后缀数组：含LCP/LCS查询
1809 struct SuffixArray {
1810     int n;
1811     vector<int> sa, rk, lc;
1812     SuffixArray(string s) : n(s.size()), sa(n), rk(n), lc(n-1) {
1813         iota(sa.begin(), sa.end(), 0);
1814         sort(sa.begin(), sa.end(), [&](int a, int b) { return s[a] < s[b]; });
1815         rk[sa[0]] = 0;
1816         for (int i = 1; i < n; i++)
1817             rk[sa[i]] = rk[sa[i-1]] + (s[sa[i]] != s[sa[i-1]]);
1818         int k = 1; vector<int> tmp, cnt(n); tmp.reserve(n);
1819         while (rk[sa[n-1]] < n-1) {
1820             tmp.clear();
1821             for (int i = 0; i < k; i++) tmp.push_back(n - k + i);
1822             for (int i : sa) if (i >= k) tmp.push_back(i - k);
1823             fill(cnt.begin(), cnt.end(), 0);
1824             for (int i = 0; i < n; i++) cnt[rk[i]]++;
1825             for (int i = 1; i < n; i++) cnt[i] += cnt[i-1];
1826             for (int i = n-1; i >= 0; i--) sa[--cnt[rk[tmp[i]]]] = tmp[i];
1827             swap(rk, tmp); rk[sa[0]] = 0;

```

```

1828         for (int i = 1; i < n; i++) {
1829             bool diff = (tmp[sa[i-1]] != tmp[sa[i]]) || (sa[i-1]+k >= n) || (sa[i]+k
1830 >= n) || (tmp[sa[i-1]+k] != tmp[sa[i]+k]);
1831             rk[sa[i]] = rk[sa[i-1]] + diff;
1832         }
1833         k *= 2;
1834     }
1835     for (int i = 0, j = 0; i < n; i++) {
1836         if (rk[i] == 0) j = 0;
1837         else {
1838             j -= j > 0;
1839             while (i + j < n && sa[rk[i]-1] + j < n && s[i+j] == s[sa[rk[i]-1]+j])
1840 j++;
1841             lc[rk[i]-1] = j;
1842         }
1843     }
1844 };
1845 // 4. 后缀自动机：压缩存储子串
1846 struct SAM {
1847     struct Node { int len, link; array<int, ALPHABET> next; Node() : len(0), link(0),
1848 next{} {} };
1849     vector<Node> t;
1850     SAM() { t.assign(2, Node()); t[0].next.fill(1); t[0].len = -1; }
1851     int newNode() { t.emplace_back(); return t.size()-1; }
1852     int extend(int p, int c) {
1853         if (t[p].next[c]) {
1854             int q = t[p].next[c];
1855             if (t[q].len == t[p].len + 1) return q;
1856             int r = newNode(); t[r].len = t[p].len + 1;
1857             t[r].link = t[q].link; t[r].next = t[q].next;
1858             t[q].link = r;
1859             while (t[p].next[c] == q) { t[p].next[c] = r; p = t[p].link; }
1860             return r;
1861         }
1862         int cur = newNode(); t[cur].len = t[p].len + 1;
1863         while (!t[p].next[c]) { t[p].next[c] = cur; p = t[p].link; }
1864         t[cur].link = extend(p, c);
1865         return cur;
1866     }
1867     int extend(int p, char c) { return extend(p, (long long)(c - 'a')); }
1868 };
1869 // 5. 回文自动机：处理回文子串
1870 struct PAM {
1871     struct Node { int len, link, cnt; array<int, ALPHABET> next; Node() : len(0),
1872 link(0), cnt(0), next{} {} };
1873     vector<Node> t; int suff; string s;
1874     PAM() { t.assign(2, Node()); t[0].link = 1; t[1].len = -1; suff = 1; s.clear(); }
1875     int newNode() { t.emplace_back(); return t.size()-1; }
1876     bool add(char c) {
1877         int pos = s.size(); s += c; int let = c - 'a', cur = suff;
1878         while (true) {

```

```

1878         int curlen = t[cur].len;
1879         if (pos - 1 - curlen >= 0 && s[pos-1-curlen] == s[pos]) break;
1880         cur = t[cur].link;
1881     }
1882     if (t[cur].next[let]) { suff = t[cur].next[let]; return false; }
1883     int num = newNode(); suff = num; t[num].len = t[cur].len + 2; t[cur].next[let] =
num;
1884     if (t[num].len == 1) { t[num].link = 1; t[num].cnt = 1; return true; }
1885     while (true) {
1886         cur = t[cur].link; int curlen = t[cur].len;
1887         if (pos - 1 - curlen >= 0 && s[pos-1-curlen] == s[pos]) {
1888             t[num].link = t[cur].next[let]; break;
1889         }
1890     }
1891     t[num].cnt = 1 + t[t[num].link].cnt; return true;
1892 }
1893 };
1894
1895 // 6. AC自动机: 多模式匹配
1896 struct AhoCorasick {
1897     struct Node { int len, link; array<int, ALPHABET> next; Node() : len(0), link(0),
next{} {} };
1898     vector<Node> t;
1899     AhoCorasick() { t.assign(2, Node()); t[0].next.fill(1); t[0].len = -1; }
1900     int newNode() { t.emplace_back(); return t.size()-1; }
1901     int add(const string& a) {
1902         int p = 1;
1903         for (char c : a) {
1904             int x = c - 'a';
1905             if (!t[p].next[x]) { t[p].next[x] = newNode(); t[t[p].next[x]].len = t[p].len
+ 1; }
1906             p = t[p].next[x];
1907         }
1908         return p;
1909     }
1910     void work() {
1911         queue<int> q; q.push(1);
1912         while (!q.empty()) {
1913             int x = q.front(); q.pop();
1914             for (int i = 0; i < ALPHABET; i++) {
1915                 if (!t[x].next[i]) t[x].next[i] = t[t[x].link].next[i];
1916                 else { t[t[x].next[i]].link = t[t[x].link].next[i]; q.push(t[x].next[i]); }
1917             }
1918         }
1919     }
1920 };
1921
1922 // 7. 字符字典树: 前缀匹配
1923 struct CharTrie {
1924     static constexpr int MAX = 1e6 + 10;
1925     int trie[MAX][ALPHABET], val[MAX], tot;
1926     CharTrie() { tot = 0; newNode(); }

```

```
1927     int newNode() { tot++; memset(trie[tot], 0, sizeof(trie[tot])); val[tot] = 0; return
    tot; }
1928     void insert(const string& s) {
1929         int p = 1;
1930         for (char c : s) {
1931             int x = c - 'a';
1932             if (!trie[p][x]) trie[p][x] = newNode();
1933             p = trie[p][x]; val[p]++;
1934         }
1935     }
1936     int query(const string& s) {
1937         int p = 1;
1938         for (char c : s) {
1939             int x = c - 'a';
1940             if (!trie[p][x]) return 0;
1941             p = trie[p][x];
1942         }
1943         return val[p];
1944     }
1945 };
1946
1947 // 8. 二进制字典树：异或最值
1948 struct BinTrie {
1949     static constexpr int MAX = 1e7 + 10, BITS = 30;
1950     int trie[MAX][2], val[MAX], tot;
1951     BinTrie() { tot = 0; newNode(); }
1952     int newNode() { tot++; memset(trie[tot], 0, sizeof(trie[tot])); val[tot] = INF;
    return tot; }
1953     void insert(int x, int i) {
1954         int p = 1;
1955         for (int j = BITS; j >= 0; j--) {
1956             int b = (x >> j) & 1;
1957             if (!trie[p][b]) trie[p][b] = newNode();
1958             p = trie[p][b]; val[p] = min(val[p], i);
1959         }
1960     }
1961     int query_max_xor(int x) {
1962         int p = 1, res = INF;
1963         for (int j = BITS; j >= 0; j--) {
1964             int b = (x >> j) & 1;
1965             p = trie[p][1 - b] ? trie[p][1 - b] : trie[p][b];
1966             res = min(res, val[p]);
1967         }
1968         return res == INF ? -1 : res;
1969     }
1970 };
1971
1972
1973 // 9. KMP前缀函数：单模式匹配
1974 vector<int> kmp_prefix(const string& s) {
1975     int n = s.size(), j = 0;
1976     vector<int> f(n + 1);
1977     for (int i = 1; i < n; i++) {
1978         while (j && s[i] != s[j]) j = f[j];
```

```
1979         if (s[i] == s[j]) j++;
1980         f[i + 1] = j;
1981     }
1982     return f;
1983 }
1984
1985 vector<int> kmp_match(const string& text, const string& pat) {
1986     vector<int> f = kmp_prefix(pat), res;
1987     int j = 0;
1988     for (int i = 0; i < text.size(); i++) {
1989         while (j && text[i] != pat[j]) j = f[j];
1990         if (text[i] == pat[j]) j++;
1991         if (j == pat.size()) { res.push_back(i - j + 1); j = f[j]; }
1992     }
1993     return res;
1994 }
1995
1996
```