
Contents

1 Misc	2		
1.1 Contest	2	6.6 Point In Polygon	13
1.1.1 Makefile	2	6.6.1 Convex Version	13
1.2 How Did We Get Here?	2	6.6.2 Offline Multiple Points Version	14
1.2.1 Macros	2	6.7 Closest Pair	14
1.2.2 Fast I/O	2	6.8 Minimum Enclosing Circle	15
1.2.3 constexpr	2		
1.2.4 Bump Allocator	3	7 Strings	16
1.3 Tools	3	7.1 Knuth-Morris-Pratt Algorithm	16
1.3.1 Floating Point Binary Search	3	7.2 Z Value	16
1.3.2 SplitMix64	3	7.3 Manacher's Algorithm	16
1.3.3 <random>	3	7.4 Minimum Rotation	16
1.3.4 x86 Stack Hack	3	7.5 Aho-Corasick Automaton	16
1.4 Algorithms	3	8 Debug List	18
1.4.1 Bit Hacks	3		
1.4.2 Infinite Grid Knight Distance	3		
1.4.3 Longest Increasing Subsequence	3		
1.4.4 Mo's Algorithm on Tree	3		
2 Data Structures	5		
2.1 GNU PBDS	5		
2.2 2D Partial Sums	5		
2.3 Heavy-Light Decomposition	5		
3 Graph	6		
3.1 Modeling	6		
3.2 Matching/Flows	6		
3.2.1 Kuhn-Munkres algorithm	6		
3.3 Shortest Path Faster Algorithm	6		
3.4 Strongly Connected Components	7		
3.5 Biconnected Components	7		
3.5.1 Articulation Points	7		
3.5.2 Bridges	7		
3.6 Centroid Decomposition	7		
4 Math	9		
4.1 Number Theory	9		
4.1.1 Mod Struct	9		
4.1.2 Miller-Rabin	9		
4.1.3 Linear Sieve	9		
4.1.4 Get Factors	9		
4.1.5 Binary GCD	9		
4.1.6 Extended GCD	9		
4.1.7 Chinese Remainder Theorem	10		
4.1.8 Pollard's Rho	10		
4.1.9 Rational Number Binary Search	10		
4.2 Combinatorics	10		
4.2.1 De Bruijn Sequence	10		
5 Numeric	11		
5.1 Long Long Multiplication	11		
6 Geometry	12		
6.1 Point	12		
6.1.1 Quaternion	12		
6.1.2 Spherical Coordinates	12		
6.2 Segments	12		
6.3 Convex Hull	12		
6.3.1 3D Hull	13		
6.4 Angular Sort	13		
6.5 Convex Polygon Minkowski Sum	13		

1. Misc

1.1. Contest

1.1.1. Makefile

```
1 .PRECIOUS: ./p%
3 %: p%
   ulimit -s unlimited && ./$<
5 p%: p%.cpp
   g++ -o $@ $< -std=c++17 -Wall -Wextra -Wshadow \
7   -fsanitize=address,undefined
```

1.2. How Did We Get Here?

1.2.1. Macros

Use vectorizations and math optimizations at your own peril.
For gcc ≥ 9 , there are `[[likely]]` and `[[unlikely]]` attributes.
Call gcc with `-fopenmp-optimized-missed-optall` for optimization info.

```
1 #define _GLIBCXX_DEBUG 1 // for debug mode
2 #define _GLIBCXX_SANITIZE_VECTOR 1 // for asan on vector
3 #pragma GCC optimize("O3", "unroll-loops")
4 #pragma GCC optimize("fast-math")
5 #pragma GCC target("avx,avx2,abm,bmi,bmi2") // tip: use before a loop
6 // before a loop
7 #pragma GCC unroll 16 // 0 or 1 -> no unrolling
8 #pragma GCC ivdep
```

1.2.2. Fast I/O

```
1 struct scanner {
2     static constexpr size_t LEN = 32 << 20;
3     char *buf, *buf_ptr, *buf_end;
4     scanner()
5         : buf(new char[LEN]), buf_ptr(buf + LEN),
6           buf_end(buf + LEN) {}
7     ~scanner() { delete[] buf; }
8     char getc() {
9         if (buf_ptr == buf_end) [[unlikely]]
10            buf_end = buf + fread_unlocked(buf, 1, LEN, stdin);
11            buf_ptr = buf;
12            return *(buf_ptr++);
13        }
14        char seek(char del) {
15            char c;
16            while ((c = getc()) < del) {}
17            return c;
18        }
19        void read(int &t) {
20            bool neg = false;
21            char c = seek('-');
22            if (c == '-') neg = true, t = 0;
23            else t = c ^ '0';
24            while ((c = getc()) >= '0') t = t * 10 + (c ^ '0');
25            if (neg) t = -t;
26        }
27    };
28    struct printer {
29        static constexpr size_t CPI = 21, LEN = 32 << 20;
30        char *buf, *buf_ptr, *buf_end, *tbuf;
31        char *int_buf, *int_buf_end;
32        printer()
33            : buf(new char[LEN]), buf_ptr(buf),
34              buf_end(buf + LEN), int_buf(new char[CPI + 1]),
35              int_buf_end(int_buf + CPI - 1) {}
36        ~printer() {
37            flush();
38            delete[] buf, delete[] int_buf;
39        }
40        void flush() {
41            fwrite_unlocked(buf, 1, buf_ptr - buf, stdout);
42            buf_ptr = buf;
43        }
44    };
45    void write(const char &c) {
46        *buf_ptr = c;
47        if (++buf_ptr == buf_end) [[unlikely]]
48            flush();
49    }
50    void write(const char *s) {
51        for (; *s != '\0'; ++s) write(*s);
52    }
53    void write(int x) {
54        if (x < 0) write('-'), x = -x;
55        if (x == 0) [[unlikely]]
56            return write('0');
57        for (tbuf = int_buf_end; x != 0; --tbuf, x /= 10)
58            *tbuf = '0' + char(x % 10);
59        write(++tbuf);
60    }
61    };
62    };
63    };
64    };
65    };
66    };
67    };
68    };
69    };
70    };
71    };
72    };
73    };
74    };
75    };
76    };
77    };
78    };
79    };
80    };
81    };
82    };
83    };
84    };
85    };
86    };
87    };
88    };
89    };
90    };
91    };
92    };
93    };
94    };
95    };
96    };
97    };
98    };
99    };
100    };
101    };
102    };
103    };
104    };
105    };
106    };
107    };
108    };
109    };
110    };
111    };
112    };
113    };
114    };
115    };
116    };
117    };
118    };
119    };
120    };
121    };
122    };
123    };
124    };
125    };
126    };
127    };
128    };
129    };
130    };
131    };
132    };
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    };
161    };
162    };
163    };
164    };
165    };
166    };
167    };
168    };
169    };
170    };
171    };
172    };
173    };
174    };
175    };
176    };
177    };
178    };
179    };
180    };
181    };
182    };
183    };
184    };
185    };
186    };
187    };
188    };
189    };
190    };
191    };
192    };
193    };
194    };
195    };
196    };
197    };
198    };
199    };
200    };
201    };
202    };
203    };
204    };
205    };
206    };
207    };
208    };
209    };
210    };
211    };
212    };
213    };
214    };
215    };
216    };
217    };
218    };
219    };
220    };
221    };
222    };
223    };
224    };
225    };
226    };
227    };
228    };
229    };
230    };
231    };
232    };
233    };
234    };
235    };
236    };
237    };
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    };
269    };
270    };
271    };
272    };
273    };
274    };
275    };
276    };
277    };
278    };
279    };
280    };
281    };
282    };
283    };
284    };
285    };
286    };
287    };
288    };
289    };
290    };
291    };
292    };
293    };
294    };
295    };
296    };
297    };
298    };
299    };
300    };
301    };
302    };
303    };
304    };
305    };
306    };
307    };
308    };
309    };
310    };
311    };
312    };
313    };
314    };
315    };
316    };
317    };
318    };
319    };
320    };
321    };
322    };
323    };
324    };
325    };
326    };
327    };
328    };
329    };
330    };
331    };
332    };
333    };
334    };
335    };
336    };
337    };
338    };
339    };
340    };
341    };
342    };
343    };
344    };
345    };
346    };
347    };
348    };
349    };
350    };
351    };
352    };
353    };
354    };
355    };
356    };
357    };
358    };
359    };
360    };
361    };
362    };
363    };
364    };
365    };
366    };
367    };
368    };
369    };
370    };
371    };
372    };
373    };
374    };
375    };
376    };
377    };
378    };
379    };
380    };
381    };
382    };
383    };
384    };
385    };
386    };
387    };
388    };
389    };
390    };
391    };
392    };
393    };
394    };
395    };
396    };
397    };
398    };
399    };
400    };
401    };
402    };
403    };
404    };
405    };
406    };
407    };
408    };
409    };
410    };
411    };
412    };
413    };
414    };
415    };
416    };
417    };
418    };
419    };
420    };
421    };
422    };
423    };
424    };
425    };
426    };
427    };
428    };
429    };
430    };
431    };
432    };
433    };
434    };
435    };
436    };
437    };
438    };
439    };
440    };
441    };
442    };
443    };
444    };
445    };
446    };
447    };
448    };
449    };
450    };
451    };
452    };
453    };
454    };
455    };
456    };
457    };
458    };
459    };
460    };
461    };
462    };
463    };
464    };
465    };
466    };
467    };
468    };
469    };
470    };
471    };
472    };
473    };
474    };
475    };
476    };
477    };
478    };
479    };
480    };
481    };
482    };
483    };
484    };
485    };
486    };
487    };
488    };
489    };
490    };
491    };
492    };
493    };
494    };
495    };
496    };
497    };
498    };
499    };
500    };
501    };
502    };
503    };
504    };
505    };
506    };
507    };
508    };
509    };
510    };
511    };
512    };
513    };
514    };
515    };
516    };
517    };
518    };
519    };
520    };
521    };
522    };
523    };
524    };
525    };
526    };
527    };
528    };
529    };
530    };
531    };
532    };
533    };
534    };
535    };
536    };
537    };
538    };
539    };
540    };
541    };
542    };
543    };
544    };
545    };
546    };
547    };
548    };
549    };
550    };
551    };
552    };
553    };
554    };
555    };
556    };
557    };
558    };
559    };
560    };
561    };
562    };
563    };
564    };
565    };
566    };
567    };
568    };
569    };
570    };
571    };
572    };
573    };
574    };
575    };
576    };
577    };
578    };
579    };
580    };
581    };
582    };
583    };
584    };
585    };
586    };
587    };
588    };
589    };
590    };
591    };
592    };
593    };
594    };
595    };
596    };
597    };
598    };
599    };
600    };
601    };
602    };
603    };
604    };
605    };
606    };
607    };
608    };
609    };
610    };
611    };
612    };
613    };
614    };
615    };
616    };
617    };
618    };
619    };
620    };
621    };
622    };
623    };
624    };
625    };
626    };
627    };
628    };
629    };
630    };
631    };
632    };
633    };
634    };
635    };
636    };
637    };
638    };
639    };
640    };
641    };
642    };
643    };
644    };
645    };
646    };
647    };
648    };
649    };
650    };
651    };
652    };
653    };
654    };
655    };
656    };
657    };
658    };
659    };
660    };
661    };
662    };
663    };
664    };
665    };
666    };
667    };
668    };
669    };
670    };
671    };
672    };
673    };
674    };
675    };
676    };
677    };
678    };
679    };
680    };
681    };
682    };
683    };
684    };
685    };
686    };
687    };
688    };
689    };
690    };
691    };
692    };
693    };
694    };
695    };
696    };
697    };
698    };
699    };
700    };
701    };
702    };
703    };
704    };
705    };
706    };
707    };
708    };
709    };
710    };
711    };
712    };
713    };
714    };
715    };
716    };
717    };
718    };
719    };
720    };
721    };
722    };
723    };
724    };
725    };
726    };
727    };
728    };
729    };
730    };
731    };
732    };
733    };
734    };
735    };
736    };
737    };
738    };
739    };
740    };
741    };
742    };
743    };
744    };
745    };
746    };
747    };
748    };
749    };
750    };
751    };
752    };
753    };
754    };
755    };
756    };
757    };
758    };
759    };
760    };
761    };
762    };
763    };
764    };
765    };
766    };
767    };
768    };
769    };
770    };
771    };
772    };
773    };
774    };
775    };
776    };
777    };
778    };
779    };
780    };
781    };
782    };
783    };
784    };
785    };
786    };
787    };
788    };
789    };
790    };
791    };
792    };
793    };
794    };
795    };
796    };
797    };
798    };
799    };
800    };
801    };
802    };
803    };
804    };
805    };
806    };
807    };
808    };
809    };
810    };
811    };
812    };
813    };
814    };
815    };
816    };
817    };
818    };
819    };
820    };
821    };
822    };
823    };
824    };
825    };
826    };
827    };
828    };
829    };
830    };
831    };
832    };
833    };
834    };
835    };
836    };
837    };
838    };
839    };
840    };
841    };
842    };
843    };
844    };
845    };
846    };
847    };
848    };
849    };
850    };
851    };
852    };
853    };
854    };
855    };
856    };
857    };
858    };
859    };
860    };
861    };
862    };
863    };
864    };
865    };
866    };
867    };
868    };
869    };
870    };
871    };
872    };
873    };
874    };
875    };
876    };
877    };
878    };
879    };
880    };
881    };
882    };
883    };
884    };
885    };
886    };
887    };
888    };
889    };
890    };
891    };
892    };
893    };
894    };
895    };
896    };
897    };
898    };
899    };
900    };
901    };
902    };
903    };
904    };
905    };
906    };
907    };
908    };
909    };
910    };
911    };
912    };
913    };
914    };
915    };
916    };
917    };
918    };
919    };
920    };
921    };
922    };
923    };
924    };
925    };
926    };
927    };
928    };
929    };
930    };
931    };
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    };
970    };
971    };
972    };
973    };
974    };
975    };
976    };
977    };
978    };
979    };
980    };
981    };
982    };
983    };
984    };
985    };
986    };
987    };
988    };
989    };
990    };
991    };
992    };
993    };
994    };
995    };
996    };
997    };
998    };
999    };
1000    };
1001    };
1002    };
1003    };
1004    };
1005    };
1006    };
1007    };
1008    };
1009    };
1010    };
1011    };
1012    };
1013    };
1014    };
1015    };
1016    };
1017    };
1018    };
1019    };
1020    };
1021    };
1022    };
1023    };
1024    };
1025    };
1026    };
1027    };
1028    };
1029    };
1030    };
1031    };
1032    };
1033    };
1034    };
1035    };
1036    };
1037    };
1038    };
1039    };
1040    };
1041    };
1042    };
1043    };
1044    };
1045    };
1046    };
1047    };
1048    };
1049    };
1050    };
1051    };
1052    };
1053    };
1054    };
1055    };
1056    };
1057    };
1058    };
1059    };
1060    };
1061    };
1062    };
1063    };
1064    };
1065    };
1066    };
1067    };
1068    };
1069    };
1070    };
1071    };
1072    };
1073    };
1074    };
1075    };
1076    };
1077    };
1078    };
1079    };
1080    };
1081    };
1082    };
1083    };
1084    };
1085    };
1086    };
1087    };
1088    };
1089    };
1090    };
1091    };
1092    };
1093    };
1094    };
1095    };
1096    };
1097    };
1098    };
1099    };
1100    };
1101    };
1102    };
1103    };
1104    };
1105    };
1106    };
1107    };
1108    };
1109    };
1110    };
1111    };
1112    };
1113    };
1114    };
1115    };
1116    };
1117    };
1118    };
1119    };
1120    };
1121    };
1122    };
1123    };
1124    };
1125    };
1126    };
1127    };
1128    };
1129    };
1130    };
1131    };
1132    };
1133    };
1134    };
1135    };
1136    };
1137    };
1138    };
1139    };
1140    };
1141    };
1142    };
1143    };
1144    };
1145    };
1146    };
1147    };
1148    };
1149    };
1150    };
1151    };
1152    };
1153    };
1154    };
1155    };
1156    };
1157    };
1158    };
1159    };
1160    };
1161    };
1162    };
1163    };
1164    };
1165    };
1166    };
1167    };
1168    };
1169    };
1170    };
1171    };
1172    };
1173    };
1174    };
1175    };
1176    };
1177    };
1178    };
1179    };
1180    };
1181    };
1182    };
1183    };
1184    };
1185    };
1186    };
1187    };
1188    };
1189    };
1190    };
1191    };
1192    };
1193    };
1194    };
1195    };
1196    };
1197    };
1198    };
1199    };
1200    };
1201    };
1202    };
1203    };
1204    };
1205    };
1206    };
1207    };
1208    };
1209    };
1210    };
1211    };
1212    };
1213    };
1214    };
1215    };
1216    };
1217    };
1218    };
1219    };
1220    };
1221    };
1222    };
1223    };
1224    };
1225    };
1226    };
1227    };
1228    };
1229    };
1230    };
1231    };
1232    };
1233    };
1234    };
1235    };
1236    };
1237    };
1238    };
1239    };
1240    };
1241    };
1242    };
1243    };
1244    };
1245    };
1246    };
1247    };
1248    };
1249    };
1250    };
1251    };
1252    };
1253    };
1254    };
1255    };
1256    };
1257    };
1258    };
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    };
1291    };
1292    };
1293    };
1294    };
1295    };
1296    };
1297    };
1298    };
1299    };
1300    };
1301    };
1302    };
1303    };
1304    };
1305    };
1306    };
1307    };
1308    };
1309    };
1310    };
1311    };
1312    };
1313    };
1314    };
1315    };
1316    };
1317    };
1318    };
1319    };
1320    };
1321    };
1322    };
1323    };
1324    };
1325    };
1326    };
1327    };
1328    };
1329    };
1330    };
1331    };
1332    };
1333    };
1334    };
1335    };
1336    };
1337    };
1338    };
1339    };
1340    };
1341    };
1342    };
1343    };
1344    };
1345    };
1346    };
1347    };
1348    };
1349    };
1350    };
1351    };
1352    };
1353    };
1354    };
1355    };
1356    };
1357    };
1358    };
1359    };
1360    };
1361    };
1362    };
1363    };
1364    };
1365    };
1366    };
1367    };
1368    };
1369    };
1370    };
1371    };
1372    };
1373    };
1374    };
1375    };
1376    };
1377    };
1378    };
1379    };
1380    };
1381    };
1382    };
1383    };
1384    };
1385    };
1386    };
1387    };
1388    };
1389    };
1390    };
1391    };
1392    };
1393    };
1394    };
1395    };
1396    };
1397    };
1398    };
1399    };
1400    };
1401    };
1402    };
1403    };
1404    };
1405    };
1406    };
1407    };
1408    };
1409    };
1410    };
1411    };
1412    };
1413    };
1414    };
1415    };
1416    };
1417    };
1418    };
1419    };
1420    };
1421    };
1422    };
1423    };
1424    };
1425    };
1426    };
1427    };
1428    };
1429    };
1430    };
1431    };
1432    };
1433    };
1434    };
1435    };
1436    };
1437    };
1438    };
1439    };
1440    };
1441    };
1442    };
1443    };
1444    };
1445    };
1446    };
1447    };
1448    };
1449    };
1450    };
1451    };
1452    };
1453    };
1454    };
1455    };
1456    };
1457    };
1458    };
1459    };
1460    };
1461    };
1462    };
1463    };
1464    };
1465    };
1466    };
1467    };
1468    };
1469    };
1470    };
1471    };
1472    };
1473    };
1474    };
1475    };
1476    };
1477    };
1478    };
1479    };
1480    };
1481    };
1482    };
1483    };
1484    };
1485    };
1486    };
1487    };
1488    };
1489    };
1490    };
1491    };
1492    };
1493    };
1494    };
1495    };
1496    };
1497    };
1498    };
1499    };
1500    };
1501    };
1502    };
1503    };
1504    };
1505    };
1506    };
1507    };
1508    };
1509    };
1510    };
1511    };
1512    };
1513    };
1514    };
1515    };
1516    };
1517    };
1518    };
1519    };
1520    };
1521    };
1522    };
1523    };
1524    };
1525    };
1526    };
1527    };
1528    };
1529    };
1530    };
1531    };
1532    };
1533    };
1534    };
1535    };
1536    };
1537    };
1538    };
1539    };
1540    };
1541    };
1542    };
1543    };
1544    };
1545    };
1546    };
1547    };
1548    };
1549    };
1550    };
1551    };
1552    };
1553    };
1554    };
1555    };
1556    };
1557    };
1558    };
1559    };
1560    };
1561    };
1562    };
1563    };
1564    };
1565    };
1566    };
1567    };
1568    };
1569    };
1570    };
1571    };
1572    };
1573    };
1574    };
1575    };
1576    };
1577    };
1578    };
1579    };
1580    };
1581    };
1582    };
1583    };
1584    };
1585    };
1586    };
1587    };
1588    };
1589    };
1590    };
1591    };
1592    };
1593    };
1594    };
1595    };
1596    };
1597    };
1598    };
1599    };
1600    };
1601    };
1602    };
1603    };
1604    };
1605    };
1606    };
1607    };
1608    };
1609    };
1610    };
1611    };
1612    };
1613    };
1614    };
1615    };
1616    };
1617    };
1618    };
1619    };
1620    };
1621    };
1622    };
1623    };
1624    };
1625    };
1626    };
1627    };
1628    };
1629    };
1630    };
1631    };
1632    };
1633    };
1634    };
1635    };
1636    };
1637    };
1638    };
1639    };
1640    };
1641    };
1642    };
1643    };
1644    };
1645    };
1646    };
1647    };
1648    };
1649    };
1650    };
1651    };
1652    };
1653    };
1654    };
1655    };
1656    };
1657    };
1658    };
1659    };
1660    };
1661    };
1662    };
1663    };
1664    };
1665    };
1666    };
1667    };
1668    };
1669    };
1670    };
1671    };
1672    };
1673    };
1674    };
1675    };
1676    };
1677    };
1678    };
1679    };
1680    };
1681    };
1682    };
1683    };
1684    };
1685    };
1686    };
1687    };
1688    };
1689    };
1690    };
1691    };
1692    };
1693    };
1694    };
1695    };
1696    };
1697    };
1698    };
1699    };
1700    };
1701    };
1702    };
1703    };
1704    };
1705    };
1706    };
1707    };
1708    };
1709    };
1710    };
1711    };
1712    };
1713    };
1714    };
1715    };
1716    };
1717    };
1718    };
1719    };
1720    };
1721    };
1722    };
1723    };
1724    };
1725    };
1726    };
1727    };
1728    };
1729    };
1730    };
1731    };
1732    };
1733    };
1734    };
1735    };
1736    };
1737    };
1738    };
1739    };
1740    };
1741    };
1742    };
1743    };
1744    };
1745    };
1746    };
1747    };
1748    };
1749    };
1750    };
1751    };
1752    };
1753    };
1754    };
1755    };
1756   
```

1.2.4. Bump Allocator

```

1 // global bump allocator
3 char mem[256 << 20]; // 256 MB
5 size_t rsp = sizeof mem;
7 void *operator new(size_t s) {
8     assert(s < rsp); // MLE
9     return (void *)&mem[rsp -= s];
11 }
13 void operator delete(void *) {}
15 // bump allocator for STL / pbds containers
17 char mem[256 << 20];
19 size_t rsp = sizeof mem;
21 template <typename T> struct bump {
22     typedef T value_type;
23     bump() {}
24     template <typename U> bump(U, ...) {}
25     T *allocate(size_t n) {
26         rsp -= n * sizeof(T);
27         rsp &= 0 - alignof(T);
28         return (T *) (mem + rsp);
29     }
30     void deallocate(T *, size_t n) {}
31 };

```

1.3. Tools

1.3.1. Floating Point Binary Search

```

1 union di {
2     double d;
3     ull i;
4 };
5 bool check(double);
6 // binary search in [L, R] with relative error 2^-eps
7 double binary_search(double L, double R, int eps) {
8     di l = {L}, r = {R}, m;
9     while (r.i - l.i > 1LL << (52 - eps)) {
10         m.i = (l.i + r.i) >> 1;
11         if (check(m.d)) r = m;
12         else l = m;
13     }
14     return l.d;
15 }

```

1.3.2. SplitMix64

```

1 using ull = unsigned long long;
2 inline ull splitmix64(ull x) {
3     // change to `static ull x = SEED;` for DRBG
4     ull z = (x += 0x9E3779B97F4A7C15);
5     z = (z ^ (z >> 30)) * 0xBF58476D1CE4E5B9;
6     z = (z ^ (z >> 27)) * 0x94D049BB133111EB;
7     return z ^ (z >> 31);
8 }

```

1.3.3. <random>

```

1 #ifdef __unix__
2 random_device rd;
3 mt19937_64 RNG(rd());
4 #else
5 const auto SEED = chrono::high_resolution_clock::now()
6     .time_since_epoch()
7     .count();
8 mt19937_64 RNG(SEED);
9 #endif
10 // random uint_fast64_t: RNG();
11 // uniform random of type T (int, double, ...) in [l, r]:
12 // uniform_int_distribution<T> dist(l, r); dist(RNG);

```

1.3.4. x86 Stack Hack

```

1 constexpr size_t size = 200 << 20; // 200MiB
2 int main() {
3     register long rsp asm("rsp");
4     char *buf = new char[size];
5     asm("movq %0, %%rsp\n" :: "r" (buf + size));
6     // do stuff
7     asm("movq %0, %%rsp\n" :: "r" (rsp));
8     delete[] buf;
9 }

```

1.4. Algorithms

1.4.1. Bit Hacks

```

1 // next permutation of x as a bit sequence
2 ull next_bits_permutation(ull x) {
3     ull c = __builtin_ctzll(x), r = x + (1ULL << c);
4     return (r ^ x) >> (c + 2) | r;
5 }
6 // iterate over all (proper) subsets of bitset s
7 void subsets(ull s) {
8     for (ull x = s; x; x--) { --x &= s; /* do stuff */ }
9 }

```

1.4.2. Infinite Grid Knight Distance

```

1 ll get_dist(ll dx, ll dy) {
2     if (++(dx = abs(dx)) > ++(dy = abs(dy))) swap(dx, dy);
3     if (dx == 1 && dy == 2) return 3;
4     if (dx == 3 && dy == 3) return 4;
5     ll lb = max(dy / 2, (dx + dy) / 3);
6     return ((dx ^ dy ^ lb) & 1) ? ++lb : lb;
7 }

```

1.4.3. Longest Increasing Subsequence

```

1 template <class I> vi lis(const vector<I> &S) {
2     if (S.empty()) return {};
3     vi prev(sz(S));
4     typedef pair<I, int> p;
5     vector<p> res;
6     rep(i, 0, sz(S)) {
7         // change 0 -> i for longest non-decreasing subsequence
8         auto it = lower_bound(all(res), p{S[i], 0});
9         if (it == res.end())
10             res.emplace_back(), it = res.end() - 1;
11         *it = {S[i], i};
12         prev[i] = it == res.begin() ? 0 : (it - 1)->second;
13     }
14     int L = sz(res), cur = res.back().second;
15     vi ans(L);
16     while (L--) ans[L] = cur, cur = prev[cur];
17     return ans;
18 }

```

1.4.4. Mo's Algorithm on Tree

```

1 void MoAlgoOnTree() {
2     Dfs(0, -1);
3     vector<int> euler(tk);
4     for (int i = 0; i < n; ++i) {
5         euler[tin[i]] = i;
6         euler[tout[i]] = i;
7     }
8     vector<int> l(q), r(q), qr(q), sp(q, -1);
9     for (int i = 0; i < q; ++i) {
10         if (tin[u[i]] > tin[v[i]]) swap(u[i], v[i]);
11         int z = GetLCA(u[i], v[i]);
12         sp[i] = z[i];
13         if (z == u) l[i] = tin[u[i]], r[i] = tin[v[i]];
14         else l[i] = tout[u[i]], r[i] = tin[v[i]];
15         qr[i] = i;
16     }
17 }

```

```
17     }
18     sort(qr.begin(), qr.end(), [&](int i, int j) {
19         if (l[i] / kB == l[j] / kB) return r[i] < r[j];
20         return l[i] / kB < l[j] / kB;
21     });
22     vector<bool> used(n);
23     // Add(v): add/remove v to/from the path based on used[v]
24     for (int i = 0, tl = 0, tr = -1; i < q; ++i) {
25         while (tl < l[qr[i]]) Add(euler[tl++]);
26         while (tl > l[qr[i]]) Add(euler[--tl]);
27         while (tr > r[qr[i]]) Add(euler[tr--]);
28         while (tr < r[qr[i]]) Add(euler[++tr]);
29         // add/remove LCA(u, v) if necessary
30     }
31 }
```

2. Data Structures

2.1. GNU PBDS

```

1 #include <ext/pb_ds/assoc_container.hpp>
2 #include <ext/pb_ds/priority_queue.hpp>
3 #include <ext/pb_ds/tree_policy.hpp>
4 using namespace __gnu_pbds;
5
6 // most std::map + order_of_key, find_by_order, split, join
7 template <typename T, typename U = null_type>
8 using ordered_map = tree<T, U, std::less<>, rb_tree_tag,
9 tree_order_statistics_node_update>;
10 // useful tags: rb_tree_tag, splay_tree_tag
11
12 template <typename T> struct myhash {
13     size_t operator()(T x) const; // splitmix, bswap(x*R), ...
14 };
15 // most of std::unordered_map, but faster (needs good hash)
16 template <typename T, typename U = null_type>
17 using hash_table = gp_hash_table<T, U, myhash<T>>;
18
19 // most std::priority_queue + modify, erase, split, join
20 using heap = priority_queue<int, std::less<>>;
21 // useful tags: pairing_heap_tag, binary_heap_tag,
22 // (rc_)?binomial_heap_tag, thin_heap_tag

```

2.2. 2D Partial Sums

```

1 using vvi = vector<vector<int>>;
2 using vvl = vector<vector<ll>>;
3 using vll = vector<ll>;
4
5 struct PrefixSum2D {
6     vll pref; // 0-based 2-D prefix sum
7     void build(const vll &v) { // creates a copy
8         int n = v.size(), m = v[0].size();
9         pref.assign(n, vll(m, 0));
10         for (int i = 0; i < n; i++) {
11             for (int j = 0; j < m; j++) {
12                 pref[i][j] = v[i][j] + (i ? pref[i - 1][j] : 0) +
13                     (j ? pref[i][j - 1] : 0) -
14                     (i && j ? pref[i - 1][j - 1] : 0);
15             }
16         }
17     }
18     ll query(int ulx, int uly, int brx, int bry) const {
19         ll ans = pref[brx][bry];
20         if (ulx) ans -= pref[ulx - 1][bry];
21         if (uly) ans -= pref[brx][uly - 1];
22         if (ulx && uly) ans += pref[ulx - 1][uly - 1];
23         return ans;
24     }
25     ll query(int ulx, int uly, int size) const {
26         return query(ulx, uly, ulx + size - 1, uly + size - 1);
27     }
28 };
29 struct PartialSum2D : PrefixSum2D {
30     vll diff; // 0 based
31     int n, m;
32     PartialSum2D(int _n, int _m) : n(_n), m(_m) {
33         diff.assign(n + 1, vll(m + 1, 0));
34     }
35     // add c from [ulx,uly] to [brx,bry]
36     void update(int ulx, int uly, int brx, int bry, ll c) {
37         diff[ulx][uly] += c;
38         diff[ulx][bry + 1] -= c;
39         diff[brx + 1][uly] -= c;
40         diff[brx + 1][bry + 1] += c;
41     }
42     void update(int ulx, int uly, int size, ll c) {
43         int brx = ulx + size - 1;
44         int bry = uly + size - 1;
45         update(ulx, uly, brx, bry, c);
46     }
47     // process the grid using prefix sum

```

```

    void process() { this->build(diff); }
49 };
50 // usage
51 PrefixSum2D pref;
52 pref.build(v); // takes 2d 0-based vector as input
53 pref.query(x1, y1, x2, y2); // sum of region
54
55 PartialSum2D part(n, m); // dimension of grid 0 based
56 part.update(x1, y1, x2, y2, 1); // add 1 in region
57 // must run after all updates
58 part.process(); // prefix sum on diff array
59 // only exists after processing
60 vll &grid = part.pref; // processed diff array
61 part.query(x1, y1, x2, y2); // gives sum of region

```

2.3. Heavy-Light Decomposition

```

1 hash
2 template <bool VALS_EDGES> struct HLD {
3     int N, tim = 0;
4     vector<vi> adj;
5     vi par, siz, depth, rt, pos;
6     Node *tree;
7     HLD(vector<vi> adj_)
8         : N(sz(adj_)), adj(adj_), par(N, -1), siz(N, 1),
9           depth(N), rt(N), pos(N), tree(new Node(0, N)) {}
10     dfsSz(0);
11     dfsHld(0);
12 }
13 void dfsSz(int v) {
14     if (par[v] != -1)
15         adj[v].erase(find(all(adj[v]), par[v]));
16     for (int &u : adj[v]) {
17         par[u] = v, depth[u] = depth[v] + 1;
18         dfsSz(u);
19         siz[v] += siz[u];
20         if (siz[u] > siz[adj[v][0]]) swap(u, adj[v][0]);
21     }
22 }
23 void dfsHld(int v) {
24     pos[v] = tim++;
25     for (int u : adj[v]) {
26         rt[u] = (u == adj[v][0] ? rt[v] : u);
27         dfsHld(u);
28     }
29 }
30 template <class B> void process(int u, int v, B op) {
31     for (; rt[u] != rt[v]; v = par[rt[v]]) {
32         if (depth[rt[u]] > depth[rt[v]]) swap(u, v);
33         op(pos[rt[v]], pos[v] + 1);
34     }
35     if (depth[u] > depth[v]) swap(u, v);
36     op(pos[u] + VALS_EDGES, pos[v] + 1);
37 }
38 void modifyPath(int u, int v, int val) {
39     process(u, v, [&](int l, int r) { tree->add(l, r, val); });
40 }
41 int queryPath(int u, int v) { // Modify depending on problem
42     int res = -1e9;
43     process(u, v, [&](int l, int r) {
44         res = max(res, tree->query(l, r));
45     });
46     return res;
47 }
48 int querySubtree(int v) { // modifySubtree is similar
49     return tree->query(pos[v] + VALS_EDGES, pos[v] + siz[v]);
50 }
51 };

```

3. Graph

3.1. Modeling

- Maximum/Minimum flow with lower bound / Circulation problem
 - Construct super source S and sink T .
 - For each edge (x, y, l, u) , connect $x \rightarrow y$ with capacity $u - l$.
 - For each vertex v , denote by $in(v)$ the difference between the sum of incoming lower bounds and the sum of outgoing lower bounds.
 - If $in(v) > 0$, connect $S \rightarrow v$ with capacity $in(v)$, otherwise, connect $v \rightarrow T$ with capacity $-in(v)$.
 - To maximize, connect $t \rightarrow s$ with capacity ∞ (skip this in circulation problem), and let f be the maximum flow from S to T . If $f \neq \sum_{v \in V, in(v) > 0} in(v)$, there's no solution. Otherwise, the maximum flow from s to t is the answer.
 - To minimize, let f be the maximum flow from S to T . Connect $t \rightarrow s$ with capacity ∞ and let the flow from S to T be f' . If $f + f' \neq \sum_{v \in V, in(v) > 0} in(v)$, there's no solution. Otherwise, f' is the answer.
 - The solution of each edge e is $l_e + f_e$, where f_e corresponds to the flow of edge e on the graph.
- Construct minimum vertex cover from maximum matching M on bipartite graph (X, Y)
 - Redirect every edge: $y \rightarrow x$ if $(x, y) \in M$, $x \rightarrow y$ otherwise.
 - DFS from unmatched vertices in X .
 - $x \in X$ is chosen iff x is unvisited.
 - $y \in Y$ is chosen iff y is visited.
- Minimum cost cyclic flow
 - Construct super source S and sink T
 - For each edge (x, y, c) , connect $x \rightarrow y$ with $(cost, cap) = (c, 1)$ if $c > 0$, otherwise connect $y \rightarrow x$ with $(cost, cap) = (-c, 1)$
 - For each edge with $c < 0$, sum these cost as K , then increase $d(y)$ by 1, decrease $d(x)$ by 1
 - For each vertex v with $d(v) > 0$, connect $S \rightarrow v$ with $(cost, cap) = (0, d(v))$
 - For each vertex v with $d(v) < 0$, connect $v \rightarrow T$ with $(cost, cap) = (0, -d(v))$
 - Flow from S to T , the answer is the cost of the flow $C + K$
- Maximum density induced subgraph
 - Binary search on answer, suppose we're checking answer T
 - Construct a max flow model, let K be the sum of all weights
 - Connect source $s \rightarrow v, v \in G$ with capacity K
 - For each edge (u, v, w) in G , connect $u \rightarrow v$ and $v \rightarrow u$ with capacity w
 - For $v \in G$, connect it with sink $v \rightarrow t$ with capacity $K + 2T - (\sum_{e \in E(v)} w(e)) - 2w(v)$
 - T is a valid answer if the maximum flow $f < K|V|$
- Minimum weight edge cover
 - For each $v \in V$ create a copy v' , and connect $u' \rightarrow v'$ with weight $w(u, v)$.
 - Connect $v \rightarrow v'$ with weight $2\mu(v)$, where $\mu(v)$ is the cost of the cheapest edge incident to v .
 - Find the minimum weight perfect matching on G' .
- Project selection problem
 - If $p_v > 0$, create edge (s, v) with capacity p_v ; otherwise, create edge (v, t) with capacity $-p_v$.
 - Create edge (u, v) with capacity w with w being the cost of choosing u without choosing v .
 - The mincut is equivalent to the maximum profit of a subset of projects.
- 0/1 quadratic programming

$$\sum_x c_x x + \sum_y c_y \bar{y} + \sum_{xy} c_{xy} x \bar{y} + \sum_{xyx'y'} c_{xyx'y'} (x \bar{y} + x' \bar{y}')$$

can be minimized by the mincut of the following graph:

- Create edge (x, t) with capacity c_x and create edge (s, y) with capacity c_y .
- Create edge (x, y) with capacity c_{xy} .
- Create edge (x, y) and edge (x', y') with capacity $c_{xyx'y'}$.

3.2. Matching/Flows

3.2.1. Kuhn-Munkres algorithm

```

1 // Maximum Weight Perfect Bipartite Matching
2 // Detect non-perfect-matching:
3 // 1. set all edge[i][j] as INF
4 // 2. if solve() >= INF, it is not perfect matching.
5
6 typedef long long ll;
7 struct KM {
8     static const int MAXN = 1050;
9     static const ll INF = 1LL << 60;
10    int n, match[MAXN], vx[MAXN], vy[MAXN];
11    ll edge[MAXN][MAXN], lx[MAXN], ly[MAXN], slack[MAXN];
12    void init(int _n) {
13        n = _n;
14        for (int i = 0; i < n; i++)
15            for (int j = 0; j < n; j++) edge[i][j] = 0;
16    }
17    void add_edge(int x, int y, ll w) { edge[x][y] = w; }
18    bool DFS(int x) {
19        vx[x] = 1;
20        for (int y = 0; y < n; y++) {
21            if (vy[y]) continue;
22            if (lx[x] + ly[y] > edge[x][y]) {
23                slack[y] = min(slack[y], lx[x] + ly[y] - edge[x][y]);
24            } else {
25                vy[y] = 1;
26                if (match[y] == -1 || DFS(match[y])) {
27                    match[y] = x;
28                    return true;
29                }
30            }
31        }
32        return false;
33    }
34    ll solve() {
35        fill(match, match + n, -1);
36        fill(lx, lx + n, -INF);
37        fill(ly, ly + n, 0);
38        for (int i = 0; i < n; i++)
39            for (int j = 0; j < n; j++)
40                lx[i] = max(lx[i], edge[i][j]);
41        for (int i = 0; i < n; i++) {
42            fill(slack, slack + n, INF);
43            while (true) {
44                fill(vx, vx + n, 0);
45                fill(vy, vy + n, 0);
46                if (DFS(i)) break;
47                ll d = INF;
48                for (int j = 0; j < n; j++)
49                    if (!vy[j]) d = min(d, slack[j]);
50                for (int j = 0; j < n; j++) {
51                    if (vx[j]) lx[j] -= d;
52                    if (vy[j]) ly[j] += d;
53                    else slack[j] -= d;
54                }
55            }
56        }
57        ll res = 0;
58        for (int i = 0; i < n; i++) {
59            res += edge[match[i]][i];
60        }
61        return res;
62    }
63 } graph;

```

3.3. Shortest Path Faster Algorithm

```

1 struct SPFA {
2     static const int maxn = 1010, INF = 1e9;
3     int dis[maxn];
4     bitset<maxn> inq, inneg;
5     queue<int> q, tq;
6     vector<pii> v[maxn];

```

```

7 void make_edge(int s, int t, int w) {
9     v[s].emplace_back(t, w);
10 }
11 void dfs(int a) {
12     inneg[a] = 1;
13     for (pii i : v[a])
14         if (!inneg[i.F]) dfs(i.F);
15 }
16 bool solve(int n, int s) { // true if have neg-cycle
17     for (int i = 0; i <= n; i++) dis[i] = INF;
18     dis[s] = 0, q.push(s);
19     for (int i = 0; i < n; i++) {
20         inq.reset();
21         int now;
22         while (!q.empty()) {
23             now = q.front(), q.pop();
24             for (pii &i : v[now]) {
25                 if (dis[i.F] > dis[now] + i.S) {
26                     dis[i.F] = dis[now] + i.S;
27                     if (!inq[i.F]) tq.push(i.F), inq[i.F] = 1;
28                 }
29             }
30             q.swap(tq);
31         }
32         bool re = !q.empty();
33         inneg.reset();
34         while (!q.empty()) {
35             if (!inneg[q.front()]) dfs(q.front());
36             q.pop();
37         }
38         return re;
39     }
40 void reset(int n) {
41     for (int i = 0; i <= n; i++) v[i].clear();
42 }
43 };

```

3.4. Strongly Connected Components

```

1 struct TarjanScc {
2     int n, step;
3     vector<int> time, low, instk, stk;
4     vector<vector<int>> e, scc;
5     TarjanScc(int n_)
6         : n(n_), step(0), time(n), low(n), instk(n), e(n) {}
7     void add_edge(int u, int v) { e[u].push_back(v); }
8     void dfs(int x) {
9         time[x] = low[x] = ++step;
10        stk.push_back(x);
11        instk[x] = 1;
12        for (int y : e[x])
13            if (!time[y]) {
14                dfs(y);
15                low[x] = min(low[x], low[y]);
16            } else if (instk[y]) {
17                low[x] = min(low[x], time[y]);
18            }
19        if (time[x] == low[x]) {
20            scc.emplace_back();
21            for (int y = -1; y != x; ) {
22                y = stk.back();
23                stk.pop_back();
24                instk[y] = 0;
25                scc.back().push_back(y);
26            }
27        }
28    }
29    void solve() {
30        for (int i = 0; i < n; i++)
31            if (!time[i]) dfs(i);
32        reverse(scc.begin(), scc.end());
33        // scc in topological order
34    }
35 };

```

3.5. Biconnected Components

3.5.1. Articulation Points

```

1 void dfs(int x, int p) {
2     tin[x] = low[x] = ++t;
3     int ch = 0;
4     for (auto u : g[x])
5         if (u.first != p) {
6             if (!ins[u.second])
7                 st.push(u.second), ins[u.second] = true;
8             if (tin[u.first]) {
9                 low[x] = min(low[x], tin[u.first]);
10                continue;
11            }
12            ++ch;
13            dfs(u.first, x);
14            low[x] = min(low[x], low[u.first]);
15            if (low[u.first] >= tin[x]) {
16                cut[x] = true;
17                ++sz;
18                while (true) {
19                    int e = st.top();
20                    st.pop();
21                    bcc[e] = sz;
22                    if (e == u.second) break;
23                }
24            }
25        }
26    if (ch == 1 && p == -1) cut[x] = false;
27 }

```

3.5.2. Bridges

```

1 // if there are multi-edges, then they are not bridges
2 void dfs(int x, int p) {
3     tin[x] = low[x] = ++t;
4     st.push(x);
5     for (auto u : g[x])
6         if (u.first != p) {
7             if (tin[u.first]) {
8                 low[x] = min(low[x], tin[u.first]);
9                 continue;
10            }
11            dfs(u.first, x);
12            low[x] = min(low[x], low[u.first]);
13            if (low[u.first] == tin[u.first]) br[u.second] = true;
14        }
15    if (tin[x] == low[x]) {
16        ++sz;
17        while (st.size()) {
18            int u = st.top();
19            st.pop();
20            bcc[u] = sz;
21            if (u == x) break;
22        }
23    }
24 }

```

3.6. Centroid Decomposition

```

1 void get_center(int now) {
2     v[now] = true;
3     vtx.push_back(now);
4     sz[now] = 1;
5     mx[now] = 0;
6     for (int u : G[now])
7         if (!v[u]) {
8             get_center(u);
9             mx[now] = max(mx[now], sz[u]);
10            sz[now] += sz[u];
11        }
12 }
13 void get_dis(int now, int d, int len) {
14     dis[d][now] = cnt;
15     v[now] = true;

```

```
17     for (auto u : G[now])
18         if (!v[u.first]) { get_dis(u, d, len + u.second); }
19 void dfs(int now, int fa, int d) {
20     get_center(now);
21     int c = -1;
22     for (int i : vtx) {
23         if (max(mx[i], (int)vtx.size() - sz[i]) <=
24             (int)vtx.size() / 2)
25             c = i;
26         v[i] = false;
27     }
28     get_dis(c, d, 0);
29     for (int i : vtx) v[i] = false;
30     v[c] = true;
31     vtx.clear();
32     dep[c] = d;
33     p[c] = fa;
34     for (auto u : G[c])
35         if (u.first != fa && !v[u.first]) {
36             dfs(u.first, c, d + 1);
37         }
38 }
```


4. Math

4.1. Number Theory

4.1.1. Mod Struct

A list of safe primes: 26003, 27767, 28319, 28979, 29243, 29759, 30467, 31009, 31669, 32311, 33017, 33739, 34471, 35219, 35971, 36739, 37511, 38291, 39083, 39887, 40699, 41521, 42353, 43191, 44041, 44899, 45767, 46643, 47527, 48419, 49319, 50227, 51143, 52067, 52999, 53939, 54887, 55841, 56799, 57767, 58741, 59721, 60709, 61699, 62699, 63709, 64727, 65751, 66779, 67817, 68863, 69917, 70979, 72047, 73119, 74197, 75281, 76369, 77463, 78563, 79669, 80779, 81899, 83019, 84143, 85269, 86399, 87531, 88667, 89807, 90949, 92093, 93239, 94387, 95537, 96687, 97839, 98991, 100143, 101297, 102453, 103609, 104767, 105927, 107087, 108247, 109407, 110567, 111727, 112887, 114047, 115207, 116367, 117527, 118687, 119847, 121007, 122167, 123327, 124487, 125647, 126807, 127967, 129127, 130287, 131447, 132607, 133767, 134927, 136087, 137247, 138407, 139567, 140727, 141887, 143047, 144207, 145367, 146527, 147687, 148847, 150007, 151167, 152327, 153487, 154647, 155807, 156967, 158127, 159287, 160447, 161607, 162767, 163927, 165087, 166247, 167407, 168567, 169727, 170887, 172047, 173207, 174367, 175527, 176687, 177847, 179007, 180167, 181327, 182487, 183647, 184807, 185967, 187127, 188287, 189447, 190607, 191767, 192927, 194087, 195247, 196407, 197567, 198727, 199887, 201047, 202207, 203367, 204527, 205687, 206847, 208007, 209167, 210327, 211487, 212647, 213807, 214967, 216127, 217287, 218447, 219607, 220767, 221927, 223087, 224247, 225407, 226567, 227727, 228887, 230047, 231207, 232367, 233527, 234687, 235847, 237007, 238167, 239327, 240487, 241647, 242807, 243967, 245127, 246287, 247447, 248607, 249767, 250927, 252087, 253247, 254407, 255567, 256727, 257887, 259047, 260207, 261367, 262527, 263687, 264847, 266007, 267167, 268327, 269487, 270647, 271807, 272967, 274127, 275287, 276447, 277607, 278767, 279927, 281087, 282247, 283407, 284567, 285727, 286887, 288047, 289207, 290367, 291527, 292687, 293847, 295007, 296167, 297327, 298487, 299647, 300807, 301967, 303127, 304287, 305447, 306607, 307767, 308927, 310087, 311247, 312407, 313567, 314727, 315887, 317047, 318207, 319367, 320527, 321687, 322847, 324007, 325167, 326327, 327487, 328647, 329807, 330967, 332127, 333287, 334447, 335607, 336767, 337927, 339087, 340247, 341407, 342567, 343727, 344887, 346047, 347207, 348367, 349527, 350687, 351847, 353007, 354167, 355327, 356487, 357647, 358807, 359967, 361127, 362287, 363447, 364607, 365767, 366927, 368087, 369247, 370407, 371567, 372727, 373887, 375047, 376207, 377367, 378527, 379687, 380847, 382007, 383167, 384327, 385487, 386647, 387807, 388967, 390127, 391287, 392447, 393607, 394767, 395927, 397087, 398247, 399407, 400567, 401727, 402887, 404047, 405207, 406367, 407527, 408687, 409847, 411007, 412167, 413327, 414487, 415647, 416807, 417967, 419127, 420287, 421447, 422607, 423767, 424927, 426087, 427247, 428407, 429567, 430727, 431887, 433047, 434207, 435367, 436527, 437687, 438847, 440007, 441167, 442327, 443487, 444647, 445807, 446967, 448127, 449287, 450447, 451607, 452767, 453927, 455087, 456247, 457407, 458567, 459727, 460887, 462047, 463207, 464367, 465527, 466687, 467847, 469007, 470167, 471327, 472487, 473647, 474807, 475967, 477127, 478287, 479447, 480607, 481767, 482927, 484087, 485247, 486407, 487567, 488727, 489887, 491047, 492207, 493367, 494527, 495687, 496847, 498007, 499167, 500327, 501487, 502647, 503807, 504967, 506127, 507287, 508447, 509607, 510767, 511927, 513087, 514247, 515407, 516567, 517727, 518887, 520047, 521207, 522367, 523527, 524687, 525847, 527007, 528167, 529327, 530487, 531647, 532807, 533967, 535127, 536287, 537447, 538607, 539767, 540927, 542087, 543247, 544407, 545567, 546727, 547887, 549047, 550207, 551367, 552527, 553687, 554847, 556007, 557167, 558327, 559487, 560647, 561807, 562967, 564127, 565287, 566447, 567607, 568767, 569927, 571087, 572247, 573407, 574567, 575727, 576887, 578047, 579207, 580367, 581527, 582687, 583847, 585007, 586167, 587327, 588487, 589647, 590807, 591967, 593127, 594287, 595447, 596607, 597767, 598927, 600087, 601247, 602407, 603567, 604727, 605887, 607047, 608207, 609367, 610527, 611687, 612847, 614007, 615167, 616327, 617487, 618647, 619807, 620967, 622127, 623287, 624447, 625607, 626767, 627927, 629087, 630247, 631407, 632567, 633727, 634887, 636047, 637207, 638367, 639527, 640687, 641847, 643007, 644167, 645327, 646487, 647647, 648807, 649967, 651127, 652287, 653447, 654607, 655767, 656927, 658087, 659247, 660407, 661567, 662727, 663887, 665047, 666207, 667367, 668527, 669687, 670847, 672007, 673167, 674327, 675487, 676647, 677807, 678967, 680127, 681287, 682447, 683607, 684767, 685927, 687087, 688247, 689407, 690567, 691727, 692887, 694047, 695207, 696367, 697527, 698687, 699847, 701007, 702167, 703327, 704487, 705647, 706807, 707967, 709127, 710287, 711447, 712607, 713767, 714927, 716087, 717247, 718407, 719567, 720727, 721887, 723047, 724207, 725367, 726527, 727687, 728847, 730007, 731167, 732327, 733487, 734647, 735807, 736967, 738127, 739287, 740447, 741607, 742767, 743927, 745087, 746247, 747407, 748567, 749727, 750887, 752047, 753207, 754367, 755527, 756687, 757847, 759007, 760167, 761327, 762487, 763647, 764807, 765967, 767127, 768287, 769447, 770607, 771767, 772927, 774087, 775247, 776407, 777567, 778727, 779887, 781047, 782207, 783367, 784527, 785687, 786847, 788007, 789167, 790327, 791487, 792647, 793807, 794967, 796127, 797287, 798447, 799607, 800767, 801927, 803087, 804247, 805407, 806567, 807727, 808887, 810047, 811207, 812367, 813527, 814687, 815847, 817007, 818167, 819327, 820487, 821647, 822807, 823967, 825127, 826287, 827447, 828607, 829767, 830927, 832087, 833247, 834407, 835567, 836727, 837887, 839047, 840207, 841367, 842527, 843687, 844847, 846007, 847167, 848327, 849487, 850647, 851807, 852967, 854127, 855287, 856447, 857607, 858767, 859927, 861087, 862247, 863407, 864567, 865727, 866887, 868047, 869207, 870367, 871527, 872687, 873847, 875007, 876167, 877327, 878487, 879647, 880807, 881967, 883127, 884287, 885447, 886607, 887767, 888927, 890087, 891247, 892407, 893567, 894727, 895887, 897047, 898207, 899367, 900527, 901687, 902847, 904007, 905167, 906327, 907487, 908647, 909807, 910967, 912127, 913287, 914447, 915607, 916767, 917927, 919087, 920247, 921407, 922567, 923727, 924887, 926047, 927207, 928367, 929527, 930687, 931847, 933007, 934167, 935327, 936487, 937647, 938807, 939967, 941127, 942287, 943447, 944607, 945767, 946927, 948087, 949247, 950407, 951567, 952727, 953887, 955047, 956207, 957367, 958527, 959687, 960847, 962007, 963167, 964327, 965487, 966647, 967807, 968967, 970127, 971287, 972447, 973607, 974767, 975927, 977087, 978247, 979407, 980567, 981727, 982887, 984047, 985207, 986367, 987527, 988687, 989847, 991007, 992167, 993327, 994487, 995647, 996807, 997967, 999127, 1000287, 1001447, 1002607, 1003767, 1004927, 1006087, 1007247, 1008407, 1009567, 1010727, 1011887, 1013047, 1014207, 1015367, 1016527, 1017687, 1018847, 1020007, 1021167, 1022327, 1023487, 1024647, 1025807, 1026967, 1028127, 1029287, 1030447, 1031607, 1032767, 1033927, 1035087, 1036247, 1037407, 1038567, 1039727, 1040887, 1042047, 1043207, 1044367, 1045527, 1046687, 1047847, 1049007, 1050167, 1051327, 1052487, 1053647, 1054807, 1055967, 1057127, 1058287, 1059447, 1060607, 1061767, 1062927, 1064087, 1065247, 1066407, 1067567, 1068727, 1069887, 1071047, 1072207, 1073367, 1074527, 1075687, 1076847, 1078007, 1079167, 1080327, 1081487, 1082647, 1083807, 1084967, 1086127, 1087287, 1088447, 1089607, 1090767, 1091927, 1093087, 1094247, 1095407, 1096567, 1097727, 1098887, 1100047, 1101207, 1102367, 1103527, 1104687, 1105847, 1107007, 1108167, 1109327, 1110487, 1111647, 1112807, 1113967, 1115127, 1116287, 1117447, 1118607, 1119767, 1120927, 1122087, 1123247, 1124407, 1125567, 1126727, 1127887, 1129047, 1130207, 1131367, 1132527, 1133687, 1134847, 1136007, 1137167, 1138327, 1139487, 1140647, 1141807, 1142967, 1144127, 1145287, 1146447, 1147607, 1148767, 1149927, 1151087, 1152247, 1153407, 1154567, 1155727, 1156887, 1158047, 1159207, 1160367, 1161527, 1162687, 1163847, 1165007, 1166167, 1167327, 1168487, 1169647, 1170807, 1171967, 1173127, 1174287, 1175447, 1176607, 1177767, 1178927, 1180087, 1181247, 1182407, 1183567, 1184727, 1185887, 1187047, 1188207, 1189367, 1190527, 1191687, 1192847, 1194007, 1195167, 1196327, 1197487, 1198647, 1199807, 1200967, 1202127, 1203287, 1204447, 1205607, 1206767, 1207927, 1209087, 1210247, 1211407, 1212567, 1213727, 1214887, 1216047, 1217207, 1218367, 1219527, 1220687, 1221847, 1223007, 1224167, 1225327, 1226487, 1227647, 1228807, 1229967, 1231127, 1232287, 1233447, 1234607, 1235767, 1236927, 1238087, 1239247, 1240407, 1241567, 1242727, 1243887, 1245047, 1246207, 1247367, 1248527, 1249687, 1250847, 1252007, 1253167, 1254327, 1255487, 1256647, 1257807, 1258967, 1260127, 1261287, 1262447, 1263607, 1264767, 1265927, 1267087, 1268247, 1269407, 1270567, 1271727, 1272887, 1274047, 1275207, 1276367, 1277527, 1278687, 1279847, 1281007, 1282167, 1283327, 1284487, 1285647, 1286807, 1287967, 1289127, 1290287, 1291447, 1292607, 1293767, 1294927, 1296087, 1297247, 1298407, 1299567, 1300727, 1301887, 1303047, 1304207, 1305367, 1306527, 1307687, 1308847, 1310007, 1311167, 1312327, 1313487, 1314647, 1315807, 1316967, 1318127, 1319287, 1320447, 1321607, 1322767, 1323927, 1325087, 1326247, 1327407, 1328567, 1329727, 1330887, 1332047, 1333207, 1334367, 1335527, 1336687, 1337847, 1339007, 1340167, 1341327, 1342487, 1343647, 1344807, 1345967, 1347127, 1348287, 1349447, 1350607, 1351767, 1352927, 1354087, 1355247, 1356407, 1357567, 1358727, 1359887, 1361047, 1362207, 1363367, 1364527, 1365687, 1366847, 1368007, 1369167, 1370327, 1371487, 1372647, 1373807, 1374967, 1376127, 1377287, 1378447, 1379607, 1380767, 1381927, 1383087, 1384247, 1385407, 1386567, 1387727, 1388887, 1390047, 1391207, 1392367, 1393527, 1394687, 1395847, 1397007, 1398167, 1399327, 1400487, 1401647, 1402807, 1403967, 1405127, 1406287, 1407447, 1408607, 1409767, 1410927, 1412087, 1413247, 1414407, 1415567, 1416727, 1417887, 1419047, 1420207, 1421367, 1422527, 1423687, 1424847, 1426007, 1427167, 1428327, 1429487, 1430647, 1431807, 1432967, 1434127, 1435287, 1436447, 1437607, 1438767, 1439927, 1441087, 1442247, 1443407, 1444567, 1445727, 1446887, 1448047, 1449207, 1450367, 1451527, 1452687, 1453847, 1455007, 1456167, 1457327, 1458487, 1459647, 1460807, 1461967, 1463127, 1464287, 1465447, 1466607, 1467767, 1468927, 1470087, 1471247, 1472407, 1473567, 1474727, 1475887, 1477047, 1478207, 1479367, 1480527, 1481687, 1482847, 1484007, 1485167, 1486327, 1487487, 1488647, 1489807, 1490967, 1492127, 1493287, 1494447, 1495607, 1496767, 1497927, 1499087, 1500247, 1501407, 1502567, 1503727, 1504887, 1506047, 1507207, 1508367, 1509527, 1510687, 1511847, 1513007, 1514167, 1515327, 1516487, 1517647, 1518807, 1519967, 1521127, 1522287, 1523447, 1524607, 1525767, 1526927, 1528087, 1529247, 1530407, 1531567, 1532727, 1533887, 1535047, 1536207, 1537367, 1538527, 1539687, 1540847, 1542007, 1543167, 1544327, 1545487, 1546647, 1547807, 1548967, 1550127, 1551287, 1552447, 1553607, 1554767, 1555927, 1557087, 1558247, 1559407, 1560567, 1561727, 1562887, 1564047, 1565207, 1566367, 1567527, 1568687, 1569847, 1571007, 1572167, 1573327, 1574487, 1575647, 1576807, 1577967, 1579127, 1580287, 1581447, 1582607, 1583767, 1584927, 1586087, 1587247, 1588407, 1589567, 1590727, 1591887, 1593047, 1594207, 1595367, 1596527, 1597687, 1598847, 1600007, 1601167, 1602327, 1603487, 1604647, 1605807, 1606967, 1608127, 1609287, 1610447, 1611607, 1612767, 1613927, 1615087, 1616247, 1617407, 1618567, 1619727, 1620887, 1622047, 1623207, 1624367, 1625527, 1626687, 1627847, 1629007, 1630167, 1631327, 1632487, 1633647, 1634807, 1635967, 1637127, 1638287, 1639447, 1640607, 1641767, 1642927, 1644087, 1645247, 1646407, 1647567, 1648727, 1649887, 1651047, 1652207, 1653367, 1654527, 1655687, 1656847, 1658007, 1659167, 1660327, 1661487, 1662647, 1663807, 1664967, 1666127, 1667287, 1668447, 1669607, 1670767, 1671927, 1673087, 1674247, 1675407, 1676567, 1677727, 1678887, 1680047, 1681207, 1682367, 1683527, 1684687, 1685847, 1687007, 1688167, 1689327, 1690487, 1691647, 16928

```

9      swap(u -= q * v, v);
11     }
    } return {s, u, a};
}

```

4.1.7. Chinese Remainder Theorem

Requires: Extended GCD

```

1 // for 0 <= a < m, 0 <= b < n, returns the smallest
3 // such that x % m == a and x % n == b
ll crt(ll a, ll m, ll b, ll n) {
5     if (n > m) swap(a, b), swap(m, n);
6     auto [x, y, g] = extgcd(m, n);
7     assert((a - b) % g == 0); // no solution
8     x = ((b - a) / g * x) % (n / g) * m + a;
9     return x < 0 ? x + m / g * n : x;
}

```

```

7     aux[t] = aux[t - p];
8     Rec(t + 1, p, n, k);
9     for (aux[t] = aux[t - p] + 1; aux[t] < k; ++aux[t])
10        Rec(t + 1, t, n, k);
11 }
12
13 int DeBruijn(int k, int n) {
14     // return cyclic string of length k^n such that every
15     // string of length n using k character appears as a
16     // substring.
17     if (k == 1) return res[0] = 0, 1;
18     fill(aux, aux + k * n, 0);
19     return sz = 0, Rec(1, 1, n, k), sz;
}

```

4.1.8. Pollard's Rho

```

1 ll f(ll x, ll mod) { return (x * x + 1) % mod; }
2 // n should be composite
3 ll pollard_rho(ll n) {
4     if (!(n & 1)) return 2;
5     while (1) {
6         ll y = 2, x = RNG() % (n - 1) + 1, res = 1;
7         for (int sz = 2; res == 1; sz *= 2) {
8             for (int i = 0; i < sz && res == 1; i++) {
9                 x = f(x, n);
10                res = __gcd(abs(x - y), n);
11            }
12            y = x;
13        }
14        if (res != 0 && res != n) return res;
15    }
}

```

4.1.9. Rational Number Binary Search

```

1 struct QQ {
2     ll p, q;
3     QQ go(QQ b, ll d) { return {p + b.p * d, q + b.q * d}; }
4 };
5 bool pred(QQ);
6 // returns smallest p/q in [lo, hi] such that
7 // pred(p/q) is true, and 0 <= p, q <= N
8 QQ frac_bs(ll N) {
9     QQ lo{0, 1}, hi{1, 0};
10    if (pred(lo)) return lo;
11    assert(pred(hi));
12    bool dir = 1, L = 1, H = 1;
13    for (; L || H; dir = !dir) {
14        ll len = 0, step = 1;
15        for (int t = 0; t < 2 && (t ? step /= 2 : step *= 2);)
16            if (QQ mid = hi.go(lo, len + step);
17                mid.p > N || mid.q > N || dir ^ pred(mid))
18                t++;
19        else len += step;
20        swap(lo, hi = hi.go(lo, len));
21        (dir ? L : H) = !len;
22    }
23    return dir ? hi : lo;
}

```

4.2. Combinatorics

4.2.1. De Bruijn Sequence

```

1 int res[kN], aux[kN], a[kN], sz;
2 void Rec(int t, int p, int n, int k) {
3     if (t > n) {
4         if (n % p == 0)
5             for (int i = 1; i <= p; ++i) res[sz++] = aux[i];
6         } else {

```

5. Numeric

5.1. Long Long Multiplication

```
1 using ull = unsigned long long;
2 using ll = long long;
3 using ld = long double;
4 // returns a * b % M where a, b < M < 2**63
5 ull mult(ull a, ull b, ull M) {
6     ll ret = a * b - M * ull(ld(a) * ld(b) / ld(M));
7     return ret + M * (ret < 0) - M * (ret >= (ll)M);
8 }
```

6. Geometry

6.1. Point

```

1 template <typename T> struct P {
2     T x, y;
3     P(T x = 0, T y = 0) : x(x), y(y) {}
4     bool operator<(const P &p) const {
5         return tie(x, y) < tie(p.x, p.y);
6     }
7     bool operator==(const P &p) const {
8         return tie(x, y) == tie(p.x, p.y);
9     }
10    P operator-(const P &p) const { return {x - p.x, y - p.y}; }
11    P operator+(const P &p) const { return {x + p.x, y + p.y}; }
12    P operator*(const P &p) const { return {x * p.x, y * p.y}; }
13    P operator/(const P &p) const { return {x / p.x, y / p.y}; }
14    T dist2() const { return x * x + y * y; }
15    double len() const { return sqrt(dist2()); }
16    P unit() const { return *this / len(); }
17    friend T dot(P a, P b) { return a.x * b.x + a.y * b.y; }
18    friend T cross(P a, P b) { return a.x * b.y - a.y * b.x; }
19    friend T cross(P a, P b, P o) {
20        return cross(a - o, b - o);
21    }
22 };
23 using pt = P<ll>;

```

6.1.1. Quaternion

```

1 constexpr double PI = 3.141592653589793;
2 constexpr double EPS = 1e-7;
3 struct Q {
4     using T = double;
5     T x, y, z, r;
6     Q(T r = 0) : x(0), y(0), z(0), r(r) {}
7     Q(T x, T y, T z, T r = 0) : x(x), y(y), z(z), r(r) {}
8     friend bool operator==(const Q &a, const Q &b) {
9         return (a - b).abs2() <= EPS;
10    }
11    friend bool operator!=(const Q &a, const Q &b) {
12        return !(a == b);
13    }
14    Q operator-(const Q &b) const { return {x - b.x, y - b.y, z - b.z, r - b.r}; }
15    Q operator+(const Q &b) const { return {x + b.x, y + b.y, z + b.z, r + b.r}; }
16    Q operator*(const Q &b) const {
17        return {x * b.x - y * b.y - z * b.z,
18                x * b.y + y * b.x - z * b.r,
19                x * b.z - y * b.r + z * b.x,
20                x * b.r + y * b.r + z * b.z};
21    }
22    Q operator/(const Q &b) const { return *this * b.inv(); }
23    T abs2() const { return x * x + y * y + z * z + r * r; }
24    T len() const { return sqrt(abs2()); }
25    Q conj() const { return {x, -y, -z, r}; }
26    Q unit() const { return *this * (1.0 / len()); }
27    Q inv() const { return conj() * (1.0 / abs2()); }
28    friend T dot(Q a, Q b) {
29        return a.x * b.x + a.y * b.y + a.z * b.z + a.r * b.r;
30    }
31    friend T cross(Q a, Q b) {
32        return {a.y * b.z - a.z * b.y, a.z * b.x - a.x * b.z,
33                a.x * b.y - a.y * b.x,
34                a.r * b.r - a.x * b.x - a.y * b.y - a.z * b.z};
35    }
36    friend Q rotation_around(Q axis, T angle) {
37        return axis.unit() * sin(angle / 2) + cos(angle / 2);
38    }
39 };

```

```

47 Q rotated_around(Q axis, T angle) {
48     Q u = rotation_around(axis, angle);
49     return u * *this / u;
50 }
51 friend Q rotation_between(Q a, Q b) {
52     a = a.unit(), b = b.unit();
53     if (a == -b) {
54         // degenerate case
55         Q ortho = abs(a.y) > EPS ? cross(a, Q(1, 0, 0))
56                                     : cross(a, Q(0, 1, 0));
57         return rotation_around(ortho, PI);
58     }
59     return (a * (a + b)).conj();
60 }

```

6.1.2. Spherical Coordinates

```

1 struct car_p {
2     double x, y, z;
3 };
4 struct sph_p {
5     double r, theta, phi;
6 };
7
8 sph_p conv(car_p p) {
9     double r = sqrt(p.x * p.x + p.y * p.y + p.z * p.z);
10    double theta = asin(p.y / r);
11    double phi = atan2(p.y, p.x);
12    return {r, theta, phi};
13 }
14 car_p conv(sph_p p) {
15     double x = p.r * cos(p.theta) * sin(p.phi);
16     double y = p.r * cos(p.theta) * cos(p.phi);
17     double z = p.r * sin(p.theta);
18     return {x, y, z};
19 }

```

6.2. Segments

```

1 // for non-collinear ABCD, if segments AB and CD intersect
2 bool intersects(pt a, pt b, pt c, pt d) {
3     if (cross(b, c, a) * cross(b, d, a) > 0) return false;
4     if (cross(d, a, c) * cross(d, b, c) > 0) return false;
5     return true;
6 }
7 // the intersection point of lines AB and CD
8 pt intersect(pt a, pt b, pt c, pt d) {
9     auto x = cross(b, c, a), y = cross(b, d, a);
10    if (x == y) {
11        // if(abs(x, y) < 1e-8) {
12        // is parallel
13    } else {
14        return d * (x / (x - y)) - c * (y / (x - y));
15    }
16 }

```

6.3. Convex Hull

```

1 // returns a convex hull in counterclockwise order
2 // for a non-strict one, change cross >= to >
3 vector<pt> convex_hull(vector<pt> p) {
4     sort(ALL(p));
5     if (p[0] == p.back()) return {p[0]};
6     int n = p.size(), t = 0;
7     vector<pt> h(n + 1);
8     for (int i = 2, s = 0; i-- < n; s = --t, reverse(ALL(p)))
9         for (pt j : p) {
10             while (t > s + 1 && cross(i, h[t - 1], h[t - 2]) >= 0)
11                 t--;
12             h[t++] = j;
13         }
14     return h.resize(t), h;
15 }

```

6.3.1. 3D Hull

```

1
3 typedef Point3D<double> P3;
5 struct PR {
6     void ins(int x) { (a == -1 ? a : b) = x; }
7     void rem(int x) { (a == x ? a : b) = -1; }
8     int cnt() { return (a != -1) + (b != -1); }
9     int a, b;
10 };
11
12 struct F {
13     P3 q;
14     int a, b, c;
15 };
16
17 vector<F> hull3d(const vector<P3> &A) {
18     assert(sz(A) >= 4);
19     vector<vector<PR>> E(sz(A), vector<PR>(sz(A), {-1, -1}));
20     #define E(x, y) E[f.x][f.y]
21     vector<F> FS;
22     auto mf = [&](int i, int j, int k, int l) {
23         P3 q = (A[j] - A[i]).cross((A[k] - A[i]));
24         if (q.dot(A[l]) > q.dot(A[i])) q = q * -1;
25         F f{q, i, j, k};
26         E(a, b).ins(k);
27         E(a, c).ins(j);
28         E(b, c).ins(i);
29         FS.push_back(f);
30     };
31     rep(i, 0, 4) rep(j, i + 1, 4) rep(k, j + 1, 4)
32     mf(i, j, k, 6 - i - j - k);
33
34     rep(i, 4, sz(A)) {
35         rep(j, 0, sz(FS)) {
36             F f = FS[j];
37             if (f.q.dot(A[i]) > f.q.dot(A[f.a])) {
38                 E(a, b).rem(f.c);
39                 E(a, c).rem(f.b);
40                 E(b, c).rem(f.a);
41                 swap(FS[j--], FS.back());
42                 FS.pop_back();
43             }
44         }
45         int nw = sz(FS);
46         rep(j, 0, nw) {
47             F f = FS[j];
48             #define C(a, b, c)
49             if (E(a, b).cnt() != 2) mf(f.a, f.b, i, f.c);
50             C(a, b, c);
51             C(a, c, b);
52             C(b, c, a);
53         }
54     }
55     for (F &it : FS)
56         if ((A[it.b] - A[it.a]).cross(A[it.c] - A[it.a]).dot(it.q) <= 0)
57             swap(it.c, it.b);
58     return FS;
59 }
60
61

```

6.4. Angular Sort

```

1 auto angle_cmp = [](const pt &a, const pt &b) {
2     auto btm = [](const pt &a) {
3         return a.y < 0 || (a.y == 0 && a.x < 0);
4     };
5     return make_tuple(btm(a), a.y * b.x, abs2(a)) <
6         make_tuple(btm(b), a.x * b.y, abs2(b));
7 };
8 void angular_sort(vector<pt> &p) {
9     sort(p.begin(), p.end(), angle_cmp);
10 }

```

6.5. Convex Polygon Minkowski Sum

```

1 // 0(n) convex polygon minkowski sum
2 // must be sorted and counterclockwise
3 vector<pt> minkowski_sum(vector<pt> p, vector<pt> q) {
4     auto diff = [](vector<pt> &c) {
5         auto rcmp = [](pt a, pt b) {
6             return pt{a.y, a.x} < pt{b.y, b.x};
7         };
8         rotate(c.begin(), min_element(ALL(c), rcmp), c.end());
9         c.push_back(c[0]);
10        vector<pt> ret;
11        for (int i = 1; i < c.size(); i++)
12            ret.push_back(c[i] - c[i - 1]);
13        return ret;
14    };
15    auto dp = diff(p), dq = diff(q);
16    pt cur = p[0] + q[0];
17    vector<pt> d(dp.size() + dq.size(), ret = {cur});
18    // include angle_cmp from angular-sort.cpp
19    merge(ALL(dp), ALL(dq), d.begin(), angle_cmp);
20    // optional: make ret strictly convex (UB if degenerate)
21    int now = 0;
22    for (int i = 1; i < d.size(); i++) {
23        if (cross(d[i], d[now]) == 0) d[now] = d[i];
24        else d[++now] = d[i];
25    }
26    d.resize(now + 1);
27    // end optional part
28    for (pt v : d) ret.push_back(cur = cur + v);
29    return ret.pop_back(), ret;
30 }

```

6.6. Point In Polygon

```

1 bool on_segment(pt a, pt b, pt p) {
2     return cross(a, b, p) == 0 && dot((p - a), (p - b)) <= 0;
3 }
4 // p can be any polygon, but this is 0(n)
5 bool inside(const vector<pt> &p, pt a) {
6     int cnt = 0, n = p.size();
7     for (int i = 0; i < n; i++) {
8         pt l = p[i], r = p[(i + 1) % n];
9         // change to return 0; for strict version
10        if (on_segment(l, r, a)) return 1;
11        cnt ^= ((a.y < l.y) - (a.y < r.y)) * cross(l, r, a) > 0;
12    }
13    return cnt;
14 }

```

6.6.1. Convex Version

```

1 // no preprocessing version
2 // p must be a strict convex hull, counterclockwise
3 // if point is inside or on border
4 bool is_inside(const vector<pt> &c, pt p) {
5     int n = c.size(), l = 1, r = n - 1;
6     if (cross(c[0], c[l], p) < 0) return false;
7     if (cross(c[n - 1], c[0], p) < 0) return false;
8     while (l < r - 1) {
9         int m = (l + r) / 2;
10        T a = cross(c[0], c[m], p);
11        if (a > 0) l = m;
12        else if (a < 0) r = m;
13        else return dot(c[0] - p, c[m] - p) <= 0;
14    }
15    if (l == r) return dot(c[0] - p, c[l] - p) <= 0;
16    else return cross(c[l], c[r], p) >= 0;
17 }
18
19 // with preprocessing version
20 vector<pt> vecs;
21 pt center;
22 // p must be a strict convex hull, counterclockwise
23 // BEWARE OF OVERFLOWS!!
24 void preprocess(vector<pt> p) {

```

```

25 for (auto &v : p) v = v * 3;
   center = p[0] + p[1] + p[2];
27 center.x /= 3, center.y /= 3;
   for (auto &v : p) v = v - center;
29 vecs = (angular_sort(p), p);
31 bool intersect_strict(pt a, pt b, pt c, pt d) {
   if (cross(b, c, a) * cross(b, d, a) > 0) return false;
33 if (cross(d, a, c) * cross(d, b, c) >= 0) return true;
   return true;
35 }
   // if point is inside or on border
37 bool query(pt p) {
   p = p * 3 - center;
39 auto pr = upper_bound(ALL(vecs), p, angle_cmp);
   if (pr == vecs.end()) pr = vecs.begin();
41 auto pl = (pr == vecs.begin()) ? vecs.back() : *(pr - 1);
   return !intersect_strict({0, 0}, p, pl, *pr);
43 }

```

6.6.2. Offline Multiple Points Version

Requires: Point, GNU PBDS

```

1
3
5 using Double = __float128;
7 using Point = pt<Double, Double>;
9 int n, m;
   vector<Point> poly;
11 vector<Point> query;
   vector<int> ans;
13
15 struct Segment {
   Point a, b;
   int id;
17 };
   vector<Segment> segs;
19
21 Double Xnow;
   inline Double get_y(const Segment &u, Double xnow = Xnow) {
23     const Point &a = u.a;
     const Point &b = u.b;
     return (a.y * (b.x - xnow) + b.y * (xnow - a.x)) /
25         (b.x - a.x);
   }
27 bool operator<(Segment u, Segment v) {
   Double yu = get_y(u);
   Double yv = get_y(v);
29 if (yu != yv) return yu < yv;
   return u.id < v.id;
31 }
   ordered_map<Segment> st;
33
35 struct Event {
   int type; // +1 insert seg, -1 remove seg, 0 query
37 Double x, y;
   int id;
39 };
   bool operator<(Event a, Event b) {
41 if (a.x != b.x) return a.x < b.x;
   if (a.type != b.type) return a.type < b.type;
43 return a.y < b.y;
   }
45 vector<Event> events;
47
49 void solve() {
   set<Double> xs;
   set<Point> ps;
   for (int i = 0; i < n; i++) {
51     xs.insert(poly[i].x);
     ps.insert(poly[i]);
53 }

```

```

55 for (int i = 0; i < n; i++) {
   Segment s{poly[i], poly[(i + 1) % n], i};
   if (s.a.x > s.b.x ||
57     (s.a.x == s.b.x && s.a.y > s.b.y)) {
     swap(s.a, s.b);
59 }
   segs.push_back(s);
   if (s.a.x != s.b.x) {
63     events.push_back({+1, s.a.x + 0.2, s.a.y, i});
     events.push_back({-1, s.b.x - 0.2, s.b.y, i});
65 }
   }
67 for (int i = 0; i < m; i++) {
   events.push_back({0, query[i].x, query[i].y, i});
69 }
   sort(events.begin(), events.end());
71 int cnt = 0;
   for (Event e : events) {
73     int i = e.id;
     Xnow = e.x;
75     if (e.type == 0) {
       Double x = e.x;
       Double y = e.y;
       Segment tmp = {{x - 1, y}, {x + 1, y}, -1};
77       auto it = st.lower_bound(tmp);
79
81       if (ps.count(query[i]) > 0) {
         ans[i] = 0;
83       } else if (xs.count(x) > 0) {
         ans[i] = -2;
85       } else if (it != st.end() &&
         get_y(*it) == get_y(tmp)) {
         ans[i] = 0;
87       } else if (it != st.begin() &&
         get_y(*prev(it)) == get_y(tmp)) {
         ans[i] = 0;
89       } else {
         int rk = st.order_of_key(tmp);
91         if (rk % 2 == 1) {
           ans[i] = 1;
93         } else {
           ans[i] = -1;
95         }
         }
97     }
   } else if (e.type == 1) {
99     st.insert(segs[i]);
     assert((int)st.size() == ++cnt);
101   } else if (e.type == -1) {
     st.erase(segs[i]);
     assert((int)st.size() == --cnt);
103   }
105 }
107 }

```

6.7. Closest Pair

```

1 vector<pt> p; // sort by x first!
   bool cmpy(const pt &a, const pt &b) const {
3     return a.y < b.y;
   }
5 ll sq(ll x) { return x * x; }
   // returns (minimum dist)^2 in [l, r)
7 ll solve(int l, int r) {
   if (r - l <= 1) return 1e18;
9   int m = (l + r) / 2;
   ll mid = p[m].x, d = min(solve(l, m), solve(m, r));
11   auto pb = p.begin();
   inplace_merge(pb + l, pb + m, pb + r, cmpy);
13   vector<pt> s;
   for (int i = l; i < r; i++)
15     if (sq(p[i].x - mid) < d) s.push_back(p[i]);
   for (int i = 0; i < s.size(); i++)
17     for (int j = i + 1;
       j < s.size() && sq(s[j].y - s[i].y) < d; j++)
19       d = min(d, dis(s[i], s[j]));

```

```

21     return d;
    }

```

6.8. Minimum Enclosing Circle

```

1
3  typedef Point<double> P;
4  double ccRadius(const P &A, const P &B, const P &C) {
5      return (B - A).dist() * (C - B).dist() * (A - C).dist() /
6             abs((B - A).cross(C - A)) / 2;
7  }
8  P ccCenter(const P &A, const P &B, const P &C) {
9      P b = C - A, c = B - A;
10     return A + (b * c.dist2() - c * b.dist2()).perp() /
11            b.cross(c) / 2;
12 }
13 pair<P, double> mec(vector<P> ps) {
14     shuffle(all(ps), mt19937(time(0)));
15     P o = ps[0];
16     double r = 0, EPS = 1 + 1e-8;
17     rep(i, 0, sz(ps)) if ((o - ps[i]).dist() > r * EPS) {
18         o = ps[i], r = 0;
19         rep(j, 0, i) if ((o - ps[j]).dist() > r * EPS) {
20             o = (ps[i] + ps[j]) / 2;
21             r = (o - ps[i]).dist();
22             rep(k, 0, j) if ((o - ps[k]).dist() > r * EPS) {
23                 o = ccCenter(ps[i], ps[j], ps[k]);
24                 r = (o - ps[i]).dist();
25             }
26         }
27     }
28     return {o, r};
29 }

```

7. Strings

7.1. Knuth-Morris-Pratt Algorithm

```

1 vector<int> pi(const string &s) {
2     vector<int> p(s.size());
3     for (int i = 1; i < s.size(); i++) {
4         int g = p[i - 1];
5         while (g && s[i] != s[g]) g = p[g - 1];
6         p[i] = g + (s[i] == s[g]);
7     }
8     return p;
9 }
10 vector<int> match(const string &s, const string &pat) {
11     vector<int> p = pi(pat + '\0' + s), res;
12     for (int i = p.size() - s.size(); i < p.size(); i++)
13         if (p[i] == pat.size())
14             res.push_back(i - 2 * pat.size());
15     return res;
16 }

```

7.2. Z Value

```

1 int z[n];
2 void zval(string s) {
3     // z[i] => longest common prefix of s and s[i:], i > 0
4     int n = s.size();
5     z[0] = 0;
6     for (int b = 0, i = 1; i < n; i++) {
7         if (z[b] + b <= i) z[i] = 0;
8         else z[i] = min(z[i - b], z[b] + b - i);
9         while (s[i + z[i]] == s[z[i]]) z[i]++;
10        if (i + z[i] > b + z[b]) b = i;
11    }
12 }

```

7.3. Manacher's Algorithm

```

1 int z[n];
2 void manacher(string s) {
3     // z[i] => longest odd palindrome centered at i
4     // s[i - z[i]] ... i + z[i]
5     // to get all palindromes (including even length)
6     // insert a '#' between each s[i] and s[i + 1]
7     int n = s.size();
8     z[0] = 0;
9     for (int b = 0, i = 1; i < n; i++) {
10        if (z[b] + b >= i)
11            z[i] = min(z[2 * b - i], b + z[b] - i);
12        else z[i] = 0;
13        while (i + z[i] + 1 < n && i - z[i] - 1 >= 0 &&
14                s[i + z[i] + 1] == s[i - z[i] - 1])
15            z[i]++;
16        if (z[i] + i > z[b] + b) b = i;
17    }
18 }

```

7.4. Minimum Rotation

```

1 int min_rotation(string s) {
2     int a = 0, n = s.size();
3     s += s;
4     for (int b = 0; b < n; b++) {
5         for (int k = 0; k < n; k++) {
6             if (a + k == b || s[a + k] < s[b + k]) {
7                 b += max(0, k - 1);
8                 break;
9             }
10            if (s[a + k] > s[b + k]) {
11                a = b;
12                break;
13            }
14        }
15    }
16    return a;
17 }

```

7.5. Aho-Corasick Automaton

```

1 struct Aho_Corasick {
2     static const int maxc = 26, maxn = 4e5;
3     struct NODES {
4         int Next[maxc], fail, ans;
5     };
6     NODES T[maxn];
7     int top, qtop, q[maxn];
8     int get_node(const int &fail) {
9         fill_n(T[top].Next, maxc, 0);
10        T[top].fail = fail;
11        T[top].ans = 0;
12        return top++;
13    }
14    int insert(const string &s) {
15        int ptr = 1;
16        for (char c : s) { // change char id
17            c -= 'a';
18            if (!T[ptr].Next[c]) T[ptr].Next[c] = get_node(ptr);
19            ptr = T[ptr].Next[c];
20        }
21        return ptr;
22    } // return ans_last_place
23    void build_fail(int ptr) {
24        int tmp;
25        for (int i = 0; i < maxc; i++)
26            if (T[ptr].Next[i]) {
27                tmp = T[ptr].fail;
28                while (tmp != 1 && !T[tmp].Next[i])
29                    tmp = T[tmp].fail;
30                if (T[tmp].Next[i] != T[ptr].Next[i])
31                    if (T[tmp].Next[i]) tmp = T[tmp].Next[i];
32                T[T[ptr].Next[i]].fail = tmp;
33                q[qtop++] = T[ptr].Next[i];
34            }
35    }
36    void AC_auto(const string &s) {
37        int ptr = 1;
38        for (char c : s) {
39            while (ptr != 1 && !T[ptr].Next[c]) ptr = T[ptr].fail;
40            if (T[ptr].Next[c]) {
41                ptr = T[ptr].Next[c];
42                T[ptr].ans++;
43            }
44        }
45    }
46    void Solve(string &s) {
47        for (char &c : s) // change char id
48            c -= 'a';
49        for (int i = 0; i < qtop; i++) build_fail(q[i]);
50        AC_auto(s);
51        for (int i = qtop - 1; i > -1; i--)
52            T[T[q[i]].fail].ans += T[q[i]].ans;
53    }
54    void reset() {
55        qtop = top = q[0] = 1;
56        get_node(1);
57    }
58 } AC;
59 // usage example
60 string s, S;
61 int n, t, ans_place[50000];
62 int main() {
63     Tie cin >> t;
64     while (t--) {
65         AC.reset();
66         cin >> S >> n;
67         for (int i = 0; i < n; i++) {
68             cin >> s;
69             ans_place[i] = AC.insert(s);
70         }
71         AC.Solve(S);
72         for (int i = 0; i < n; i++)
73             cout << AC.T[ans_place[i]].ans << '\n';
74     }
75 }

```


75 | }

8. Debug List

- 1 - Pre-submit:
 - 2 - Did you make a typo when copying a template?
 - 3 - Test more cases if unsure.
 - 4 - Write a naive solution and check small cases.
 - 5 - Submit the correct file.
- 7 - General Debugging:
 - 8 - Read the whole problem again.
 - 9 - Have a teammate read the problem.
 - 10 - Have a teammate read your code.
 - 11 - Explain your solution to them (or a rubber duck).
 - 12 - Print the code and its output / debug output.
 - 13 - Go to the toilet.
- 15 - Wrong Answer:
 - 16 - Any possible overflows?
 - 17 - > `__int128`?
 - 18 - Try `__ftrapv` or `#pragma GCC optimize("trapv")`
 - 19 - Floating point errors?
 - 20 - > `long double`?
 - 21 - turn off math optimizations
 - 22 - check for `==`, `>=`, `acos(1.000000001)`, etc.
 - 23 - Did you forget to sort or unique?
 - 24 - Generate large and worst "corner" cases.
 - 25 - Check your `'m' / 'n', 'i' / 'j'` and `'x' / 'y'`.
 - 26 - Are everything initialized or reset properly?
 - 27 - Are you sure about the STL thing you are using?
 - 28 - Read `cppreference` (should be available).
 - 29 - Print everything and run it on pen and paper.
- 31 - Time Limit Exceeded:
 - 32 - Calculate your time complexity again.
 - 33 - Does the program actually end?
 - 34 - Check for `while(q.size())` etc.
 - 35 - Test the largest cases locally.
 - 36 - Did you do unnecessary stuff?
 - 37 - e.g. pass vectors by value
 - 38 - e.g. `memset` for every test case
 - 39 - Is your constant factor reasonable?
- 41 - Runtime Error:
 - 42 - Check memory usage.
 - 43 - Forget to clear or destroy stuff?
 - 44 - > `vector::shrink_to_fit()`
 - 45 - Stack overflow?
 - 46 - Bad pointer / array access?
 - 47 - Try `-fsanitize=address`
 - 48 - Division by zero? NaN's?