

Contents

1 Foundations	1
1.1 PyMath	1
1.2 Java Integer	1
1.3 Java String	1
1.4 Java String builder	2
1.5 Java Math	2
2 Math	2
2.1 formula	2
2.2 extended gcd	2
3 Tree	2
3.1 SegmentTree	2
3.2 HLD	3
3.3 PST	3
4 Graph	4
4.1 cut vertex AND bridges	4
4.2 SCC - Tarjan	4
4.3 BCC - Tarjan	4
4.4 Convex	4
5 String	5
5.1 KMP	5
5.2 Trie	5
5.3 ACAM	5

1 Foundations

1.1 PyMath

```

1 import math
2
3     math.ceil(x) #上高斯
4     math.floor(x) #下高斯
5     math.factorial(x) #階乘
6     math.fabs(x) #絕對值
7     math.fsum(arr) #求和
8     math.gcd(x, y)
9     math.exp(x) # e^x
10    math.log(x, base)
11    math.log2(x)
12    math.log10(x)
13    math.sqrt(x)
14    math.pow(x, y, mod)
15    math.sin(x) # cos, tan, asin, acos, atan,
16      atan2, sinh ...
17    math.hypot(x, y) #歐幾里得範數
18    math.degrees(x) #x從弧度轉角度
19    math.radians(x) #x從角度轉弧度
20    math.gamma(x) #x的gamma函數
21    math.pi #const
22    math.e #const
23    math.inf

```

1.2 Java Integer

```

1 // 常量
2 MAX_VALUE, MIN_VALUE, BYTES, SIZE, TYPE
3
4 // 轉換/解析
5 static int parseInt(String s)
6 static int parseInt(String s, int radix)
7 static int parseUnsignedInt(String s)
8 static int parseUnsignedInt(String s, int
      radix)
9 static Integer valueOf(int i)
10 static Integer valueOf(String s)
11 static Integer valueOf(String s, int radix)
12 static String toString(int i)
13 static String toString(int i, int radix)
14 static String toUnsignedString(int i)
15 static String toUnsignedString(int i, int
      radix)
16 static long toUnsignedLong(int x)
17 static Integer decode(String nm)
      // 支援 0x/0/# 前綴
18 static Integer getInteger(String nm[, int
      val]) // 從系統屬性讀取整數
19
20 // 比較/雜湊/聚合
21 static int compare(int x, int y)
22 static int compareUnsigned(int x, int y)
23 static int hashCode(int value)
24 static int min(int a, int b)
25 static int max(int a, int b)
26 static int sum(int a, int b)
27
28 // 位元操作
29 static int bitCount(int i) // 設定位數
30 static int highestOneBit(int i)
31 static int lowestOneBit(int i)
32 static int numberOfLeadingZeros(int i)
33 static int numberOfTrailingZeros(int i)
34 static int rotateLeft(int i, int distance)
35 static int rotateRight(int i, int distance)
36 static int reverse(int i)
37 static int reverseBytes(int i)
38
39 // 無號運算
40 static int divideUnsigned(int dividend, int
      divisor)

```

```

41 static int remainderUnsigned(int dividend,
      int divisor)

```

1.3 Java String

```

1 // 查詢
2 int length()
3 boolean isEmpty()
4 boolean isBlank() // (since 11)
5 char charAt(int index)
6 int codePointAt(int index)
7 int codePointBefore(int index)
8 int codePointCount(int beginIndex, int
      endIndex)
9 boolean contains(CharSequence s)
10 boolean startsWith(String prefix[, int
      toffset])
11 boolean endsWith(String suffix)
12 int indexOf(String str[, int fromIndex])
13 int lastIndexOf(String str[, int
      fromIndex])
14
15 // 取子字串/子序列
16 String substring(int beginIndex)
17 String substring(int beginIndex, int
      endIndex)
18 CharSequence subSequence(int beginIndex, int
      endIndex)
19
20 // 比較/等價
21 boolean equals(Object obj)
22 boolean equalsIgnoreCase(String
      anotherString)
23 int compareTo(String anotherString)
24 int compareToIgnoreCase(String str)
25 boolean matches(String regex)
26 boolean regionMatches(int toffset, String
      other, int offset, int len)
27 boolean regionMatches(boolean ignoreCase,
      int toffset, String other, int offset,
      int len)
28
29 // 建構/轉換/連接
30 String concat(String str)
31 String replace(char oldChar, char newChar)
32 String replace(CharSequence target,
      CharSequence replacement)
33 String replaceAll(String regex, String
      replacement)
34 String replaceFirst(String regex, String
      replacement)
35 String[] split(String regex[, int limit])
36 String toLowerCase()
37 String toUpperCase()
38 String trim()
39 String strip() // (since 11)
40 String stripLeading() // (since 11)
41 String stripTrailing() // (since 11)
42 String repeat(int count) // (since 11)
43 IntStream chars()
44 Stream<String> lines() // (since 11)
45 String intern()
46
47 // 靜態工具
48 static String format(String format,
      Object... args)
49 static String join(CharSequence delimiter,
      CharSequence... elements)
50 static String join(CharSequence delimiter,
      Iterable<? extends CharSequence>
      elements)
51 static String
      valueOf(primitive/char[]/Object)
52 static String copyValueOf(char[] data[, int
      offset, int count])

```

1.4 Java String builder

```
1 // 長度 / 容量
2 int length()
3 int capacity()
4 void ensureCapacity(int minimumCapacity)
5 void trimToSize()
6 void setLength(int newLength)
7
8 // 存取 / 修改
9 char      charAt(int index)
10 void     setCharAt(int index, char ch)
11 StringBuilder append(... 各種型別 ...)
12 StringBuilder insert(int offset, ... 各種型別 ...
   ...)
13 StringBuilder delete(int start, int end)
14 StringBuilder deleteCharAt(int index)
15 StringBuilder replace(int start, int end,
   String str)
16 StringBuilder reverse()
17
18 // 子串 / 查找
19 String      substring(int start)
20 String      substring(int start, int end)
21 CharSequence subSequence(int start, int end)
22 int        indexOf(String str[, int
   fromIndex])
23 int        lastIndexOf(String str[, int
   fromIndex])
24
25 // 轉換
26 String toString()
```

1.5 Java Math

```
1 // 常量
2 static final double E, PI
3
4 // 絶對值/比較
5 static int/long/float,double abs(x)
6 static T max(a, b)
7 static T min(a, b)
8
9 // 取整/四捨五入
10 static double floor(double a)
11 static double ceil(double a)
12 static double rint(double a)
13 // 最接近整數(偶數優先)
14 static long round(double a) / int
15 // round(float a)
16 static int floorDiv(int x, int y)
17 static int floorMod(int x, int y)
18
19 // 溢位保護(exact 系列, Java 8+)
20 static int/long addExact(a, b)
21 static int/long subtractExact(a, b)
22 static int/long multiplyExact(a, b)
23 static int/long incrementExact(a)
24 static int/long decrementExact(a)
25 static int/int long value)
26 static int/int long negateExact(a)
27
28 // 指對數/幕根
29 static double pow(double a, double b)
30 static double sqrt(double a)
31 static double cbrt(double a)
32 static double exp(double a)
33 static double expm1(double x)
34 static double log(double a)
35 static double log10(double a)
36 static double log1p(double x)
37
38 // 三角/雙曲
39 static double sin/cos/tan(double a)
40 static double asin/acos/atan(double
41 static double atan2(double y, double
```

```

78     if(l <= L && R <= r) return
79         seg[id].sum ;
80
81     push(id) ;
82     int M = (L + R) >> 1 ;
83     if(r <= M) return query(l, r, L, M,
84         lc) ;
85     else if(l > M) return query(l, r,
86         M+1, R, rc) ;
87     else return query(l, r, L, M, lc) +
88         query(l, r, M+1, R, rc) ;
89 }
90 }tree ;

```

3.2 HLD

```

1 /* HLD */
2 int fa[Maxn], top[Maxn], son[Maxn],
3     sz[Maxn], dep[Maxn] = {0}, dfn[Maxn],
4     rnk[Maxn], dfscnt = 0 ;
5
6 void dfs1(int u, int from){
7     fa[u] = from ;
8     dep[u] = dep[from] + 1 ;
9     sz[u] = 1 ;
10
11    for ( auto v : g[u] ) if(v != from){
12        dfs1(v, u) ;
13        sz[u] += sz[v] ;
14        if(son[u] == -1 || sz[v] > sz[son[u]])
15            son[u] = v ;
16    }
17
18    void dfs2(int u, int t){
19        top[u] = t ;
20        dfn[u] = ++dfscnt ;
21        rnk[dfscnt] = u ;
22
23        if(son[u] == -1) return ;
24
25        for ( auto v : g[u] ) if(v != fa[u] && v
26            != son[u]){
27            dfs2(v, v) ;
28        }
29
30    /* Segment Tree */
31    #define lc (id << 1)
32    #define rc ((id << 1) | 1)
33
34    struct ColorSeg{
35        int left, right, tot ;
36
37        ColorSeg operator+(const ColorSeg &o)
38            const {
39                if(tot == 0) return o ;
40                if(o.tot == 0) return *this ;
41
42                ColorSeg tmp ;
43                tmp.left = left ;
44                tmp.right = o.right ;
45                tmp.tot = tot + o.tot - (right ==
46                    o.left) ;
47
48                return tmp ;
49            }
50
51    struct Node{
52        ColorSeg color ;
53        int tag ;
54    }seg[Maxn << 2] ;
55
56    class SegmentTree{

```

```

56     private:
57         void pull(int id){
58             // normal pull
59         }
60
61         void AddTag(int id, int tag){
62             // normal AddTag
63         }
64
65         void push(int id){
66             // normal push
67         }
68
69         void modify(int l, int r, int tag, int
70             L=1, int R=n, int id=1){
71             // normal modify
72         }
73
74         ColorSeg query(int l, int r, int L=1, int
75             R=n, int id=1){
76             // normal query
77         }
78
79         public:
80             void build(int L=1, int R=n, int id=1){
81                 // normal build
82
83                 // update val from u to v (simple path)
84                 void update(int u, int v, int val){
85                     while(top[u] != top[v]){
86                         if(dep[top[u]] < dep[top[v]]) swap(u,
87                             v) ;
88                         modify(dfn[top[u]], dfn[u], val) ;
89                         u = fa[top[u]] ;
90
91                         if(dep[u] < dep[v]) swap(u, v) ;
92                         modify(dfn[v], dfn[u], val) ;
93
94                     // get sum from u to v (simple path)
95                     int get(int u, int v){
96                         pair<int, ColorSeg> U, V ;
97                         ColorSeg M ;
98                         U = {u, {0, 0, 0}} ;
99                         V = {v, {0, 0, 0}} ;
100
101                         while(top[U.first] != top[V.first]){
102                             if(dep[top[U.first]] <
103                                 dep[top[V.first]]) swap(U, V) ;
104                             U.second = query(dfn[top[U.first]],
105                                 dfn[U.first]) + U.second ;
106                             U.first = fa[top[U.first]] ;
107
108                             if(dep[U.first] < dep[V.first]) swap(U,
109                                 V) ;
110
111                             M = query(dfn[V.first], dfn[U.first]) ;
112                         }
113
114                         void init(){
115                             memset(son, -1, sizeof(son)) ;
116                         }
117
118                     // Find range k-th largest number
119                     struct Node{
120                         int sum, left, right ;
121                     }seg[Maxn + 20 * Maxn] ;
122
123                 }
124
125             }
126
127         }
128
129         void swap(int &a, int &b) {
130             int t = a ;
131             a = b ;
132             b = t ;
133         }
134
135         void modify(int id, int val) {
136             pull(id) ;
137             addTag(id, val) ;
138             push(id) ;
139         }
140
141         void addTag(int id, int val) {
142             if(id < 0 || id > n) return ;
143             if(seg[id].tot == 0) return ;
144
145             if(id < n) seg[id].tot += val ;
146             if(id < n && id >= 0) seg[id].sum += val ;
147             if(id < n && id >= 0) seg[id].left += val ;
148             if(id < n && id >= 0) seg[id].right += val ;
149
150             if(id < n && id >= 0) addTag(id << 1, val) ;
151             if(id < n && id >= 0) addTag(id << 1 | 1, val) ;
152
153             if(id < n && id >= 0) push(id) ;
154         }
155
156         void pull(int id) {
157             if(id < 0 || id > n) return ;
158             if(seg[id].tot == 0) return ;
159
160             if(id < n) seg[id].sum = seg[id].tot ;
161             if(id < n && id >= 0) seg[id].left = seg[id].tot ;
162             if(id < n && id >= 0) seg[id].right = seg[id].tot ;
163
164             if(id < n && id >= 0) pull(id << 1) ;
165             if(id < n && id >= 0) pull(id << 1 | 1) ;
166
167             if(id < n && id >= 0) push(id) ;
168         }
169
170         void addTag(int id, int val) {
171             if(id < 0 || id > n) return ;
172             if(seg[id].tot == 0) return ;
173
174             if(id < n) seg[id].tot += val ;
175             if(id < n && id >= 0) seg[id].sum += val ;
176             if(id < n && id >= 0) seg[id].left += val ;
177             if(id < n && id >= 0) seg[id].right += val ;
178
179             if(id < n && id >= 0) addTag(id << 1, val) ;
180             if(id < n && id >= 0) addTag(id << 1 | 1, val) ;
181
182             if(id < n && id >= 0) push(id) ;
183         }
184
185         void push(int id) {
186             if(id < 0 || id > n) return ;
187             if(seg[id].tot == 0) return ;
188
189             if(id < n) seg[id].tot = seg[id].sum ;
190             if(id < n && id >= 0) seg[id].sum = seg[id].tot ;
191             if(id < n && id >= 0) seg[id].left = seg[id].tot ;
192             if(id < n && id >= 0) seg[id].right = seg[id].tot ;
193
194             if(id < n && id >= 0) push(id << 1) ;
195             if(id < n && id >= 0) push(id << 1 | 1) ;
196
197             if(id < n && id >= 0) pull(id) ;
198         }
199
200         void modify(int l, int r, int tag, int
201             L=1, int R=n, int id=1){
202             // normal modify
203         }
204
205         ColorSeg query(int l, int r, int L=1, int
206             R=n, int id=1){
207             // normal query
208         }
209
210         public:
211             PersistentSegmentTree(int _n){
212                 n = _n ;
213                 cnt = 0 ;
214
215                 int root = build(1, n) ;
216                 version.push_back(root) ;
217             }
218
219             void update(int ver, int idx){
220                 auto upd = [&](auto &&self, const int
221                     cur, int L, int R){
222                     int cur_cnt = cnt++ ;
223
224                     if(L == R){
225                         seg[cur_cnt] = {seg[cur].sum + 1, 0,
226                             0} ;
227                         return cur_cnt ;
228                     }
229
230                     int M = (L + R) >> 1 ;
231                     int lc = seg[cur].left ;
232                     int rc = seg[cur].right ;
233
234                     if(idx <= M) lc = self(self,
235                         seg[cur].left, L, M) ;
236                     else rc = self(self, seg[cur].right,
237                         M+1, R) ;
238
239                     seg[cur_cnt] = {seg[lc].sum +
240                         seg[rc].sum, lc, rc} ;
241
242                     return cur_cnt ;
243                 };
244
245                 int root = upd(upd, version[ver], 1, n) ;
246                 version.push_back(root) ;
247             }
248
249             int query(int verL, int verR, int k){
250                 auto qry = [&](auto &&self, const int
251                     cur_old, const int cur_new, int L,
252                     int R){
253                     if(L == R) return L ;
254
255                     int old_l = seg[cur_old].left, old_r =
256                         seg[cur_old].right ;
257                     int new_l = seg[cur_new].left, new_r =
258                         seg[cur_new].right ;
259
260                     int dl = seg[new_l].sum -
261                         seg[old_l].sum ;
262                     int dr = seg[new_r].sum -
263                         seg[old_r].sum ;
264
265                     int M = (L + R) >> 1 ;
266
267                     if(k <= old_l) return L ;
268                     if(k >= old_r) return R ;
269
270                     if(k <= new_l) return L ;
271                     if(k >= new_r) return R ;
272
273                     if(k <= old_r && k >= new_l) return M ;
274
275                     if(k <= old_r && k >= new_r) return
276                         self(self, cur_new, L, R) ;
277
278                     if(k <= old_r && k <= new_l) return
279                         self(self, cur_old, L, R) ;
280
281                     if(k >= old_l && k <= new_r) return
282                         self(self, cur_new, L, R) ;
283
284                     if(k >= old_l && k >= new_r) return
285                         self(self, cur_old, L, R) ;
286
287                     if(k > old_r && k < new_l) return
288                         self(self, cur_new, L, R) ;
289
290                     if(k > old_r && k < new_r) return
291                         self(self, cur_old, L, R) ;
292
293                     if(k > old_l && k < new_r) return
294                         self(self, cur_new, L, R) ;
295
296                     if(k > old_l && k > new_r) return
297                         self(self, cur_old, L, R) ;
298
299                     if(k > old_l && k > new_l) return
300                         self(self, cur_new, L, R) ;
301
302                     if(k < old_r && k > new_r) return
303                         self(self, cur_old, L, R) ;
304
305                     if(k < old_r && k > new_l) return
306                         self(self, cur_new, L, R) ;
307
308                     if(k < old_l && k > new_r) return
309                         self(self, cur_old, L, R) ;
310
311                     if(k < old_l && k > new_l) return
312                         self(self, cur_new, L, R) ;
313
314                     if(k < old_l && k < new_r) return
315                         self(self, cur_new, L, R) ;
316
317                     if(k < old_l && k < new_l) return
318                         self(self, cur_old, L, R) ;
319
320                     if(k > old_r && k < new_r) return
321                         self(self, cur_new, L, R) ;
322
323                     if(k > old_r && k < new_l) return
324                         self(self, cur_old, L, R) ;
325
326                     if(k > old_l && k < new_r) return
327                         self(self, cur_new, L, R) ;
328
329                     if(k > old_l && k < new_l) return
330                         self(self, cur_old, L, R) ;
331
332                     if(k > old_l && k > new_r) return
333                         self(self, cur_new, L, R) ;
334
335                     if(k > old_r && k > new_r) return
336                         self(self, cur_old, L, R) ;
337
338                     if(k > old_r && k > new_l) return
339                         self(self, cur_new, L, R) ;
340
341                     if(k > old_l && k > new_r) return
342                         self(self, cur_old, L, R) ;
343
344                     if(k > old_l && k > new_l) return
345                         self(self, cur_new, L, R) ;
346
347                     if(k < old_l && k > new_r) return
348                         self(self, cur_new, L, R) ;
349
350                     if(k < old_l && k > new_l) return
351                         self(self, cur_old, L, R) ;
352
353                     if(k > old_r && k < new_r) return
354                         self(self, cur_new, L, R) ;
355
356                     if(k > old_r && k < new_l) return
357                         self(self, cur_old, L, R) ;
358
359                     if(k > old_l && k < new_r) return
360                         self(self, cur_new, L, R) ;
361
362                     if(k > old_l && k < new_l) return
363                         self(self, cur_old, L, R) ;
364
365                     if(k > old_l && k > new_r) return
366                         self(self, cur_new, L, R) ;
367
368                     if(k > old_r && k > new_r) return
369                         self(self, cur_old, L, R) ;
370
371                     if(k > old_r && k > new_l) return
372                         self(self, cur_new, L, R) ;
373
374                     if(k > old_l && k > new_r) return
375                         self(self, cur_old, L, R) ;
376
377                     if(k > old_l && k > new_l) return
378                         self(self, cur_new, L, R) ;
379
380                     if(k < old_l && k > new_r) return
381                         self(self, cur_new, L, R) ;
382
383                     if(k < old_l && k > new_l) return
384                         self(self, cur_old, L, R) ;
385
386                     if(k > old_r && k < new_r) return
387                         self(self, cur_new, L, R) ;
388
389                     if(k > old_r && k < new_l) return
390                         self(self, cur_old, L, R) ;
391
392                     if(k > old_l && k < new_r) return
393                         self(self, cur_new, L, R) ;
394
395                     if(k > old_l && k < new_l) return
396                         self(self, cur_old, L, R) ;
397
398                     if(k > old_l && k > new_r) return
399                         self(self, cur_new, L, R) ;
400
401                     if(k > old_r && k > new_r) return
402                         self(self, cur_old, L, R) ;
403
404                     if(k > old_r && k > new_l) return
405                         self(self, cur_new, L, R) ;
406
407                     if(k > old_l && k > new_r) return
408                         self(self, cur_old, L, R) ;
409
410                     if(k > old_l && k > new_l) return
411                         self(self, cur_new, L, R) ;
412
413                     if(k < old_l && k > new_r) return
414                         self(self, cur_new, L, R) ;
415
416                     if(k < old_l && k > new_l) return
417                         self(self, cur_old, L, R) ;
418
419                     if(k > old_r && k < new_r) return
420                         self(self, cur_new, L, R) ;
421
422                     if(k > old_r && k < new_l) return
423                         self(self, cur_old, L, R) ;
424
425                     if(k > old_l && k < new_r) return
426                         self(self, cur_new, L, R) ;
427
428                     if(k > old_l && k < new_l) return
429                         self(self, cur_old, L, R) ;
430
431                     if(k > old_l && k > new_r) return
432                         self(self, cur_new, L, R) ;
433
434                     if(k > old_r && k > new_r) return
435                         self(self, cur_old, L, R) ;
436
437                     if(k > old_r && k > new_l) return
438                         self(self, cur_new, L, R) ;
439
440                     if(k > old_l && k > new_r) return
441                         self(self, cur_old, L, R) ;
442
443                     if(k > old_l && k > new_l) return
444                         self(self, cur_new, L, R) ;
445
446                     if(k < old_l && k > new_r) return
447                         self(self, cur_new, L, R) ;
448
449                     if(k < old_l && k > new_l) return
450                         self(self, cur_old, L, R) ;
451
452                     if(k > old_r && k < new_r) return
453                         self(self, cur_new, L, R) ;
454
455                     if(k > old_r && k < new_l) return
456                         self(self, cur_old, L, R) ;
457
458                     if(k > old_l && k < new_r) return
459                         self(self, cur_new, L, R) ;
460
461                     if(k > old_l && k < new_l) return
462                         self(self, cur_old, L, R) ;
463
464                     if(k > old_l && k > new_r) return
465                         self(self, cur_new, L, R) ;
466
467                     if(k > old_r && k > new_r) return
468                         self(self, cur_old, L, R) ;
469
470                     if(k > old_r && k > new_l) return
471                         self(self, cur_new, L, R) ;
472
473                     if(k > old_l && k > new_r) return
474                         self(self, cur_old, L, R) ;
475
476                     if(k > old_l && k > new_l) return
477                         self(self, cur_new, L, R) ;
478
479                     if(k < old_l && k > new_r) return
480                         self(self, cur_new, L, R) ;
481
482                     if(k < old_l && k > new_l) return
483                         self(self, cur_old, L, R) ;
484
485                     if(k > old_r && k < new_r) return
486                         self(self, cur_new, L, R) ;
487
488                     if(k > old_r && k < new_l) return
489                         self(self, cur_old, L, R) ;
490
491                     if(k > old_l && k < new_r) return
492                         self(self, cur_new, L, R) ;
493
494                     if(k > old_l && k < new_l) return
495                         self(self, cur_old, L, R) ;
496
497                     if(k > old_l && k > new_r) return
498                         self(self, cur_new, L, R) ;
499
500                     if(k > old_r && k > new_r) return
501                         self(self, cur_old, L, R) ;
502
503                     if(k > old_r && k > new_l) return
504                         self(self, cur_new, L, R) ;
505
506                     if(k > old_l && k > new_r) return
507                         self(self, cur_old, L, R) ;
508
509                     if(k > old_l && k > new_l) return
510                         self(self, cur_new, L, R) ;
511
512                     if(k < old_l && k > new_r) return
513                         self(self, cur_new, L, R) ;
514
515                     if(k < old_l && k > new_l) return
516                         self(self, cur_old, L, R) ;
517
518                     if(k > old_r && k < new_r) return
519                         self(self, cur_new, L, R) ;
520
521                     if(k > old_r && k < new_l) return
522                         self(self, cur_old, L, R) ;
523
524                     if(k > old_l && k < new_r) return
525                         self(self, cur_new, L, R) ;
526
527                     if(k > old_l && k < new_l) return
528                         self(self, cur_old, L, R) ;
529
530                     if(k > old_l && k > new_r) return
531                         self(self, cur_new, L, R) ;
532
533                     if(k > old_r && k > new_r) return
534                         self(self, cur_old, L, R) ;
535
536                     if(k > old_r && k > new_l) return
537                         self(self, cur_new, L, R) ;
538
539                     if(k > old_l && k > new_r) return
540                         self(self, cur_old, L, R) ;
541
542                     if(k > old_l && k > new_l) return
543                         self(self, cur_new, L, R) ;
544
545                     if(k < old_l && k > new_r) return
546                         self(self, cur_new, L, R) ;
547
548                     if(k < old_l && k > new_l) return
549                         self(self, cur_old, L, R) ;
550
551                     if(k > old_r && k < new_r) return
552                         self(self, cur_new, L, R) ;
553
554                     if(k > old_r && k < new_l) return
555                         self(self, cur_old, L, R) ;
556
557                     if(k > old_l && k < new_r) return
558                         self(self, cur_new, L, R) ;
559
560                     if(k > old_l && k < new_l) return
561                         self(self, cur_old, L, R) ;
562
563                     if(k > old_l && k > new_r) return
564                         self(self, cur_new, L, R) ;
565
566                     if(k > old_r && k > new_r) return
567                         self(self, cur_old, L, R) ;
568
569                     if(k > old_r && k > new_l) return
570                         self(self, cur_new, L, R) ;
571
572                     if(k > old_l && k > new_r) return
573                         self(self, cur_old, L, R) ;
574
575                     if(k > old_l && k > new_l) return
576                         self(self, cur_new, L, R) ;
577
578                     if(k < old_l && k > new_r) return
579                         self(self, cur_new, L, R) ;
580
581                     if(k < old_l && k > new_l) return
582                         self(self, cur_old, L, R) ;
583
584                     if(k > old_r && k < new_r) return
585                         self(self, cur_new, L, R) ;
586
587                     if(k > old_r && k < new_l) return
588                         self(self, cur_old, L, R) ;
589
590                     if(k > old_l && k < new_r) return
591                         self(self, cur_new, L, R) ;
592
593                     if(k > old_l && k < new_l) return
594                         self(self, cur_old, L, R) ;
595
596                     if(k > old_l && k > new_r) return
597                         self(self, cur_new, L, R) ;
598
599                     if(k > old_r && k > new_r) return
600                         self(self, cur_old, L, R) ;
601
602                     if(k > old_r && k > new_l) return
603                         self(self, cur_new, L, R) ;
604
605                     if(k > old_l && k > new_r) return
606                         self(self, cur_old, L, R) ;
607
608                     if(k > old_l && k > new_l) return
609                         self(self, cur_new, L, R) ;
610
611                     if(k < old_l && k > new_r) return
612                         self(self, cur_new, L, R) ;
613
614                     if(k < old_l && k > new_l) return
615                         self(self, cur_old, L, R) ;
616
617                     if(k > old_r && k < new_r) return
618                         self(self, cur_new, L, R) ;
619
620                     if(k > old_r && k < new_l) return
621                         self(self, cur_old, L, R) ;
622
623                     if(k > old_l && k < new_r) return
624                         self(self, cur_new, L, R) ;
625
626                     if(k > old_l && k < new_l) return
627                         self(self, cur_old, L, R) ;
628
629                     if(k > old_l && k > new_r) return
630                         self(self, cur_new, L, R) ;
631
632                     if(k > old_r && k > new_r) return
633                         self(self, cur_old, L, R) ;
634
635                     if(k > old_r && k > new_l) return
636                         self(self, cur_new, L, R) ;
637
638                     if(k > old_l && k > new_r) return
639                         self(self, cur_old, L, R) ;
640
641                     if(k > old_l && k > new_l) return
642                         self(self, cur_new, L, R) ;
643
644                     if(k < old_l && k > new_r) return
645                         self(self, cur_new, L, R) ;
646
647                     if(k < old_l && k > new_l) return
648                         self(self, cur_old, L, R) ;
649
650                     if(k > old_r && k < new_r) return
651                         self(self, cur_new, L, R) ;
652
653                     if(k > old_r && k < new_l) return
654                         self(self, cur_old, L, R) ;
655
656                     if(k > old_l && k < new_r) return
657                         self(self, cur_new, L, R) ;
658
659                     if(k > old_l && k < new_l) return
660                         self(self, cur_old, L, R) ;
661
662                     if(k > old_l && k > new_r) return
663                         self(self, cur_new, L, R) ;
664
665                     if(k > old_r && k > new_r) return
666                         self(self, cur_old, L, R) ;
667
668                     if(k > old_r && k > new_l) return
669                         self(self, cur_new, L, R) ;
670
671                     if(k > old_l && k > new_r) return
672                         self(self, cur_old, L, R) ;
673
674                     if(k > old_l && k > new_l) return
675                         self(self, cur_new, L, R) ;
676
677                     if(k < old_l && k > new_r) return
678                         self(self, cur_new, L, R) ;
679
680                     if(k < old_l && k > new_l) return
681                         self(self, cur_old, L, R) ;
682
683                     if(k > old_r && k < new_r) return
684                         self(self, cur_new, L, R) ;
685
686                     if(k > old_r && k < new_l) return
687                         self(self, cur_old, L, R) ;
688
689                     if(k > old_l && k < new_r) return
690                         self(self, cur_new, L, R) ;
691
692                     if(k > old_l && k < new_l) return
693                         self(self, cur_old, L, R) ;
694
695                     if(k > old_l && k > new_r) return
696                         self(self, cur_new, L, R) ;
697
698                     if(k > old_r && k > new_r) return
699                         self(self, cur_old, L, R) ;
700
701                     if(k > old_r && k > new_l) return
702                         self(self, cur_new, L, R) ;
703
704                     if(k > old_l && k > new_r) return
705                         self(self, cur_old, L, R) ;
706
707                     if(k > old_l && k > new_l) return
708                         self(self, cur_new, L, R) ;
709
710                     if(k < old_l && k > new_r) return
711                         self(self, cur_new, L, R) ;
712
713                     if(k < old_l && k > new_l) return
714                         self(self, cur_old, L, R) ;
715
716                     if(k > old_r && k < new_r) return
717                         self(self, cur_new, L, R) ;
718
719                     if(k > old_r && k < new_l) return
720                         self(self, cur_old, L, R) ;
721
722                     if(k > old_l && k < new_r) return
723                         self(self, cur_new, L, R) ;
724
725                     if(k > old_l && k < new_l) return
726                         self(self, cur_old, L, R) ;
727
728                     if(k > old_l && k > new_r) return
729                         self(self, cur_new, L, R) ;
730
731                     if(k > old_r && k > new_r) return
732                         self(self, cur_old, L, R) ;
733
734                     if(k > old_r && k > new_l) return
735                         self(self, cur_new, L, R) ;
736
737                     if(k > old_l && k > new_r) return
738                         self(self, cur_old, L, R) ;
739
740                     if(k > old_l && k > new_l) return
741                         self(self, cur_new, L, R) ;
742
743                     if(k < old_l && k > new_r) return
744                         self(self, cur_new, L, R) ;
745
746                     if(k < old_l && k > new_l) return
747                         self(self, cur_old, L, R) ;
748
749                     if(k > old_r && k < new_r) return
750                         self(self, cur_new, L, R) ;
751
752                     if(k > old_r && k < new_l) return
753                         self(self, cur_old, L, R) ;
754
755                     if(k > old_l && k < new_r) return
756                         self(self, cur_new, L, R) ;
757
758                     if(k > old_l && k < new_l) return
759                         self(self, cur_old, L, R) ;
760
761                     if(k > old_l && k > new_r) return
762                         self(self, cur_new, L, R) ;
763
764                     if(k > old_r && k > new_r) return
765                         self(self, cur_old, L, R) ;
766
767                     if(k > old_r && k > new_l) return
768                         self(self, cur_new, L, R) ;
769
770                     if(k > old_l && k > new_r) return
771                         self(self, cur_old, L, R) ;
772
773                     if(k > old_l && k > new_l) return
774                         self(self, cur_new, L, R) ;
775
776                     if(k < old_l && k > new_r) return
777                         self(self, cur_new, L, R) ;
778
779                     if(k < old_l && k > new_l) return
780                         self(self, cur_old, L, R) ;
781
782                     if(k > old_r && k < new_r) return
783                         self(self, cur_new, L, R) ;
784
785                     if(k > old_r && k < new_l) return
786                         self(self, cur_old, L, R) ;
787
788                     if(k > old_l && k < new_r) return
789                         self(self, cur_new, L, R) ;
790
791                     if(k > old_l && k < new_l) return
792                         self(self, cur_old, L, R) ;
793
794                     if(k > old_l && k > new_r) return
795                         self(self, cur_new, L, R) ;
796
797                     if(k > old_r && k > new_r) return
798                         self(self, cur_old, L, R) ;
799
800                     if(k > old_r && k > new_l) return
801                         self(self, cur_new, L, R) ;
802
803                     if(k > old_l && k > new_r) return
804                         self(self, cur_old, L, R) ;
805
806                     if(k > old
```

```
72     if(dl >= k) return self(self, old_l,  
73         new_l, L, M) ;  
74     k -= dl ;  
75     return self(self, old_r, new_r, M+1,  
76         R) ;  
77 } ;  
78  
79     int idx = qry(qry, version[verL-1],  
80         version[verR], 1, n) ;  
81     return idx ;  
82 } ;
```

4 Graph

4.1 cut vertex AND bridges

```

1 int dfn[Maxn] = {-1}, low[Maxn] = {-1},
2     dfscnt ;
3
4 void dfs(int u, int fa){
5     dfn[u] = low[u] = ++dfscnt ;
6     int child = 0 ;
7
8     for ( auto v : g[u] ) if(v != fa){
9         if(dfn[v] == -1){
10             child++ ;
11             dfs(v, u) ;
12             low[u] = min(low[u], low[v]) ;
13
14             if(low[v] >= dfn[u]){
15                 // this edge is a bridge
16             }
17
18             if(u != fa && low[v] >= dfn[u]){
19                 // this node v is a articulation point
20             }
21             else low[u] = min(low[u], dfn[v]) ;
22         }
23     }
24     if(u == fa && child > 1){
25         // this node u is a articulation point
26     }
27 }
```

4.2 SCC - Tarjan

```

1 vector<int> scc[Maxn] ;
2 int dfn[Maxn], low[Maxn], sccId[Maxn],
   dfscnt = 0, cnt_scc = 0 ;
3 stack<int> st ;
4 bitset<Maxn> inSt, vis ;
5
6 void dfs(int u, int from){
7   dfn[u] = low[u] = ++dfscnt ;
8   st.push(u) ;
9   inSt[u] = 1 ;
10
11  for ( auto v : g[u] ){
12    if(!inSt[v] && dfn[v] != -1) continue
13    if(dfn[v] == -1) dfs(v, u) ;
14    low[u] = min(low[u], low[v]) ;
15  }
16
17  if(dfn[u] == low[u]){
18    cnt_scc++ ;
19    int x ;
20
21    do{
22      x = st.top() ;
23      st.pop() ;
24
25      inSt[x] = 0 ;

```

```

26     sccId[x] = cnt_scc ;
27     scc[cnt_scc].push_back(x) ;
28 }
29 while(x != u) ;
30 }
31 }
32
33 // SCC to DAG (after dfs)
34 vector<int> dag[Maxn] ;
35
36 void scc_to_dag(){
37     vector<int> dag[Maxn] ;
38     for ( int u=1 ; u<=n ; u++ ){
39         for ( auto v : g[u] ){
40             if(sccId[u] != sccId[v]){
41                 dag[sccId[u]].push_back(sccId[v])
42             }
43         }
44     }
45 }
46
47 void init(){
48     memset(dfn, -1, sizeof(dfn)) ;
49     memset(low, -1, sizeof(low)) ;
50 }
51
52 int main(){
53     init() ;
54     input() ;
55     for ( int i=1 ; i<=n ; i++ ) if(dfn[i]
56                                     == -1){
57         dfs(i, i) ;
58     }

```

4.3 BCC - Tarjam

```

36     for ( int i=head[u] ; i!=-1 ; i=e[i].next
37         ){
38         int v = e[i].v ;
39
40         if(vis_bcc[v] != -1 || bz[i]) continue ;
41         dfs2(v, id) ;
42     }
43 }
44 void init(){
45     memset(dfn, -1, sizeof(dfn)) ;
46     memset(head, -1, sizeof(head)) ;
47     memset(vis_bcc, -1, sizeof(vis_bcc)) ;
48 }
49
50 int main(){
51     init() ;
52     input() ;
53     for ( int i=1 ; i<=n ; i++ ) if(dfn[i]
54         == -1){
55         dfs1(i, 0) ;
56     }
57     for ( int i=1 ; i<=n ; i++ ) if(vis_bcc[i]
58         == -1){
59         bcc.push_back(vector<int>()) ;
60         dfs2(i, bcc_cnt++) ;
61     }
62 }
```

4.4 Convex

```

1 struct Coordinate{
2     long long x, y ;
3
4     friend bool operator<(const Coordinate&a,
5         const Coordinate& b){
6         if(a.x == b.x) return a.y < b.y ;
7         return a.x < b.x ;
8     }
9
10    friend bool operator==(const Coordinate&
11        a, const Coordinate& b){
12        return a.x == b.x && a.y == b.y ;
13    }
14 } ;
15
16 vector<Coordinate> nodes ;
17
18 long long cross(const Coordinate& o, const
19     Coordinate& a, const Coordinate& b){
20     return (a.x - o.x) * (b.y - o.y) - (a.y -
21         o.y) * (b.x - o.x) ;
22 }
23
24 void input(){
25     nodes.clear() ;
26
27     int n, x, y ;
28     char c ;
29     cin >> n ;
30
31     for ( int i=0 ; i<n ; i++ ){
32         cin >> x >> y >> c ;
33         if(c == 'Y') nodes.push_back({x, y}) ;
34     }
35 }
36
37 void monotone(){
38     sort(nodes.begin(), nodes.end()) ;
39
40     int n = unique(nodes.begin(), nodes.end())
41             - nodes.begin() ;
42
43     vector<Coordinate> ch(n+1) ;
44
45     int m = 0 ;

```

```

for ( int i=0 ; i<n ; i++ ){
    while(m > 1 && cross(ch[m-2], ch[m-1],
        nodes[i]) < 0) m-- ;
    ch[m++] = nodes[i] ;
}
for ( int i=n-2, t=m ; i>=0 ; i-- ){
    while(m > t && cross(ch[m-2], ch[m-1],
        nodes[i]) < 0) m-- ;
    ch[m++] = nodes[i] ;
}
if(n > 1) m-- ;
cout << m << endl ;

for ( int i=0 ; i<m ; i++ ) cout <<
    ch[i].x << " " << ch[i].y << endl ;
}

```

5 String

5.1 KMP

```

int Next[N] ;
void kmp(string &str){
    Next[0] = -1 ;
    if(str.size() <= 1) return ;
    Next[1] = 0 ;

    int cur = 2, check = 0 ;

    while(cur < str.size()){
        if(str[cur - 1] == str[check])
            Next[cur++] = ++check ;
        else if(check > 0) check =
            Next[check] ;
        else Next[cur++] = 0 ;
    }
}

int main(){
    ios::sync_with_stdio(false) ;
    cin.tie(nullptr) ;
    cout.tie(nullptr) ;

    string s1, s2 ;
    while(cin >> s1){
        s2 = s1 ;
        reverse(s2.begin(), s2.end()) ;
        kmp(s2) ;

        int x=0, y=0 ;
        while(x < s1.size() && y < s2.size()){
            if(s1[x] == s2[y]){
                x++ ;
                y++ ;
            }
            else if(y > 0) y = Next[y] ;
            else x++ ;
        }
        cout << s1 << s2.substr(y) << endl ;
    }

    return 0 ;
}

```

5.2 Trie

```
class TrieNode{
public:
    set<int> end ;
    TrieNode *next[26] ;
```

```

6   TrieNode(){
7     for ( int i=0 ; i<26 ; i++ ) next[i] = nullptr ;
8   }
9 };
10
11 class Trie{
12 private:
13   int cnt ;
14   TrieNode *root ;
15 public:
16   Trie() : cnt(0) {
17     root = new TrieNode() ;
18   }
19
20   void insert(string &str, int n){
21     TrieNode* node = root ;
22     for ( auto s : str ){
23       int path = s - 'a' ;
24
25       if(node->next[path] == nullptr)
26         node->next[path] = new TrieNode() ;
27       node = node->next[path] ;
28     }
29     node->end.insert(n) ;
30   }
31
32   void search(string &str){
33     TrieNode* node = root ;
34     for ( auto s : str ){
35       int path = s - 'a' ;
36       if(node->next[path] == nullptr)
37         return ;
38       node = node->next[path] ;
39
40       int flg = 0 ;
41       for ( auto n : node->end ){
42         if(flg) cout << " " ;
43         else flg = 1 ;
44
45         cout << n ;
46     }
47
48   void clear(TrieNode* node) {
49     if (!node) return ;
50     for (int i = 0; i < 26; i++) {
51       if (node->next[i]) {
52         clear(node->next[i]) ;
53       }
54     }
55     delete node ;
56   }
57
58 ~Trie(){
59   clear(root) ;
60 }
61 };

```

5.3 ACAM

```

1 class ACAutomation{
2 private:
3   vector<int> fail, end, order ;
4   vector<vector<int>> tree ;
5
6   int base, alpha ;
7
8   int new_node(){
9     tree.emplace_back(alpha, 0) ;
10    fail.push_back(0) ;
11
12    return tree.size() - 1 ;
13 }
14 public:
15
16   ACAutomation(int _base='a', int _alpha=26)
17     : base(_base), alpha(_alpha) {
18       clear() ;
19   }
20
21   void clear(){
22     fail.assign(1, 0) ;
23     order.clear() ;
24     end.clear() ;
25     tree.assign(1, vector<int>(alpha, 0)) ;
26   }
27
28   void add_pattern(const string &pattern){
29     int u = 0 ;
30     for ( auto ch : pattern ){
31       int v = ch - base ;
32
33       if(tree[u][v] == 0) tree[u][v] =
34         new_node() ;
35       u = tree[u][v] ;
36     }
37
38     end.push_back(u) ;
39   }
40
41   void build(){
42     queue<int> q ;
43     order.clear() ;
44     order.push_back(0) ;
45
46     for ( int i=0 ; i<alpha ; i++ )
47       if(tree[0][i] > 0){
48         q.push(tree[0][i]) ;
49     }
50
51     while(!q.empty()){
52       int u = q.front() ; q.pop() ;
53       order.push_back(u) ;
54
55       for ( int i=0 ; i<alpha ; i++ ){
56         if(tree[u][i] == 0) tree[u][i] =
57           tree[fail[u]][i] ;
58         else{
59           fail[tree[u][i]] = tree[fail[u]][i] ;
60           q.push(tree[u][i]) ;
61         }
62       }
63     }
64
65   vector<int> count_per_pattern(const string
66     &text) const {
67     int u = 0 ;
68     vector<int> vis(tree.size(), 0) ;
69
70     for ( char ch : text ){
71       u = tree[u][ch - base] ;
72       vis[u]++;
73     }
74
75     for ( int i=order.size()-1 ; i>=1 ; i-- )
76       if (int x = order[i] ;
77           vis[fail[x]] += vis[x] ) ;
78
79     vector<int> ans(end.size(), 0) ;
80     for ( int id=0 ; id<end.size() ; id++ ){
81       ans[id] = vis[end[id]] ;
82     }
83
84     return ans ;
85   }
86
87 };

```