

```

1: // g++ p.cpp -o p && ./p < 1.in
2: #include <iostream>
3: #include <string>
4: #include <algorithm>
5: #include <functional>
6: #include <cmath>
7: #include <array>
8: #include <vector>
9: #include <list>
10: #include <deque>
11: #include <queue>
12: #include <stack>
13: #include <map>
14: #include <set>
15: #include <unordered_map>
16: #include <bitset>
17: #include <climits>
18: #include <cfloat>
19:
20: using namespace std;
21:
22: #define SN scanf("\n")
23: #define SI(x) scanf("%d",&x)
24: #define SII(x,y) scanf("%d %d",&x,&y)
25: #define SL(x) scanf("%lld",&x)
26: #define SD(x) scanf("%lf",&x)
27: #define SC(x) scanf("%c",&x)
28: #define FOR(i, s, k) for(int i=s; i<k; i++)
29: #define REP(i, n) FOR(i, 0, n)
30: #define INF INT_MAX
31: #define EPS 1e-9
32: #define PI acos(-1)
33:
34: typedef long long int lint;
35: typedef unsigned long long int ulint;
36: typedef pair<int, int> ii;
37: typedef pair<double, int> di;
38: typedef vector<int> vi;
39: typedef vector<lint> vl;
40: typedef vector<double> vd;
41: typedef vector<bool> vb;
42: typedef list<int> li;
43: typedef vector<vector<int>> vvi;
44: typedef vector<vector<double>> vvd;
45: typedef vector<pair<int, int>> vii;
46: typedef list<pair<int, int>> lii;
47: typedef vector<list<int>> vli;
48: typedef vector<list<pair<int, int>>> vlai;
49: typedef vector<list<pair<double, int>>> vldi;
50:
51: /* Union Find Integer */
52: map<int, pair<int, unsigned int>> Sets;
53: void AddSet(int x){ Sets.insert(make_pair(x, make_pair(x, 1))); }
54: int Find(int x){
55:     if(Sets[x].first == x){ return x; }
56:     else{ return Sets[x].first = Find(Sets[x].first); }
57: }
58: void Union(int x, int y) {
59:     int parentX = Find(x), parentY = Find(y);
60:     int rankX = Sets[parentX].second, rankY = Sets[parentY].second;
61:     if (parentX == parentY){ return; }
62:     else if(rankX < rankY){
63:         Sets[parentX].first = parentY;
64:         Sets[parentY].second += Sets[parentX].second;
65:     }else{
66:         Sets[parentY].first = parentX;
67:         Sets[parentX].second += Sets[parentY].second;
68:     }
69: }

```

```

70: int Size(int x){ return Sets[Find(x)].second; }
71: void Reset(){ Sets.clear(); }
72:
73: /* DFS */
74: void DFS(vl ii &adj, vb &visited, int x){
75:     cout<<x;
76:     visited[x] = true;
77:     for(ii e : adj[x]){
78:         int y = e.second;
79:         int w = e.first;
80:         if(!visited[y]){
81:             DFS(adj, visited, y);
82:         }
83:     }
84: }
85:
86: /* Topological sort */
87: vi topologicalSort(vl ii adj){
88:     int n = adj.size();
89:     vi o(n,-1); vi pred(n,0);
90:     REP(i,n){
91:         for(ii e : adj[i]){
92:             int j = e.second;
93:             pred[j]++;
94:         }
95:     }
96:     stack<int> S; int i = 1;
97:     REP(u,n){
98:         if(pred[u]==0){
99:             if(o[u]==-1){ S.push(u); }
100:             while(!S.empty()){
101:                 int v = S.top(); S.pop();
102:                 o[v] = i; i++;
103:                 for(ii e : adj[v]){
104:                     int y = e.second;
105:                     pred[y]--;
106:                     if(pred[y]==0){ S.push(y); }
107:                 }
108:             }
109:         }
110:     }
111:     return o;
112: }
113:
114: /* Kruskal's Algorithm (Minimum spanning tree) */
115: typedef struct Edge{
116:     ii e;
117:     int w;
118: } Edge;
119: bool compareWeight(Edge a, Edge b){ return a.w < b.w; }
120: vector<Edge> Kruskal(vvi adj){
121:     int n = adj.size();
122:     vector<Edge> mst;
123:     vector<Edge> edges;
124:     REP(i,n){
125:         REP(j,n){
126:             if(adj[i][j]!=0){
127:                 edges.push_back({make_pair(j,i),adj[i][j]});
128:             }
129:         }
130:     }
131:     sort(edges.begin(), edges.end(), compareWeight);
132:     REP(i,n){ AddSet(i); }
133:     for(Edge e : edges){
134:         if(Find(e.e.first)!=Find(e.e.second)){
135:             mst.push_back(e);
136:             Union(e.e.first,e.e.second);
137:         }
138:     }

```

```

139:   Reset();
140:   return mst;
141: }
142:
143: /* Prim's Algorithm (Minimum spanning tree) */
144: int Prim(vlii adj){
145:     int n = adj.size();
146:     vb visited(n, false);
147:     priority_queue<ii, vii, greater<ii>> Q;
148:     int cost = 0;
149:     Q.push(make_pair(0,0));
150:
151:     while(!Q.empty()){
152:         ii p = Q.top(); Q.pop();
153:         int v = p.second;
154:         int w = p.first;
155:
156:         if(!visited[v]){
157:             cost += w;
158:             visited[v]=true;
159:             for(ii nei : adj[v]){
160:                 if(!visited[nei.second]){
161:                     Q.push(nei);
162:                 }
163:             }
164:         }
165:     }
166:     return cost;
167: }
168:
169: /* Dijkstra's Algorithm (Shortest paths from source) */
170: vi Dijkstra(vlii adj, int src){
171:     int n = adj.size();
172:     priority_queue<ii, vii, greater<ii>> PQ;
173:     vi dist(n, INF);
174:     vi parent(n, -1);
175:     dist[src] = 0;
176:     PQ.push(make_pair(0,src));
177:     while(!PQ.empty()){
178:         int u = PQ.top().second; PQ.pop();
179:         for(ii p : adj[u]){
180:             int v = p.second; int w = p.first;
181:             if(dist[u]+w<dist[v]){
182:                 dist[v] = dist[u]+w;
183:                 parent[v] = u;
184:                 PQ.push(make_pair(dist[v],v));
185:             }
186:         }
187:     }
188:     return dist;
189: }
190:
191: /* Bellman-Ford's Algorithm (Shortest paths from source and negative weight) */
192: pair<bool,vd> BellmanFordCycle(vldi adj, int src){
193:     int n = adj.size();
194:     deque<int> Q, Qp;
195:     vd dist(n,DBL_MAX);
196:     vi parent(n, -1);
197:     dist[src] = 0;
198:     Q.push_back(src);
199:     REP(i,n){
200:         while(!Q.empty()){
201:             int v;
202:             v = Q.front(); Q.pop_front();
203:
204:             for(di p : adj[v]){
205:                 int w = p.second; double c = p.first;
206:                 if(dist[v]+c<dist[w]){
207:                     dist[w] = dist[v]+c;

```

```

208:         parent[w] = v;
209:         if (find(Qp.begin(), Qp.end(), w) == Qp.end()) {
210:             Qp.push_back(w);
211:         }
212:     }
213: }
214: }
215:     swap(Q, Qp);
216: }
217: return make_pair(!Q.empty(), dist);
218: }
219:
220:
221: /* Ford-Fulkerson's Algorithm (Maximum Flow) */
222: bool FordFulkersonBFS(vvi residualAdj, int s, int t, vi &parent){
223:     int n = residualAdj.size();
224:     vb visited(n, false);
225:     queue<int> q;
226:     q.push(s);
227:     visited[s] = true;
228:     parent[s] = -1;
229:
230:     while(!q.empty()){
231:         int u = q.front(); q.pop();
232:         REP(v, n){
233:             if(!visited[v] && residualAdj[u][v]>0) {
234:                 q.push(v);
235:                 parent[v] = u;
236:                 visited[v] = true;
237:             }
238:         }
239:     }
240:     return visited[t];
241: }
242: int FordFulkerson(vvi adj, int s, int t) {
243:     int n = adj.size();
244:     vvi residualAdj(n, vi(n));
245:     REP(i, n){ REP(j, n){ residualAdj[i][j] = adj[i][j]; } }
246:     vi parent(n);
247:     int maxFlow = 0;
248:
249:     while(FordFulkersonBFS(residualAdj, s, t, parent)){
250:         int pathFlow = INF;
251:         int u;
252:         int v = t;
253:         while(v != s){
254:             u = parent[v];
255:             pathFlow = min(pathFlow, residualAdj[u][v]);
256:             v = u;
257:         }
258:
259:         v = t;
260:         while(v != s){
261:             u = parent[v];
262:             residualAdj[u][v] -= pathFlow;
263:             residualAdj[v][u] += pathFlow;
264:             v = u;
265:         }
266:         maxFlow += pathFlow;
267:     }
268:     return maxFlow;
269: }
270:
271: /* Longest common subsequence non recursive */
272: // "abcd", "wabwd" => 3
273: int lcs(string a, string b){
274:     int m=a.size();
275:     int n=b.size();
276:     vvi memo(m+1, vi(n+1));

```

```

277:
278:     REP(i,m+1){
279:         REP(j,n+1){
280:             if(i==0 || j==0){
281:                 memo[i][j]=0;
282:             }else if(a[i-1] == b[j-1]){
283:                 memo[i][j]=memo[i-1][j-1]+1;
284:             }else{
285:                 memo[i][j]=max(memo[i-1][j],memo[i][j-1]);
286:             }
287:         }
288:     }
289:     return memo[m][n];
290: }
291:
292: /* Longest increasing subsequence O(n^2) */
293: int lis1(vi nums) {
294:     int n = nums.size();
295:     vi d(n,1);
296:     REP(i,n){
297:         REP(j,i){
298:             if(nums[j]<nums[i]) d[i]=max(d[i],d[j]+1);
299:         }
300:     }
301:
302:     int ans = d[0];
303:     FOR(i,1,n) ans = max(ans, d[i]);
304:     return ans;
305: }
306:
307: /* Longest increasing subsequence O(n.log(n)) */
308: int lis2(vi nums) {
309:     int n = nums.size();
310:     vi d(n+1,INF); d[0]=-INF;
311:     REP(i,n){
312:         int j = upper_bound(d.begin(),d.end(),nums[i])-d.begin();
313:         if (d[j-1]<nums[i] && nums[i]<d[j]) d[j] = nums[i];
314:     }
315:
316:     int ans = 0;
317:     REP(i,n+1){
318:         if (d[i]<INF) ans = i;
319:     }
320:     return ans;
321: }
322:
323: /* Compute positive modulo */
324: lint modulo(lint a,lint b){ return (a%b+b)%b; }
325:
326: /* Compute GCD */
327: lint gcd(lint a, lint b){
328:     if(b==0){
329:         return a;
330:     }else{
331:         return gcd(b, a%b);
332:     }
333: }
334:
335: /* Compute LCM */
336: lint lcm(lint a, lint b){
337:     return a*b/gcd(a,b);
338: }
339:
340: /* Find Bezeout relation */
341: pair<lint,pair<lint,lint>> bezout(lint a, lint b){
342:     lint s = 0; lint sp = 1;
343:     lint t = 1; lint tp = 0;
344:     lint r = b; lint rp = a;
345:     lint q, temp;

```

```

346:  while(r!=0){
347:      q = rp/r;
348:      temp=rp; rp=r; r=temp-q*rp;
349:      temp=sp; sp=s; s=temp-q*sp;
350:      temp=tp; tp=t; t=temp-q*tp;
351:  }
352:  return make_pair(rp,make_pair(sp,tp));
353: }
354:
355: /* Compute inverse modulo m of a O(n.log(n)) */
356: lint modularInverse(lint a, lint m){
357:     lint m0 = m, t, q;
358:     lint x0 = 0, x1 = 1;
359:     if(m==1){ return 0; }
360:
361:     //extended Euclid algorithm
362:     while(a>1){
363:         q = a/m;
364:         t = m;
365:         m = a%m;
366:         a = t;
367:         t = x0;
368:         x0 = x1-q*x0;
369:         x1 = t;
370:     }
371:
372:     if(x1<0){ x1 += m0; } //make x1 positive
373:     return x1;
374: }
375:
376: /* Chinese Remainder Theorem O(n.log(n)) */
377: //returns the smallest x s.t.
378: //x = a[i] (mod r[i]) for all i between 0 and n-1
379: //assumption: a[i]s are pairwise coprime
380: lint chineseRemainder(vl a, vl r){
381:     int n = a.size();
382:     uint prod=1; REP(i,n){ prod*=a[i]; }
383:
384:     lint result = 0;
385:     REP(i,n){
386:         lint pp = prod/a[i];
387:         result += r[i]*modularInverse(pp, a[i])*pp;
388:     }
389:     return result%prod;
390: }
391:
392: /* Generate all permutations O(n!.n) */
393: void permutations(vi currentConfig){
394:     int m = currentConfig.size();
395:
396:     //things to do for current perm
397:     REP(i,m){cout<<currentConfig[i];}cout<<endl;
398:
399:     vi nextConfig = currentConfig;
400:     //Find largest k s.t. a[k]<a[k+1]
401:     int k=m-1-1; while(k>=0 && nextConfig[k]>=nextConfig[k+1]){ k--; }
402:     //If k does not exist then this is the last permutation
403:     if(k<0){ return; }
404:     //Find largest l s.t. a[k]<a[l]
405:     int l=m-1; while(l>=0 && currentConfig[k]>=currentConfig[l]){ l--; }
406:     //Swap value a[k] and a[l]
407:     swap(nextConfig[k],nextConfig[l]);
408:     //Reverse sequence from a[k+1] to a[m]
409:     reverse(nextConfig.begin()+k+1, nextConfig.end());
410:
411:     permutations(nextConfig);
412: }
413:
414: /* Geometric structs */

```

```

415: typedef struct Point{
416:     double x;
417:     double y;
418: } Point;
419:
420: /* Return the angle defined by ABC (degree) */
421: double angle(Point a, Point b, Point c){
422:     double aa = pow(b.x-a.x,2) + pow(b.y-a.y,2);
423:     double bb = pow(b.x-c.x,2) + pow(b.y-c.y,2);
424:     double cc = pow(c.x-a.x,2) + pow(c.y-a.y,2);
425:     return acos((aa+bb-cc)/sqrt(4*aa*bb))*180/PI;
426: }
427:
428: /* Return the centroid of a polygon */
429: Point centroid(vector<Point> points){
430:     int n = points.size();
431:     Point centroid = {0, 0};
432:     double area = 0;
433:     double x0=0; double y0=0;
434:     double x1=0; double y1=0;
435:     double partialArea = 0.0;
436:
437:     REP(i,n){
438:         x0 = points[i].x;
439:         y0 = points[i].y;
440:         x1 = points[(i+1)%n].x;
441:         y1 = points[(i+1)%n].y;
442:         partialArea = x0*y1 - x1*y0;
443:         area += partialArea;
444:         centroid.x += (x0+x1)*partialArea;
445:         centroid.y += (y0+y1)*partialArea;
446:     }
447:
448:     area *= 0.5;
449:     centroid.x /= (6.0*area);
450:     centroid.y /= (6.0*area);
451:
452:     return centroid;
453: }
454:
455: /* Geometry algorithms from geeksforgeeks.org */
456: //given collinear points p, q, r check if q lies on pr
457: bool onSegment(Point p, Point q, Point r){
458:     if (q.x <= max(p.x, r.x) && q.x >= min(p.x, r.x) &&
459:         q.y <= max(p.y, r.y) && q.y >= min(p.y, r.y))
460:         return true;
461:     return false;
462: }
463:
464: //find orientation of ordered triplet (p, q, r)
465: //p,q,r collinear => 0
466: //clockwise => 1
467: //counterclockwise =>2
468: int orientation(Point p, Point q, Point r){
469:     int val = (q.y-p.y)*(r.x-q.x)-(q.x-p.x)*(r.y-q.y);
470:     if (val == 0) return 0;
471:     return (val > 0)? 1: 2;
472: }
473:
474: //check if line segment p1q1 and p2q2 intersect.
475: bool doIntersect(Point p1, Point q1, Point p2, Point q2) {
476:     int o1 = orientation(p1, q1, p2);
477:     int o2 = orientation(p1, q1, q2);
478:     int o3 = orientation(p2, q2, p1);
479:     int o4 = orientation(p2, q2, q1);
480:
481:     if (o1 != o2 && o3 != o4) return true;
482:
483:     if (o1 == 0 && onSegment(p1, p2, q1)) return true;

```

```

484:     if (o2 == 0 && onSegment(p1, q2, q1)) return true;
485:     if (o3 == 0 && onSegment(p2, p1, q2)) return true;
486:
487:     if (o4 == 0 && onSegment(p2, q1, q2)) return true;
488:     return false;
489: }
490:
491: //check if p lies inside the polygon
492: bool isInside(vector<Point> polygon, Point p){
493:     int n = polygon.size();
494:     if (n < 3) return false;
495:
496:     Point extreme = {10000, p.y+1001};
497:     int count = 0, i = 0;
498:     do{
499:         int next = (i+1)%n;
500:         if (doIntersect(polygon[i], polygon[next], p, extreme)){
501:             if (orientation(polygon[i], p, polygon[next]) == 0)
502:                 return onSegment(polygon[i], p, polygon[next]);
503:             count++;
504:         }
505:         i = next;
506:     } while (i != 0);
507:
508:     return count&1; //same as (count%2 == 1)
509: }
510:
511: /* Grahamâ\200\231s scan (convex hull) O(n.log(n)) from geeksforgeeks.org */
512: Point p0;
513: //utility function to find next to top in a stack
514: Point nextToTop(stack<Point> &S){
515:     Point p = S.top();
516:     S.pop();
517:     Point res = S.top();
518:     S.push(p);
519:     return res;
520: }
521: //utility function to swap two points
522: void swap(Point &p1, Point &p2){
523:     Point temp = p1;
524:     p1 = p2;
525:     p2 = temp;
526: }
527: //utility function for distance between p1 and p2
528: int distSq(Point p1, Point p2) {
529:     return (p1.x-p2.x)*(p1.x-p2.x)+(p1.y-p2.y)*(p1.y-p2.y);
530: }
531: //utility function to sort an array of points
532: int compare(const void *vp1, const void *vp2){
533:     Point *p1 = (Point *)vp1;
534:     Point *p2 = (Point *)vp2;
535:
536:     int o = orientation(p0, *p1, *p2);
537:     if(o==0) return (distSq(p0, *p2) >= distSq(p0, *p1))? -1 : 1;
538:
539:     return (o == 2)? -1: 1;
540: }
541: //returns convex hull of a set of n points.
542: vector<Point> convexHull(vector<Point> points){
543:     int n = points.size();
544:     int ymin = points[0].y, min = 0;
545:     for(int i = 1; i < n; i++){
546:         int y = points[i].y;
547:         if((y < ymin) || (ymin == y && points[i].x < points[min].x)){
548:             ymin = points[i].y, min = i;
549:         }
550:     }
551:     swap(points[0], points[min]);
552:

```



```

553: p0 = points[0];
554: qsort(&points[1], n-1, sizeof(Point), compare);
555:
556: int m = 1;
557: for (int i=1; i<n; i++){
558:     while(i<n-1 && orientation(p0,points[i],points[i+1])==0) i++;
559:     points[m] = points[i];
560:     m++;
561: }
562:
563: stack<Point> S;
564: S.push(points[0]);
565: S.push(points[1]);
566: S.push(points[2]);
567:
568: for (int i = 3; i < m; i++){
569:     while(S.size()>1 && orientation(nextToTop(S),S.top(),points[i])!=2) S.pop();
570:     S.push(points[i]);
571: }
572:
573: vector<Point> hull;
574: while (!S.empty()){
575:     Point p = S.top();
576:     hull.push_back(p);
577:     S.pop();
578: }
579: return hull;
580: }
581:
582: int main(){
583:
584:     int t; SI(t); FOR(testcase, 1, t+1){
585:
586:         double a; string b; SD(a); SN; cin>>b;
587:
588:         vii c = {{1,3},{5,0},{1,2},{2,3}};
589:         sort(c.begin(), c.end(), [](const ii a, const ii b){
590:             if(a.second==b.second) return a.first>=b.first;
591:             else return a.second>=b.second;
592:         }); // (2;3) (1;3) (1;2) (5;0)
593:
594:         printf("%.4f\n", 2.436729092);
595:
596:         printf("Case #%d: \n", testcase);
597:     }
598:
599:     return 0;
600: }

```