\*\*\* **Binary Search Double** \*\*\*

1. #include<bits/stdc++.h>
2. **using** **namespace** std;
4. **double** area(**double** a, **double** b, **double** c){
6. **double** p = (a+b+c)/2.0;
7. **double** ans = sqrt(p\*(p - a)\*(p - b)\*(p - c));
8. **return** ans;
9. }
11. **double** bin\_search(**double** a, **double** b, **double** c, **double** r)
12. {
13. **double** lo = 0, high = a, mid = 0,step = 0;
14. **double** ans = 0;
15. **double** ABC = area(a,b,c);
17. **while**(lo - high < 1e-8 && step++ < 1000){
19. mid = (lo+high)/2;
20. **double** a1 = mid;
21. **double** b1 = (a1 \* b)/a;
22. **double** c1 = (a1 \* c)/a;
23. **double** ADE = area(a1,b1,c1);
24. **double** BDEC = ABC - ADE;
25. **double** r1 = ADE/BDEC;
27. **if**(r1 > r)high = mid;
28. **else** lo = mid;
29. }
31. **return** mid;
32. }
34. **int** main()
35. {
36. **int** t;
37. cin >> t;
39. **for**(**int** kase = 1; kase <= t; kase++){
41. **double** a,b,c,r;
42. scanf("%lf %lf %lf %lf",&a,&b,&c,&r);
44. **double** res = bin\_search(a,b,c,r);
46. printf("Case %d: ",kase);
47. printf("%lf\n",res);
49. }
50. }

**\*\*\* LIS \*\*\***

1. **int** LIS(vector<**int**>&v) {
2. set<**int**>s;
3. set<**int**>::iterator it;
4. **for**(**int** i : v){
5. it = s.lower\_bound(i);
6. **if**(it != s.end()){
7. s.erase(\*it);
8. }
9. s.insert(i);
10. }
11. **return** s.size();
12. }
14. **int** main() {
15. ios\_base::sync\_with\_stdio(0);cin.tie(0);cout.tie(0);
16. vector<**int**>v;
17. **int** n;
18. cin >> n;
19. **for**(**int** i = 0; i < n; i++){
20. **int** x;
21. cin >> x;
22. v.push\_back(x);
23. }
24. cout << LIS(v) << endl;
26. }

**\*\*\* IF GIVEN POINT IS ON THE SEGEMNT \*\*\***

1. This gives the point of intersection of two lines, but **if** we are given line segments instead of lines, we have to also recheck that the point so computed actually lies on both the line segments.
2. If the line segment is specified by points (x1, y1) and (x2, y2), then to check **if** (x, y) is on the segment we have to just check that
4. min (x1, x2) <= x <= max (x1, x2)
5. min (y1, y2) <= y <= max (y1, y2)
6. The pseudo code **for** the above implementation:
8. determinant = a1 b2 - a2 b1
9. **if** (determinant == 0)
10. {
11. // Lines are parallel
12. }
13. **else**
14. {
15. x = (c1b2 - c2b1)/determinant
16. y = (a1c2 - a2c1)/determinant
17. }

**\*\*\* Mod Inverse \*\*\***

1. **int** ext\_gcd ( **int** A, **int** B, **int** \*X, **int** \*Y ){
2. **int** x2, y2, x1, y1, x, y, r2, r1, q, r;
3. x2 = 1; y2 = 0;
4. x1 = 0; y1 = 1;
5. **for** (r2 = A, r1 = B; r1 != 0; r2 = r1, r1 = r, x2 = x1, y2 = y1, x1 = x, y1 = y ) {
6. q = r2 / r1;
7. r = r2 % r1;
8. x = x2 - (q \* x1);
9. y = y2 - (q \* y1);
10. }
11. \*X = x2; \*Y = y2;
12. **return** r2;
13. }
14. ///IF M is Prime
16. **int** x = bigmod( a, m - 2, m ); // (ax)%m = 1

19. /// IF M is not Prime
20. **int** mod\_inv(**int** a,**int** m){
21. **int** x,y;
22. **int** g = ext\_gcd(a,m,&x,&y);
23. **if**(g != 1)**return** 0;
24. **else**{
25. x = (x % m + m) % m;
26. **return** x;
27. }
28. }
30. ///Finding the modular inverse for every number modulo  m
32. **void** modInv(){
34. inv[1] = 1;
35. **for**(**int** i = 2; i < m; ++i){
36. inv[i] = (m - (m/i) \* inv[m%i] % m) % m;
37. }
38. }