1. #Template

1.1. C++ Template

1.2. Java Template

1.3. Python Template

2. Data Structures

2.1. 2D RMQ

2.2. ABI 2D

2.3. KD Tree

2.4. LiChao\_dynamic

2.5. Persistent Segment Tree

2.6. Persistent Treap

2.7. RB Tree

2.8. Rectangle Union O(n log n)

2.9. Rectilinear MST O(n log n)

2.10. Treap O(n log n)

3. Dynamic Programming

3.1. Convex Hull Optimization

3.2. Divide and Conquer Optimizations

4. Geometry

4.1. Delaunay Triangulation

4.2. Minimum Enclosing Disk O(N) expected time

4.3. Minkowski O( n+m )

4.4. Pick Theorem O(n)

4.5. Primitives

4.6. Segment Line Intersection

5. Graph

5.1. Dinic O(NM)

5.2. Dominator Tree O((N+M)logN)

5.3. DSU On Tree O(NlogN)

5.4. Heavy Light Decomposition

5.5. Hopcroft-Karp Bipartite Matching O(Msqrt(N))

5.6. Hungarian O(N^3)

5.7. Max Flow Min Cost

5.8. Minimum Arborescences O(MlogN)

5.9. Punto de Art. y Bridges O(N)

5.10. SQRT On Tree

5.11. Stable Marriage

5.12. StoerWagner O(N^3)

5.13. Tree Isomorphism O(NlogN)

6. Number Theory

6.1. Algoritmo Shanka-Tonelli (x^2 = a(mod p) )

6.2. Extended GCD ( ax+by = gcd(a,b) )

6.3. Fast Modulo Transform O(NlogN)

6.4. FFT O(NlogN)

6.5. Find a primitive root of a prime number

6.6. Floyds Cycle-Finding algorithm

6.7. Gauss O(N^3)

6.8. Inverso Modular para factorial

6.9. Inverso Modular

6.10. Josephus

6.11. Linear Recurrence Solver O( N^2logK )

6.12. Matrix Exponentiation O( N^3log(N) )

6.13. Miller-Rabin is prime ( probability test )

6.14. Modular Equations ( ax = b(n) )

6.15. Modular Multiplication of big numbers

6.16. Newton Raphston

6.17. Newton's Method

6.18. Parametric Self-Dual Simplex method O(n+m)

6.19. Phi

6.20. Pollard Rho O(sqrt(s(n))) expected

6.21. Shanks' Algorithm O( sqrt(N) ) ( a^x = b(mod m) )

6.22. Simpson Rule

6.23. Teorema Chino del Resto

7. String

7.1. Aho Corasick

7.2. Lyndon Decomposition O( N )

7.3. Manacher O( N )

7.4. Palindrome Tree O( N )

7.5. Suffix Array O( NlogN )

7.6. Suffix Automata O( N )

7.7. Tandems O( NlogN )

7.8. Z Algorithm O( N )

8. Misc

*1. #Template*

**1.1. C++ Template**

1. #define optimizar\_io ios\_base::sync\_with\_stdio(0);cin.tie(0);

2. #include <inttypes.h>

3. **static** **void** main2() {

4. **char** \*ppp;

5. printf("hello world %p\n", &ppp);

6. }

7. **static** **void** run\_with\_stack\_size(**void** (\*func)(),size\_t stsize){

8. **char** \***stack**, \*send;

9. **stack**=(**char** \*)malloc(stsize);

10. send=**stack**+stsize-16;

11. send=(**char** \*)((uintptr\_t)send/16\*16);

12. asm volatile(

13. "mov %%rsp, (%0)\n"

14. "mov %0, %%rsp\n"

15. :

16. : "r" (send));

17. func();

18. asm volatile(

19. "mov (%0), %%rsp\n"

20. :

21. : "r" (send));

22. free(**stack**);

23. }

24. **int** main() {

25. run\_with\_stack\_size(main2, 64\*1024\*1024);

26. }

**1.2. Java Template**

1. **import** java.io.IOException;

2. **import** java.math.\*;

3. **import** java.util.\*;

4.

5. **public** **class** main {

6.

7. **public** **static** **void** main(String[] args)**throws** IOException{

8. //FileReader rd = new FileReader("a.in");

9. Scanner cin = **new** Scanner(System.in);

10.

11. **while**( cin.hasNext() )

12. **int** y = cin.nextInt();

13. List<Integer> B = **new** ArrayList<Integer>();

14.

15. **int** [] C = **new** **int**[10];

16.

17. **for**( **int** i = 1; i <= 100; i += 5 ) B.add(i);

18.

19. Collections.sort(B);

20. **int** a = Collections.binarySearch(B, 7);

21.

22. B.set(2, 7);

23. BigInteger d = cin.nextBigInteger();

24.

25. System.out.println(B.get(2));

26. System.out.printf("%d", 5);

27. cin.close();

28. }

29. }

**1.3. Python Template**

1. **import** **string**

2. **import** math

3. **import** fractions

4.

5. @memoize

6. def funcion( s ):

7. print s[0:2]

8. s = sorted( s )

9. **return** s

10.

11. arr = []

12. arr.append( 5 )

13. arr.append( 1 )

14. arr = funcion(arr)

15. print arr[0:5]

16. **for** x in range(0, 10):

17. arr.append( x )

18. gg = fractions.gcd(10, 65)

19.

20. **while** True:

21. try:

22. n, c, d = raw\_input().split()

23. **import** math

24. print pow( **long**(c), **long**(n), **long**(d) )

25. except EOFError:

26. break

*2. Data Structures*

**2.1. 2D RMQ**

1. **void** build( ){ // O(n\*m\*log(n)\*log(m))

2. **for**(**int** i = 0; i < n ; i ++){

3. **for**(**int** j = 0; j < m ; j ++)

4. table[0][0][i][j] = Matrix[i][j];

5. **for**(**int** lj = 1; lj <= log2( m ); lj ++)

6. **for**(**int** j = 0; j + (1<<(lj-1)) < m; j ++)

7. table[0][lj][i][j] =

8. min(table[0][lj-1][i][j],

9. table[0][lj-1][i][j+(1<<(lj-1))]);

10. }

11. **for**(**int** li = 1; li <= log2(n); li ++ )

12. **for**(**int** i = 0; i < n; i ++ )

13. **for**(**int** lj = 0; lj <= log2(m); lj++ )

14. **for**(**int** j = 0; j < m; j ++ )

15. table[li][lj][i][j] =

16. min(table[li-1][lj][i][j],

17. table[li-1][lj][i+(1<<(li-1))][j]);

18. }

19. **int** Query(**int** x1,**int** y1,**int** x2,**int** y2){

20. **int** lenx=x2-x1+1;

21. **int** kx=log2(lenx);

22. **int** leny=y2-y1+1;

23. **int** ky=log2(leny);

24. **int** min\_R1 = min ( table[kx][ky][x1][y1] ,

25. table[kx][ky][x1][y2-(1<<ky) + 1] );

26. **int** min\_R2 = min ( table[kx][ky][x2-(1<<kx) + 1][y1],

27. table[kx][ky][x2-(1<<kx)+1][y2-(1<<ky)+1] );

28. **return** min ( min\_R1, min\_R2 );

29. }

**2.2. ABI 2D**

1. **vector** <**int**> V[1000003], tree[1000003];

2. **void** init ( ) {

3. **for** ( **int** i = 1; i <= N; i ++ )

4. **if** ( V[i].size() > 1 ) {

5. V[i] = **vector** <**int**> ( 1, 0 );

6. tree[i] = **vector** <**int**> ( 1, 0 );

7. }

8. **for** ( **int** i = 1; i <= n; i ++ )

9. **for** ( **int** j = P[i].y; j <= N; j += (j&-j) ){

10. V[j].push\_back ( P[i].z );

11. tree[j].push\_back ( 0 );

12. }

13. }

14. **void** update ( **int** x, **int** y, **int** v ) {

15. **int** lo, l;

16. **for** ( **int** i = x; i <= N; i += (i&-i) ) {

17. lo = lower\_bound ( V[i].begin(), V[i].end(), y )

18. - V[i].begin();

19. l = V[i].size();

20. **for** ( **int** j = lo; j < l; j += (j&-j) )

21. tree[i][j] = max ( tree[i][j], v );

22. }

23. }

24. **int** query ( **int** x, **int** y ) {

25. **if** ( x <= 0 || y <= 0 ) **return** 0;

26. **int** ret = 0, lo;

27. **for** ( **int** i = x-1; i > 0; i -= (i&-i) ) {

28. lo = lower\_bound ( V[i].begin(), V[i].end(), y )

29. - V[i].begin() - 1;

30. **for** ( **int** j = lo; j > 0; j -= (j&-j) )

31. ret = max ( ret, tree[i][j] );

32. }

33. **return** ret;

34. }

**2.3. KD Tree**

1. **struct** point {

2. **int** x, y;

3. } P[maxn];

4. **bool** cmpx ( **const** point &a, **const** point &b ) {

5. **return** a.x < b.x;

6. }

7. **bool** cmpy ( **const** point &a, **const** point &b ) {

8. **return** a.y < b.y;

9. }

10. **inline** ll dist ( point a, point b ) {

11. **return** 1ll\*(a.x-b.x)\*(a.x-b.x)+1ll\*(a.y-b.y)\*(a.y-b.y);

12. }

13. **struct** kd {

14. kd \*h1, \*h2;

15. point p;

16. }\*KD;

17. **void** init ( **int** ini, **int** fin, kd \*nod, **int** split ) {

18. sort ( P+ini, P+1+fin, (!split)?cmpx : cmpy );

19. **int** piv = ( ini+fin )>> 1;

20. nod->p = P[piv];

21. **if** ( ini < piv ) {

22. nod->h1 = **new** kd();

23. init ( ini, piv-1, nod->h1, split^1 );

24. }

25. **if** ( piv+1 <= fin ) {

26. nod->h2 = **new** kd();

27. init ( piv+1, fin, nod->h2, split^1 );

28. }

29. }

30. ll best;

31. **void** query ( kd \*nod, point p, **int** split ) {

32. best = min ( best, dist ( p, nod->p ) );

33. ll tmp = ( !split )? p.x - nod->p.x : p.y - nod->p.y;

34. **if** ( tmp < 0 ) {

35. **if** ( nod->h1 )

36. query ( nod->h1, p, split^1 );

37. **if** ( nod->h2 && tmp\*tmp < best )

38. query ( nod->h2, p, split^1 );

39. } **else** {

40. **if** ( nod->h2 )

41. query ( nod->h2, p, split^1 );

42. **if** ( nod->h1 && tmp\*tmp < best )

43. query ( nod->h1, p, split^1 );

44. }

45. }

**2.4. LiChao\_dynamic**

1. **struct** LiChao\_max{

2. **struct** line {

3. **int** a, b;

4. line() { a = 0; b = 0; }

5. line(**int** \_a, **int** \_b) { a = \_a; b = \_b; }

6. int64\_t eval(**int** x) { **return** a \* 1ll \* x + (int64\_t)b; }

7. };

8. **struct** node {

9. node \*l, \*r; line f;

10. node() { f = line(); l = nullptr; r = nullptr; }

11. node(**int** a,**int** b){f=line(a,b);l=nullptr;r=nullptr;}

12. node(line v) { f = v; l = nullptr; r = nullptr; }

13. };

14. **typedef** node\* pnode;

15. pnode root; **int** sz;

16. **void** init(**int** \_sz) { sz = \_sz + 1; root = nullptr; }

17. **void** add\_line(**int** a, **int** b) {

18. line v = line(a, b); insert(v, -sz, sz, root);

19. }

20. int64\_t query(**int** x) { **return** query(x, -sz, sz, root); }

21. **void** insert(line &v, **int** l, **int** r, pnode &nd){

22. **if**(!nd) { nd = **new** node(v); **return**; }

23. int64\_t trl = nd->f.eval(l), trr = nd->f.eval(r);

24. int64\_t vl = v.eval(l), vr = v.eval(r);

25. **int** mid = (l + r) >> 1;

26.

27. //max

28. **if**(trl >= vl && trr >= vr) **return**;

29. **if**(trl < vl && trr < vr){nd->f=v;**return**;}

30. **if**(trl < vl) swap(nd->f, v);

31. **if**(nd->f.eval(mid) > v.eval(mid))

32. insert(v, mid+1, r,nd->r);

33. **else** swap(nd->f, v), insert(v, l, mid, nd->l);

34.

35. /\* min

36. if(trl <= vl && trr <= vr) return;

37. if(trl > vl && trr > vr) { nd->f = v; return; }

38. if(trl > vl) swap(nd->f, v);

39. if(nd->f.eval(mid) < v.eval(mid))

40. insert(v, mid + 1, r, nd->r);

41. else swap(nd->f, v), insert(v, l, mid, nd->l); \*/

42. }

43. int64\_t query(**int** x, **int** l, **int** r, pnode &nd){

44. **if**(!nd) **return** -inf; //min-> inf

45. **if**(l == r) **return** nd->f.eval(x);

46.

47. **int** mid = (l + r) >> 1;

48. **if**(mid >= x) //min

49. **return** max(nd->f.eval(x), query(x, l, mid, nd->l));

50. **return** max(nd->f.eval(x), query(x, mid + 1, r, nd->r));

51. }

52. };

53.

**2.5. Persistent Segment Tree**

1. /\* \* query -> get the number of elements in [a,b] interval

2. who's values lie inside [x,y] interval

3. \* k\_th -> find the k\_th element in [a,b] sorted

4. interval -> O(logN) time per operation \*/

5. **struct** pst {

6. **struct** node {

7. **int** l, r, cant;

8. node ( ) { l = r = cant = 0; }

9. }tree[MAXN\*24];

10. **int** n, roots[MAXN];

11. pst ( ) { n = 0; }

12. **void** init ( **int** ini, **int** fin, **int** lv ) {

13. **if** ( ini == fin ) **return**;

14. tree[lv].l = ++n;

15. tree[lv].r = ++n;

16. **int** piv = (ini+fin)>>1;

17. init ( ini, piv, tree[lv].l );

18. init ( piv+1, fin, tree[lv].r );

19. }

20. **void** add ( **int** ini, **int** fin, **int** nod, **int** ant, **int** p ){

21. **if** ( ini == fin ) {

22. tree[nod].cant = tree[ant].cant + 1;

23. **return**;

24. }

25. **int** piv = (ini+fin)>>1;

26. **if** ( p <= piv ) {

27. tree[nod].r = tree[ant].r;

28. tree[nod].l = ++n;

29. add ( ini, piv, tree[nod].l, tree[ant].l, p );

30. } **else** {

31. tree[nod].l = tree[ant].l;

32. tree[nod].r = ++n;

33. add ( piv+1, fin, tree[nod].r, tree[ant].r, p);

34. }

35. tree[nod].cant=tree[tree[nod].l].cant

36. +tree[tree[nod].r].cant;

37. }

38. **int** query ( **int** ini, **int** fin, **int** nod, **int** a, **int** b ) {

39. **if** ( a <= ini && b >= fin ) **return** tree[nod].cant;

40. **if** ( a > fin || b < ini ) **return** 0;

41. **int** piv = (ini+fin)>>1;

42. **return** query ( ini, piv, tree[nod].l, a, b ) +

43. query ( piv+1, fin, tree[nod].r, a, b );

44. }

45. **int** query ( **int** a, **int** b, **int** x, **int** y ) {

46. **return** query ( 1, N, roots[b], x, y ) -

47. query ( 1, N, roots[a-1], x, y );

48. }

49. **int** k\_th ( **int** ini, **int** fin, **int** a, **int** b, **int** k ) {

50. **if** ( ini == fin ) **return** ini;

51. **int** piv = ( ini + fin ) >> 1;

52. **int** c= tree[tree[b].l].cant - tree[tree[a].l].cant;

53. **if** ( c >= k )

54. **return** k\_th (ini,piv,tree[a].l, tree[b].l, k );

55. **return** k\_th (piv+1,fin,tree[a].r, tree[b].r, k-c );

56. }

57. **int** k\_th ( **int** a, **int** b, **int** k ) {

58. **return** k\_th ( 1, N, roots[a], roots[b], k );

59. }

60. };

**2.6. Persistent Treap**

1. /\* Careful with memory and recommended

2. to use Garbage Collection \*/

3. **typedef** **struct** item\* pitem;

4. **struct** item {

5. **int** val, sz;

6. pitem l, r;

7. item ( ) {

8. val = 0;

9. sz = 1;

10. l = r = 0;

11. }

12. };

13. **int** sz ( pitem t ) { **return** (t)? t->sz : 0; }

14. **void** upd\_sz ( pitem t ) {

15. t->sz = sz(t->l) + sz(t->r) + 1;

16. }

17. **typedef** tuple<pitem,pitem> tupla;

18. tupla split ( pitem v, **int** k ) {

19. **if** ( !v ) **return** make\_tuple ( v, v );

20. pitem l, r, ret;

21. ret = **new** item();

22. ret->val = v->val;

23. **if** ( k >= sz(v->l) + 1 ) {

24. tie(l,r) = split ( v->r, k-sz(v->l)-1 );

25. ret->l = v->l;

26. ret->r = l;

27. upd\_sz ( ret );

28. **return** make\_tuple ( ret, r );

29. } **else** {

30. tie(l,r) = split ( v->l, k );

31. ret->r = v->r;

32. ret->l = r;

33. upd\_sz ( ret );

34. **return** make\_tuple( l, ret );

35. }

36. }

37. pitem merge ( pitem l, pitem r ) {

38. **if** ( !l ) **return** r;

39. **if** ( !r ) **return** l;

40. pitem clone = **new** item();

41. **int** tl = sz(l), tr = sz(r);

42. **if** ( rand() % (tl+tr) < tl ) {

43. clone->val = l->val;

44. clone->l = l->l;

45. clone->r = merge ( l->r, r );

46. } **else** {

47. clone->val = r->val;

48. clone->r = r->r;

49. clone->l = merge ( l, r->l );

50. }

51. upd\_sz ( clone );

52. **return** clone;

53. }

**2.7. RB Tree**

1. #include <ext/pb\_ds/assoc\_container.hpp>

2. #include <ext/pb\_ds/tree\_policy.hpp>

3. **using** **namespace** \_\_gnu\_pbds;

4. **typedef** tree<

5. **int**,

6. null\_type,

7. less<**int**>,

8. rb\_tree\_tag,

9. tree\_order\_statistics\_node\_update>

10. ordered\_set;

11. ordered\_set X; //declaracion

12. X.insert(1); // insertar

13. X.erase( X.find( 2 ) ); //eliminar

14. cout<<\*X.find\_by\_order(1)<<endl;// k-th menor elemento

15. cout<<X.order\_of\_key(-5)<<endl;//lower\_bound(cant. de menores hay)

**2.8. Rectangle Union O(n log n)**

1. **struct** rectangle {

2. ll xl, yl, xh, yh;

3. };

4. ll rectangle\_area(**vector**<rectangle> &rs) {

5. **vector**<ll> ys; // coordinate compression

6. **for** (**auto** r : rs) {

7. ys.push\_back(r.yl);

8. ys.push\_back(r.yh);

9. }

10. sort(ys.begin(), ys.end());

11. ys.erase(unique(ys.begin(), ys.end()), ys.end());

12. **int** n = ys.size(); // measure tree

13. **vector**<ll> C(8 \* n), A(8 \* n);

14. function<**void**(**int**, **int**, **int**, **int**, **int**, **int**)> aux =

15. [&](**int** a, **int** b, **int** c, **int** l, **int** r, **int** k) {

16. **if** ((a = max(a,l)) >= (b = min(b,r)))

17. **return**;

18. **if** (a == l && b == r) C[k] += c;

19. **else** {

20. aux(a, b, c, l, (l+r)/2, 2\*k+1);

21. aux(a, b, c, (l+r)/2, r, 2\*k+2);

22. }

23. **if** (C[k]) A[k] = ys[r] - ys[l];

24. **else** A[k] = A[2\*k+1] + A[2\*k+2];

25. };

26. **struct** event {

27. ll x, l, h, c;

28. };

29. **vector**<event> es;

30. **for** (**auto** r : rs) {

31. **int** l = lower\_bound(ys.begin(), ys.end(), r.yl)

32. - ys.begin();

33. **int** h = lower\_bound(ys.begin(), ys.end(), r.yh)

34. - ys.begin();

35. es.push\_back({ r.xl, l, h, +1 });

36. es.push\_back({ r.xh, l, h, -1 });

37. }

38. sort(es.begin(), es.end(), [](event a, event b)

39. {**return** a.x != b.x ? a.x < b.x : a.c > b.c;});

40. ll area = 0, prev = 0;

41. **for** (**auto** &e : es) {

42. area += (e.x - prev) \* A[0];

43. prev = e.x;

44. aux(e.l, e.h, e.c, 0, n, 0);

45. }

46. **return** area;

47. }

**2.9. Rectilinear MST O(n log n)**

1. **typedef** complex<ll> point;

2. ll rectilinear\_mst(**vector**<point> ps){

3. **vector**<**int**> id(ps.size());

4. iota(id.begin(), id.end(), 0);

5. **struct** edge{

6. **int** src, dst;

7. ll weight;

8. };

9. **vector**<edge> edges;

10. **for** (**int** s = 0; s < 2; ++s){

11. **for** (**int** t = 0; t < 2; ++t){

12. sort(id.begin(), id.end(), [&](**int** i, **int** j){

13. **return** real(ps[i] - ps[j]) <

14. imag(ps[j] - ps[i]);

15. });

16. map<ll, **int**> sweep;

17. **for** (**int** i : id){

18. **for** (**auto** it = sweep.lower\_bound(-imag(ps[i]));

19. it != sweep.end(); sweep.erase(it++)){

20. **int** j = it->second;

21. **if** (imag(ps[j] - ps[i]) < real(ps[j] - ps[i]))

22. break;

23. ll d = abs(real(ps[i] - ps[j]))

24. + abs(imag(ps[i] - ps[j]));

25. edges.push\_back({ i, j, d });

26. }

27. sweep[-imag(ps[i])] = i;

28. }

29. **for** (**auto** &p : ps)

30. p = point(imag(p), real(p));

31. }

32. **for** (**auto** &p : ps)

33. p = point(-real(p), imag(p));

34. }

35. ll cost = 0;

36. sort(edges.begin(), edges.end(), [](edge a, edge b){

37. **return** a.weight < b.weight;

38. });

39. union\_find uf(ps.size());

40. **for** (edge e : edges)

41. **if** (uf.join(e.src, e.dst))

42. cost += e.weight;

43. **return** cost;

44. }

**2.10. Treap O(n log n)**

1. **typedef** **struct** item\* pitem;

2. **struct** item {

3. **int** prio, sz;

4. par men, v;

5. **bool** rev;

6. pitem l, r;

7. item ( **int** x, **int** i ) {

8. rev = **false**;

9. prio = rand();

10. sz = 1;

11. v = par ( x,i );

12. men = par ( x,i );

13. l = r = 0;

14. }

15. }\*root;

16. **inline** **int** sz ( pitem t ) { **return** t? t->sz : 0; }

17. **inline** par val ( pitem t ) {

18. **return** t? t->men : par(1<<30,1<<30);

19. }

20. **void** updata ( pitem t ) {

21. **if** ( !t ) **return**;

22. t->sz = sz(t->l) + sz(t->r) + 1;

23. t->men = min ( { t->v, val(t->l), val(t->r) } );

24. }

25. **void** push ( pitem t ) {

26. **if** ( !t || !t->rev ) **return**;

27. swap ( t->l, t->r );

28. **if** ( t->l ) t->l->rev ^= 1;

29. **if** ( t->r ) t->r->rev ^= 1;

30. t->rev = 0;

31. }

32. tuple <pitem,pitem> split ( pitem t, **int** k ) {

33. **if** ( !t ) **return** make\_tuple(nullptr,nullptr);

34. push(t);

35. pitem l, r;

36. **if** ( k >= sz(t->l)+1 ) {

37. tie(l,r) = split ( t->r, k-sz(t->l)-1 );

38. t->r = l;

39. updata ( t );

40. **return** make\_tuple ( t, r );

41. } **else** {

42. tie(l,r) = split ( t->l, k );

43. t->l = r;

44. updata ( t );

45. **return** make\_tuple ( l, t );

46. }

47. }

48. pitem merge ( pitem l, pitem r ) {

49. **if** ( !l ) **return** r;

50. **if** ( !r ) **return** l;

51. push(l), push(r);

52. **if** ( l->prio > r->prio ) {

53. l->r = merge ( l->r, r );

54. updata(l);

55. **return** l;

56. } **else** {

57. r->l = merge ( l, r->l );

58. updata(r);

59. **return** r;

60. }

61. }

*3. Dynamic Programming*

**3.1. Convex Hull Optimization**

1. //para buscar maximo

2. **typedef** complex<ll> point;

3. **typedef** **vector**<point> hull;

4. ll cross(point a, point b){**return** imag(conj(a) \* b);}

5. ll dot(point a, point b){ **return** real(conj(a) \* b); }

6. **void** add(point a, hull &ch){

7. **for**(**int** n = (**int**)ch.size(); n > 1 &&

8. cross(ch[n-1]-ch[n-2], a-ch[n-2]) >= 0; n--)

9. ch.pop\_back();

10. ch.push\_back(a);

11. }

12. ll eval(point a, hull &ch){

13. **int** lo = 0, hi = (**int**)ch.size()-1;

14. **while**(lo < hi){

15. **int** m = (lo + hi)/2;

16. **if**( dot(ch[m], a) >= dot(ch[m+1], a) ) hi = m;

17. **else** lo = m + 1;

18. }

19. **return** dot(ch[lo], a);

20. }

21. hull merge(**const** hull &a, **const** hull &b){

22. **int** n =(**int**)a.size(), m =(**int**)b.size(), x=0, y=0;

23. hull c;

24. **while**(x < n && y < m){

25. **if**(real(a[x]) <= real(b[y])) add(a[x++], c);

26. **else** add(b[y++], c);

27. }

28. **while** (x < n) add(a[x++], c);

29. **while** (y < m) add(b[y++], c);

30. **return** c;

31. }

32. **struct** dyn{

33. **vector**<hull> H;

34. **void** add(point p){

35. hull h; h.push\_back(p);

36. **for** (**int** i = 0; i < (**int**)H.size(); ++i){

37. hull &ch = H[i];

38. **if** (ch.empty()){ ch = h; **return**; }

39. h = merge(h, ch);

40. ch.clear();

41. }

42. **if** (!h.empty()) H.push\_back(h);

43. }

44. ll query(point p){

45. ll answer = -1ll<<60;

46. **for** (**int** i = 0; i < (**int**)H.size(); ++i){

47. hull &ch = H[i];

48. **if**(ch.empty()) **continue**;

49. answer = max( answer, eval(p, ch) );

50. }

51. **return** answer;

52. }

53. };

**3.2. Divide and Conquer Optimizations**

1. **void** compute(**int** k, **int** L, **int** R, **int** optL, **int** optR){

2. **if** (L > R) **return**;

3. **int** m = (L + R) / 2, opt = -1;

4. dp[m][1] = oo;

5. **for** (**int** i = optL; i <= min(m, optR); i++){

6. i64 t = dp[i - 1][0] + w(i, m);

7. **if** (dp[m][1] > t)

8. dp[m][1] = t, opt = i;

9. }

10. compute(k, L, m - 1, optL, opt);

11. compute(k, m + 1, R, opt, optR);

12. }

*4. Geometry*

**4.1. Delaunay Triangulation**

1. /\*Incremental Randomized Expected O(NlogN)\*/

2. **int** n;

3. point P[maxn];

4. **struct** edge {

5. **int** t;

6. **int** side;

7. edge ( ) { t = -1, side = 0; }

8. edge ( **int** tt, **int** s ) { t = tt, side = s; }

9. };

10. **struct** triangle {

11. point p[3];

12. edge e[3];

13. **int** child[3];

14. triangle () {}

15. triangle(**const** point&p0,**const** point&p1,**const** point&p2){

16. p[0] = p0, p[1] = p1, p[2] = p2;

17. child[0] = child[1] = child[2] = 0;

18. }

19. **bool** inside(**const** point &pp) **const** {

20. point a = p[0]-pp, b = p[1]-pp, c = p[2]-pp;

21. **return** cross(a, b) >= 0 &&

22. cross(b, c) >= 0 &&

23. cross(c, a) >= 0;

24. }

25. };

26. triangle T[maxn\*3];

27. **int** ct;

28. **bool** is\_leaf ( **int** t ) {

29. **return** !T[t].child[0]&&!T[t].child[1]&&!T[t].child[2];

30. }

31. **void** add\_edge ( edge a, edge b ) {

32. **if** ( a.t != -1 ) T[a.t].e[a.side] = b;

33. **if** ( b.t != -1 ) T[b.t].e[b.side] = a;

34. }

35. **struct** Triangulation {

36. Triangulation ( ) {

37. **int** M = 1e5 \* 3;//multiplicar el maximo valor por 3

38. T[0]=triangle(point(-M,-M),point(M,-M),point(0,M));

39. ct = 1;

40. }

41. **int** find ( **int** t, **const** point &p ) {

42. **while** ( !is\_leaf(t) ) {

43. **for** ( **int** i = 0; i < 3; i ++ )

44. **if** (T[t].child[i]&&T[T[t].child[i]].inside(p)){

45. t = T[t].child[i];

46. break;

47. }

48. }

49. **return** t;

50. }

51. **void** add\_point ( **const** point &p ) {

52. **int** t = find ( 0, p ), tab, tbc, tca;

53. tab = ct;

54. T[ct++] = triangle ( T[t].p[0], T[t].p[1], p );

55. tbc = ct;

56. T[ct++] = triangle ( T[t].p[1], T[t].p[2], p );

57. tca = ct;

58. T[ct++] = triangle ( T[t].p[2], T[t].p[0], p );

59. add\_edge ( {tab,0}, {tbc,1} );

60. add\_edge ( {tbc,0}, {tca,1} );

61. add\_edge ( {tca,0}, {tab,1} );

62. add\_edge ( {tab,2}, T[t].e[2] );

63. add\_edge ( {tbc,2}, T[t].e[0] );

64. add\_edge ( {tca,2}, T[t].e[1] );

65. T[t].child[0] = tab;

66. T[t].child[1] = tbc;

67. T[t].child[2] = tca;

68. flip ( tab, 2 );

69. flip ( tbc, 2 );

70. flip ( tca, 2 );

71. }

72. **void** flip ( **int** ti, **int** pi ) {

73. **int** tj = T[ti].e[pi].t;

74. **int** pj = T[ti].e[pi].side;

75. **if** ( tj == -1 ) **return**;

76. **if** (!incircle(T[ti].p[0],T[ti].p[1],

77. T[ti].p[2],T[tj].p[pj]))

78. **return**;

79. **int** tk = ct;

80. T[ct++]=triangle(T[ti].p[(pi+1)%3],

81. T[tj].p[pj],T[ti].p[pi]);

82. **int** tl = ct;

83. T[ct++] = triangle ( T[tj].p[(pj+1)%3],

84. T[ti].p[pi], T[tj].p[pj] );

85. add\_edge ( {tk,0}, {tl,0} );

86. add\_edge ( {tk,1}, T[ti].e[(pi+2)%3] );

87. add\_edge ( {tk,2}, T[tj].e[(pj+1)%3] );

88. add\_edge ( {tl,1}, T[tj].e[(pj+2)%3] );

89. add\_edge ( {tl,2}, T[ti].e[(pi+1)%3] );

90. T[ti].child[0] = tk, T[ti].child[1] = tl,

91. T[ti].child[2] = 0;

92. T[tj].child[0] = tk, T[tj].child[1] = tl,

93. T[tj].child[2] = 0;

94. flip ( tk, 1 );

95. flip ( tk, 2 );

96. flip ( tl, 1 );

97. flip ( tl, 2 );

98. }

99. } delaunay;

100. **void** triangulate ( ) {

101. delaunay = Triangulation();

102. random\_shuffle ( P+1, P+1+n );

103. **for** ( **int** i = 1; i <= n; i ++ )

104. delaunay.add\_point ( P[i] );

105. }

**4.2. Minimum Enclosing Disk O(N) expected time**

1. circle circumcircle ( **const** point &a,

2. **const** point &b, **const** point &c ) {

3. **if** ( abs( cross( a - c, b - c ) ) > eps ) {

4. point o = three\_point\_circle ( a, b, c );

5. **return** { o, abs ( o - a ) };

6. }

7. point p = min ( { a, b, c } );

8. point q = max ( { a, b, c } );

9. **return** circle { (p+q)\*0.5, abs(p-q)\*0.5 };

10. }

11. circle min\_enclosing\_disk\_with\_2\_points ( **vector**<point> &p,

12. **int** n, **int** a, **int** b ) {

13. circle ret =circle {(p[a]+p[b])\*0.5,abs(p[a]-p[b])\*0.5};

14. **for** ( **int** i = 0; i <= n; i ++ ) {

15. db d = abs ( ret.p - p[i] );

16. **if** ( d <= ret.r + eps ) **continue**;

17. ret = circumcircle ( p[a], p[b], p[i] );

18. }

19. **return** ret;

20. }

21. circle min\_enclosing\_disk\_with\_1\_point ( **vector**<point> &p,

22. **int** n, **int** a ) {

23. circle ret = circle { p[a], 0 };

24. **for** ( **int** i = 0; i <= n; i ++ ) {

25. db d = abs ( ret.p - p[i] );

26. **if** ( d <= ret.r + eps ) **continue**;

27. ret =min\_enclosing\_disk\_with\_2\_points( p, i, a, i );

28. }

29. **return** ret;

30. }

31. circle min\_enclosing\_disk ( **vector**<point> &p ) {

32. srand(42);

33. random\_shuffle ( p.begin(), p.end() );

34.

35. **int** n = p.size() - 1;

36. circle ret = circle { p[0], 0 };

37. **for** ( **int** i = 1; i <= n; i ++ ) {

38. db d = abs ( ret.p - p[i] );

39. **if** ( d <= ret.r + eps ) **continue**;

40. ret = min\_enclosing\_disk\_with\_1\_point ( p, i, i );

41. }

42. **return** ret;

43. }

**4.3. Minkowski O( n+m )**

1. /\* Minkowski sum of two convex polygons.

2. Note: Polygons MUST be counterclockwise \*/

3. polygon minkowski(polygon &A, polygon &B){

4. **int** na = (**int**)A.size(), nb = (**int**)B.size();

5. **if** (A.empty() || B.empty()) **return** polygon();

6. rotate(A.begin(),

7. min\_element(A.begin(), A.end()), A.end());

8. rotate(B.begin(),

9. min\_element(B.begin(), B.end()), B.end());

10. **int** pa = 0, pb = 0;

11. polygon M;

12. **while** (pa < na && pb < nb){

13. M.push\_back(A[pa] + B[pb]);

14. **double** x = cross(A[(pa + 1) % na] - A[pa],

15. B[(pb + 1) % nb] - B[pb]);

16. **if** (x <= eps) pb++;

17. **if** (-eps <= x) pa++;

18. }

19. **while** (pa < na) M.push\_back(A[pa++] + B[0]);

20. **while** (pb < nb) M.push\_back(B[pb++] + A[0]);

21. **return** M;

22. }

**4.4. Pick Theorem O(n)**

1. /\*A = I + B/2 - 1:

2. A = Area of the polygon

3. I = Number of integer coordinates points inside

4. B = Number of integer coordinates points on the boundary

5. Polygon's vertex must have integer coordinates \*/

6. **typedef** complex<ll> point;

7. **struct** segment { point p, q; };

8. ll points\_on\_segment(**const** segment &s){

9. point p = s.p - s.q;

10. **return** \_\_gcd(abs(p.real()), abs(p.imag()));

11. }

12. //<Lattice points (not in boundary),

13. // Lattice points on boundary>

14. pair<ll, ll> pick\_theorem(polygon &P){

15. ll A = area2(P), B = 0, I = 0;

16. **for** (**int** i = 0, n = P.size(); i < n; ++i)

17. B += points\_on\_segment({P[i], P[NEXT(i)]});

18. A = abs(A);

19. I = (A - B) / 2 + 1;

20. **return** {I, B};

21. }

**4.5. Primitives**

1. /\*\* 1- Base element

2. 2- The traveling direction of the point (ccw)

3. 3- Intersection

4. 4- Distance.

5. 5- End point

6. 6- Polygon inclusion decision point

7. 7- Area of a polygon

8. 8- Perturbative deformation of the polygon

9. 9- triangulation

10. 10-Convex hull (Andrew's Monotone Chain)

11. 11-Convexity determination

12. 12-Cutting of a convex polygon

13. 13-Diameter of a convex polygon

14. 14-End point of a convex polygon

15. 15-Convex polygon inclusion decision point

16. 16-Incircle

17. 17-Closest Pair Point

18. 18-Intriangle

19. 19-Three Point Circle

20. 20-Circle\_circle\_intersect

21. 21-Tangents Point Circle

22. 22-Circle-Line-Intersection

23. 23-Centroid of a (possibly nonconvex) Polygon

24. 24-Point rotate \*\*/

25.

26. ///----1-Base element----

27. **struct** point {

28. db x, y;

29. point ( db xx = 0, db yy = 0 ): x(xx), y(yy) { }

30. point operator + ( **const** point &a ) **const** {

31. **return** { x+a.x, y+a.y };

32. }

33. point operator - ( **const** point &a ) **const** {

34. **return** { x-a.x, y-a.y };

35. }

36. point operator \* ( **const** db &c ) **const** {

37. **return** { x\*c, y\*c };

38. }

39. point operator \* ( **const** point &p ) **const** {

40. **return** { x\*p.x - y\*p.y, x\*p.y + y\*p.x };

41. }

42. point operator / ( **const** db &c ) **const** {

43. **return** { x/c, y/c };

44. }

45. point operator / ( **const** point &a ) **const** {

46. **return** point { x\*a.x + y\*a.y, y\*a.x - x\*a.y } /

47. /\*divide 2 complejos\*/( a.x\*a.x + a.y\*a.y );

48. }

49. **bool** operator < ( **const** point &a ) **const** {

50. **if** ( abs( x-a.x ) > eps )

51. **return** x+eps < a.x;

52. **return** y+eps < a.y;

53. }

54. };

55. **typedef** **vector**<point> polygon;

56. **struct** line : **public** **vector**<point> {

57. line(**const** point &a, **const** point &b) {

58. push\_back(a); push\_back(b);

59. }

60. };

61. **struct** circle {

62. point p;

63. db r;

64. };

65. db cross ( **const** point &a, **const** point &b ) {

66. **return** a.x\*b.y - a.y\*b.x;

67. }

68. db dot ( **const** point &a, **const** point &b ) {

69. **return** a.x\*b.x + a.y\*b.y;

70. }

71. db norm ( **const** point &p ) {

72. **return** dot ( p, p );

73. }

74. db abs ( **const** point &p ) {

75. **return** sqrt ( norm(p) );

76. }

77. db arg ( **const** point &p ) {

78. **return** atan2 ( p.y, p.x );

79. }

80. point conj ( **const** point &p ) {

81. **return** point { p.x, -p.y };

82. }

83. point crosspoint(**const** line &l, **const** line &m) {

84. db A = cross(l[1] - l[0], m[1] - m[0]);

85. db B = cross(l[1] - l[0], l[1] - m[0]);

86. **if** (abs(A)<eps&&abs(B)<eps) **return** m[0];//same line

87. **if** (abs(A)<eps)assert(**false**);//PRECONDITION NOT SATISFIED

88. **return** m[0] + (m[1] - m[0])\* B / A;

89. }

90.

91. //---2-The traveling direction of the point------

92. **int** ccw(point a, point b, point c) {

93. b = b-a; c = c-a;

94. **if** (cross(b, c) > 0)**return** +1; // counter clockwise

95. **if** (cross(b, c) < 0)**return** -1; // clockwise

96. **if** (dot(b, c) < 0) **return** +2; // c--a--b on line

97. **if** (norm(b) < norm(c))**return** -2;// a--b--c on line

98. **return** 0;

99. }

100.

101. ///----3-Intersection------

102. **bool** intersectLL(**const** line &l, **const** line &m) {

103. **return** abs(cross(l[1]-l[0],m[1]-m[0]))>eps//non-parallel

104. ||abs(cross(l[1]-l[0],m[0]-l[0]))<eps;//same line

105. }

106. **bool** intersectLS(**const** line &l, **const** line &s) {

107. **return** cross(l[1]-l[0], s[0]-l[0])\* // s[0] is left of l

108. cross(l[1]-l[0],s[1]-l[0])<eps;//s[1] is right of l

109. }

110. **bool** intersectLP(**const** line &l, **const** point &p) {

111. **return** abs(cross(l[1]-p, l[0]-p)) < eps;

112. }

113. **bool** intersectSS(**const** line &s, **const** line &t) {

114. **return** ccw(s[0],s[1],t[0])\*ccw(s[0],s[1],t[1]) <= 0 &&

115. ccw(t[0],t[1],s[0])\*ccw(t[0],t[1],s[1]) <= 0;

116. }

117. **bool** intersectSP(**const** line &s, **const** point &p) {

118. **return** abs(s[0]-p)+abs(s[1]-p)-abs(s[1]-s[0])<eps;

119. //triangle inequality

120. }

121.

122. ///---4-Distance-------------

123. point projection(**const** line &l, **const** point &p) {

124. db t = dot(p-l[0], l[0]-l[1]) / norm(l[0]-l[1]);

125. **return** l[0] + (l[0]-l[1])\*t;

126. }

127. point reflection(**const** line &l, **const** point &p) {

128. **return** p + point(2,0)\*(projection(l, p) - p);

129. }

130. **double** distanceLP(**const** line &l, **const** point &p) {

131. **return** abs(p - projection(l, p));

132. }

133. **double** distanceLL(**const** line &l, **const** line &m) {

134. **return** intersectLL(l, m) ? 0 : distanceLP(l, m[0]);

135. }

136. **double** distanceLS(**const** line &l, **const** line &s) {

137. **if** (intersectLS(l, s)) **return** 0;

138. **return** min(distanceLP(l, s[0]), distanceLP(l, s[1]));

139. }

140. **double** distanceSP(**const** line &s, **const** point &p) {

141. **const** point r = projection(s, p);

142. **if** (intersectSP(s, r)) **return** abs(r - p);

143. **return** min(abs(s[0] - p), abs(s[1] - p));

144. }

145. **double** distanceSS(**const** line &s, **const** line &t) {

146. **if** (intersectSS(s, t)) **return** 0;

147. **return** min(min(distanceSP(s, t[0]), distanceSP(s, t[1])),

148. min(distanceSP(t, s[0]), distanceSP(t, s[1])));

149. }

150.

151. ///---5-End point----------------------------

152. point extreme(**const** polygon &G, **const** line &l ) {

153. **int** k = 0;

154. **for** (**int** i = 1; i < (**int**)G.size(); ++i)

155. **if** (dot(G[i], l[1] - l[0]) > dot(G[k], l[1] - l[0]))

156. k = i;

157. **return** G[k];

158. }

159.

160. ///----6-Polygon inclusion decision point----

161. #define curr(G, i) G[i]

162. #define next(G, i) G[(i+1)%G.size()]

163. enum { OUT, ON, IN };

164. **int** contains(**const** polygon &G, **const** point& p) {

165. **bool** in = **false**;

166. **for** (**int** i = 0; i < (**int**)G.size(); ++i) {

167. point a = curr(G,i) - p, b = next(G,i) - p;

168. **if** (a.y > b.y) swap(a, b);

169. **if** (a.y <= 0 && 0 < b.y)

170. **if** (cross(a, b) < 0) in = !in;

171. **if** (cross(a, b) == 0 && dot(a, b) <= 0) **return** ON;

172. }

173. **return** in ? IN : OUT;

174. }

175.

176. ///----7-Area of a polygon---------------------------

177. **double** area2(**const** polygon& G) {

178. **double** A = 0;

179. **for** (**int** i = 0; i < (**int**)G.size(); ++i)

180. A += cross(curr(G, i), next(G, i));

181. **return** A;

182. }

183.

184. ///-----8-Perturbative deformation of a polygon---

185. #define prev(G,i) G[(i-1+G.size())%G.size()]

186. polygon shrink\_polygon(**const** polygon &G, **double** len) {

187. polygon res;

188. **for** (**int** i = 0; i < (**int**)G.size(); ++i) {

189. point a = prev(G,i), b = curr(G,i), c = next(G,i);

190. point u = (b - a) / abs(b - a);

191. **double** th = arg((c - b)/ u) \* 0.5;

192. res.push\_back( b + u \* point(-sin(th), cos(th))

193. \* len / cos(th) );

194. }

195. **return** res;

196. }

197.

198. ///-----9-triangulation-----------------------------

199. polygon make\_triangle(**const** point&a,**const** point&b,

200. **const** point&c){

201. polygon ret(3);

202. ret[0] = a; ret[1] = b; ret[2] = c;

203. **return** ret;

204. }

205. **bool** triangle\_contains(**const** polygon&tri,**const** point&p){

206. **return** ccw(tri[0], tri[1], p) >= 0 &&

207. ccw(tri[1], tri[2], p) >= 0 &&

208. ccw(tri[2], tri[0], p) >= 0;

209. }

210. **bool** ear\_Q(**int** i, **int** j, **int** k, **const** polygon& G) {

211. polygon tri = make\_triangle(G[i], G[j], G[k]);

212. **if** (ccw(tri[0], tri[1], tri[2]) <= 0) **return** **false**;

213. **for** (**int** m = 0; m < (**int**)G.size(); ++m)

214. **if** (m != i && m != j && m != k)

215. **if** (triangle\_contains(tri, G[m]))

216. **return** **false**;

217. **return** **true**;

218. }

219. **void** triangulate(**const** polygon& G, **vector**<polygon>& t) {

220. **const** **int** n = G.size();

221. **vector**<**int**> l, r;

222. **for** (**int** i = 0; i < n; ++i) {

223. l.push\_back( (i-1+n) % n );

224. r.push\_back( (i+1+n) % n );

225. }

226. **int** i = n-1;

227. **while** ((**int**)t.size() < n-2) {

228. i = r[i];

229. **if** (ear\_Q(l[i], i, r[i], G)) {

230. t.push\_back(make\_triangle(G[l[i]], G[i], G[r[i]]));

231. l[ r[i] ] = l[i];

232. r[ l[i] ] = r[i];

233. }

234. }

235. }

236.

237. ///---10-Convex\_hull----------------------

238. **vector**<point> convex\_hull(**vector**<point> ps) {

239. **int** n = ps.size(), k = 0;

240. sort(ps.begin(), ps.end());

241. **vector**<point> ch(2\*n);

242. **for** (**int** i = 0; i < n; ch[k++] = ps[i++]) // lower-hull

243. **while** (k >= 2 && ccw(ch[k-2], ch[k-1], ps[i]) <= 0)--k;

244. **for** (**int** i = n-2,t = k+1;i>=0;ch[k++]=ps[i--])//upper-hull

245. **while** (k >= t && ccw(ch[k-2], ch[k-1], ps[i]) <= 0)--k;

246. ch.resize(k-1);

247. **return** ch;

248. }

249.

250. ///--11-Convexity determination-------------------------

251. **bool** isconvex(**const** polygon &G) {

252. **for** (**int** i = 0; i < (**int**)G.size(); ++i)

253. **if** (ccw(prev(G, i), curr(G, i), next(G, i)) > 0)

254. **return** **false**;

255. **return** **true**;

256. }

257.

258. ///---12-Cutting of a convex polygon-----------------

259. polygon convex\_cut(**const** polygon& G, **const** line& l) {

260. polygon Q;

261. **for** (**int** i = 0; i < (**int**)G.size(); ++i) {

262. point A = curr(G, i), B = next(G, i);

263. **if** (ccw(l[0], l[1], A) != -1) Q.push\_back(A);

264. **if** (ccw(l[0], l[1], A)\*ccw(l[0], l[1], B) < 0)

265. Q.push\_back(crosspoint(line(A, B), l));

266. }

267. **return** Q;

268. }

269.

270. ///--13-Diameter of a convex polygon-------

271. #define diff(G, i) (next(G, i) - curr(G, i))

272. **double** convex\_diameter(**const** polygon &pt) {

273. **const** **int** n = pt.size();

274. **int** is = 0, js = 0;

275. **for** (**int** i = 1; i < n; ++i) {

276. **if** (pt[i].y > pt[is].y) is = i;

277. **if** (pt[i].y < pt[js].y) js = i;

278. }

279. **double** maxd = norm(pt[is]-pt[js]);

280.

281. **int** i, maxi, j, maxj;

282. i = maxi = is;

283. j = maxj = js;

284. **do** {

285. **if** (cross(diff(pt,i), diff(pt,j)) >= 0) j = (j+1) % n;

286. **else** i = (i+1) % n;

287. **if** (norm(pt[i]-pt[j]) > maxd) {

288. maxd = norm(pt[i]-pt[j]);

289. maxi = i; maxj = j;

290. }

291. } **while** (i != is || j != js);

292. **return** maxd;

293. }

294.

295. ///---14-End point of a convex polygon-----------

296. #define d(k) (dot(G[k], l.back() - \*l.begin()))

297. point convex\_extreme(**const** polygon &G, **const** line &l) {

298. **const** **int** n = G.size();

299. **int** a = 0, b = n;

300. **if** (d(0) >= d(n-1) && d(0) >= d(1)) **return** G[0];

301. **while** (a < b) {

302. **int** c = (a + b) / 2;

303. **if** (d(c) >= d(c-1) && d(c) >= d(c+1)) **return** G[c];

304. **if** (d(a+1) > d(a)) {

305. **if** (d(c+1) <= d(c) || d(a) > d(c)) b = c;

306. **else** a = c;

307. } **else** {

308. **if** (d(c+1) > d(c) || d(a) >= d(c)) a = c;

309. **else** b = c;

310. }

311. }

312.

313. **return** G[0];

314. }

315.

316. ///---15-Convex polygon inclusion decision point-------

317. ///enum { OUT, ON, IN };

318. **int** convex\_contains(**const** polygon &G, **const** point &p) {

319. **const** **int** n = G.size();

320. point g = (G[0] + G[n/3] + G[2\*n/3]) / 3.0;//inner-point

321. **int** a = 0, b = n;

322. **while** (a+1 < b) { // invariant: c is in fan g-P[a]-P[b]

323. **int** c = (a + b) / 2;

324. **if** (cross(G[a]-g, G[c]-g) > 0) { // angle < 180 deg

325. **if** (cross(G[a]-g, p-g) > 0 && cross(G[c]-g, p-g) < 0)

326. b = c;

327. **else** a = c;

328. } **else** {

329. **if** (cross(G[a]-g, p-g) < 0 && cross(G[c]-g, p-g) > 0)

330. a = c;

331. **else** b = c;

332. }

333. }

334. b %= n;

335. **if** (cross(G[a] - p, G[b] - p) < 0) **return** OUT;

336. **if** (cross(G[a] - p, G[b] - p) > 0) **return** IN;

337. **return** ON;

338. }

339.

340. ///--------16-Incircle-------------------------------

341. **bool** incircle(point a, point b, point c, point p) {

342. a = a-p; b = b-p; c = c-p;

343. **return** norm(a) \* cross(b, c)

344. + norm(b) \* cross(c, a)

345. + norm(c) \* cross(a, b) >= 0;

346. // < : inside, = cocircular, > outside

347. }

348.

349. ///--17-closestPair-------------------------------

350. pair<point,point> closestPair(polygon p) {

351. **int** n = p.size(), s = 0, t = 1, m = 2, S[n];

352. S[0] = 0, S[1] = 1;

353. sort(p.begin(), p.end()); // "p < q" <=> "p.x < q.x"

354. **double** d = norm(p[s]-p[t]);

355. **for** (**int** i = 2; i < n; S[m++] = i++)

356. **for**(**int** j = 0; j < m; j ++){

357. **if** (norm(p[S[j]]-p[i])<d)

358. d = norm(p[s = S[j]]-p[t = i]);

359. **if** (p[S[j]].x < p[i].x - d) S[j--] = S[--m];

360. }

361. **return** make\_pair( p[s], p[t] );

362. }

363.

364. ///--18-Intriangle----------------

365. **bool** intriangle(point a, point b, point c, point p) {

366. a = a-p; b = b-p; c = c-p;

367. **return** cross(a, b) >= 0 &&

368. cross(b, c) >= 0 &&

369. cross(c, a) >= 0;

370. }

371.

372. ///----19-Three Point Circle------------------------

373. point three\_point\_circle(**const** point&a,**const** point&b,

374. **const** point&c){

375. point x = (b - a)/norm(b-a), y = (c - a)/norm(c-a);

376. **return** (y-x)/(conj(x)\*y - x\*conj(y)) + a;

377. }

378.

379. ///--20-Circle\_circle\_intersect--------------------

380. pair<point, point> circle\_circle\_intersect(**const** point&c1,

381. **const** **double**& r1, **const** point& c2, **const** **double**& r2) {

382. point A = conj(c2-c1);

383. point B = ((c2-c1)\*conj(c2-c1))\*-1.0 + r2\*r2-r1\*r1 ;

384. point C = (c2-c1)\*r1\*r1;

385. point D = B\*B-A\*C\*4.0;

386.

387. complex <db> q ( D.x, D.y );

388. q = sqrt(q);

389. D = { real(q), imag(q) };

390. point z1 = (B\*-1.0+D)/(A\*2.0)+c1,

391. z2 = (B\*-1.0-D)/(A\*2.0)+c1;

392. **return** pair<point, point>(z1, z2);

393. }

394.

395. ///--21-Tangents Point Circle----------

396. **vector**<point> tangent(point p, circle c) {

397. **double** D = abs(p - c.p);

398. **if** (D + eps < c.r) **return** {};

399. point t = c.p - p;

400. **double** theta = asin( c.r / D );

401. **double** d = cos(theta) \* D;

402. t = t / abs(t) \* d;

403. **if** ( abs(D - c.r) < eps ) **return** {p + t};

404. point rot( cos(theta), sin(theta) );

405. **return** {p + t \* rot, p + t \* conj(rot)};

406. }

407.

408. ///-22-Circle-Line-Intersection------------------------

409. **vector**<point> intersectLC( line l, circle c ){

410. point u = l[0] - l[1], v = l[0] - c.p;

411. **double** a = dot(u,u), b = dot(u,v),

412. cc = dot(v,v) - c.r \* c.r;

413. **double** det = b \* b - a \* cc;

414. **if** ( det < eps ) **return** { };

415. **else** **return** { l[0] + u \* (-b + sqrt(det)) / a,

416. l[0] + u \* (-b - sqrt(det)) / a };

417. }

418.

419. ///-23--Centroid of a (possibly nonconvex) Polygon

420. point centroid(**const** polygon &poly) {

421. point c(0, 0);

422. **double** scale = 3.0 \* area2(poly);

423. **for** (**int** i = 0, n = poly.size(); i < n; ++i) {

424. **int** j = (i+1)%n;

425. c=c+(poly[i]+poly[j])\*(cross(poly[i],poly[j]));

426. }

427. **return** c / scale;

428. }

429.

430. ///-24-Point rotate--------------------------

431. **inline** point rotate(point A,**double** ang){//respect to origin

432. **double** r = sqrt(dot(A , A));

433. **double** oang = atan2(A.y , A.x);

434. **return** (point){cos(ang + oang),sin(ang + oang)} \* r;

435. }

436.

**4.6. Segment Line Intersection**

1. /// O( (N+I)log(N+I))

2. /// I-> cantidad de intersecciones

3. **int** n; db X;

4. **struct** segment {

5. line l;

6. db m, n;

7. }s[maxn];

8. **struct** event {

9. point p;

10. /// 1->inicio segmento, -1->fin segmento,

11. /// 0->interseccion

12. **int** tip, id, a, b;

13. /// a y b son los segmentos q se cortan

14. **bool** operator < ( **const** event &e ) **const** {

15. **if** ( p != e.p )

16. **return** p < e.p;

17. **return** (tip==0);

18. }

19. };

20. multiset <event> e;

21. **struct** status {

22. **int** id;

23. db get\_Y ( ) **const** {

24. **return** s[id].m \* X + s[id].n;

25. }

26. **bool** operator < ( **const** status &a ) **const** {

27. **return** get\_Y ( ) + eps < a.get\_Y ( );

28. }

29. };

30. multiset <status> st;

31. **int** sol = 0;

32. set <par> mark;

33. **void** intersectar ( **auto** it, **auto** it1 ) {

34. **if** ( it->id == it1->id ||

35. mark.count( par ( it->id, it1->id ) ) )

36. **return**;

37. **if** ( intersectSS ( s[it->id].l, s[it1->id].l ) ) {

38. point p = crosspoint ( s[it->id].l, s[it1->id].l );

39. **if** ( X-eps <= p.x ) {

40. sol ++;

41. e.insert ( event { p, 0, 0, it->id, it1->id } );

42. mark.insert ( par ( it->id, it1->id ) );

43. mark.insert ( par ( it1->id, it->id ) );

44. }

45. }

46. }

47. **void** insertar ( event i, **bool** band ) {

48. **if** ( band ) X += eps\*2.0;

49. st.insert ( status { i.id } );

50. **auto** it = st.find ( status { i.id } );

51. **if** ( band ) X -= eps\*2.0;

52. **if** ( it != st.begin() ) {

53. **auto** it1 = it;

54. it1--;

55. intersectar ( it, it1 );

56. }

57. **auto** it1 = it;

58. it1++;

59. **if** ( it1 != st.end() ) intersectar ( it, it1 );

60. }

61. **void** eliminar ( event i ) {

62. **auto** it = st.find ( status { i.id } );

63. **auto** it1 = it, it2 = it;

64. it1--, it2++;

65. **if** ( it != st.begin() && it2 != st.end() )

66. intersectar ( it1, it2 );

67. st.erase ( it );

68. }

69. **void** cruzar ( event i ) {

70. st.erase ( status { i.a } );

71. st.erase ( status { i.b } );

72. event tmp;

73. tmp.id = i.a;

74. insertar ( tmp, **true** );

75. tmp.id = i.b;

76. insertar ( tmp, **true** );

77. }

78. **void** clear ( ) {

79. sol = 0;

80. e.clear(), st.clear(), mark.clear();

81. }

82. **int** main() {

83. cin >> n;

84. point a, b;

85. **for** ( **int** i = 1; i <= n; i ++ ) {

86. cin >> a.x >> a.y >> b.x >> b.y;

87. a = rotate ( a, eps\*5.0 );

88. b = rotate ( b, eps\*5.0 );

89. **if** ( b < a ) swap ( a, b );

90. s[i].l = line ( a, b );

91. s[i].m = (a.y-b.y)/(a.x-b.x);

92. s[i].n = a.y - s[i].m\*a.x;

93. e.insert ( event { a, 1, i, 0, 0 } );

94. e.insert ( event { b, -1, i, 0, 0 } );

95. }

96. **while** ( !e.empty() ) {

97. event i = (\*e.begin());

98. e.erase ( e.begin() );

99. X = i.p.x;

100. **if** ( i.tip == 1 ) insertar ( i, **false** );

101. **if** ( i.tip == -1 ) eliminar ( i );

102. **if** ( i.tip == 0 ) cruzar ( i );

103. }

104. cout << sol;

105. clear ( );

106. }

*5. Graph*

**5.1. Dinic O(NM)**

1. **int** pos, Index[MAXN];///index = -1, pos = 0

2. **int** lv[MAXN], Id[MAXN], in, fin, n;

3. **struct** edges{ ///N cant de nodos

4. **int** nod, newn, cap, next;

5. edges( **int** a = 0, **int** b = 0, **int** c = 0, **int** e = 0 ){

6. nod = a, newn = b, cap = c, next = e;

7. }

8. **int** nextn ( **int** a ){

9. **return** ( nod == a )? newn : nod;

10. }

11. }G[MAXE];

12. ///nod, newn, cap

13. **void** insertar( **int** a, **int** b, **int** c ){

14. G[pos] = edges( a, b, c, Index[a] );

15. Index[a] = pos ++;

16. G[pos] = edges( b, a, 0, Index[b] );

17. Index[b] = pos ++;

18. }

19. **queue**<**int**> Q;

20. **bool** Bfs( **int** limt ){

21. **while**( !Q.empty() ) Q.pop();

22. fill( lv, lv + n+1, 0);

23. lv[in] = 1;

24. Q.push( in );

25. **while**( !Q.empty() ) {

26. **int** nod = Q.front();

27. Q.pop();

28. **for**( **int** i = Index[nod]; i != -1; i = G[i].next ){

29. **int** newn = G[i].newn;

30. **if**( lv[newn] != 0 || G[i].cap < limt )**continue**;

31. lv[newn] = lv[nod] + 1;

32. Q.push( newn );

33. **if**( newn == fin ) **return** **true**;

34. }

35. }

36. **return** **false**;

37. }

38. **bool** Dfs( **int** nod, **int** limt ){

39. **if**( nod == fin ) **return** **true**;

40. **for**( ; Id[nod] != -1; Id[nod] = G[Id[nod]].next ){

41. **int** newn = G[Id[nod]].newn;

42. **if**( lv[nod] + 1 == lv[newn] &&

43. G[Id[nod]].cap >= limt && Dfs( newn, limt ) ){

44. G[Id[nod]].cap -= limt;

45. G[Id[nod]^1].cap += limt;

46. **return** **true**;

47. }

48. }

49. **return** **false**;

50. }

51. **int** Dinic( ){

52. **int** flow = 0;

53. **for**( **int** limt = 1024; limt > 0; ){

54. **if**( !Bfs( limt ) ){

55. limt >>= 1;

56. **continue**;

57. }

58. **for**( **int** i = 0; i <= n; i ++ )

59. Id[i] = Index[i];

60. **while**( limt > 0 && Dfs( in, limt ) )

61. flow += limt;

62. }

63. **return** flow;

64. }

**5.2. Dominator Tree O((N+M)logN)**

1. **struct** graph{

2. **int** n;

3. **vector**<**vector**<**int**> > adj, radj, to;

4. graph(**int** n) : n(n), adj(n), radj(n), to(n) {}

5. **void** add\_edge(**int** src, **int** dst){

6. adj[src].push\_back(dst);

7. radj[dst].push\_back(src);

8. }

9. **vector**<**int**> rank, semi, low, anc;

10. **int** eval(**int** v){

11. **if** (anc[v] < n && anc[anc[v]] < n){

12. **int** x = eval(anc[v]);

13. **if** (rank[semi[low[v]]] > rank[semi[x]])

14. low[v] = x;

15. anc[v] = anc[anc[v]];

16. }

17. **return** low[v];

18. }

19. **vector**<**int**> prev, ord;

20. **void** dfs(**int** u){

21. rank[u] = ord.size();

22. ord.push\_back(u);

23. **for** (**int** i = 0; i < (**int**) adj[u].size(); ++i){

24. **int** v = adj[u][i];

25. **if** (rank[v] < n)

26. **continue**;

27. dfs(v);

28. prev[v] = u;

29. }

30. }

31. **vector**<**int**> idom; // idom[u] is an immediate dominator of u

32. **void** dominator\_tree(**int** r){

33. idom.assign(n, n);

34. prev = rank = anc = idom;

35. semi.resize(n);

36. **for** (**int** i = 0; i < n; ++i)

37. semi[i] = i;

38. low = semi;

39. ord.clear();

40. dfs(r);

41. **vector**<**vector**<**int**> > dom(n);

42. **for** (**int** x = (**int**) ord.size() - 1; x >= 1; --x){

43. **int** w = ord[x];

44. **for** (**int** j = 0; j < (**int**) radj[w].size(); ++j){

45. **int** v = radj[w][j];

46. **int** u = eval(v);

47. **if** (rank[semi[w]] > rank[semi[u]])

48. semi[w] = semi[u];

49. }

50. dom[semi[w]].push\_back(w);

51. anc[w] = prev[w];

52. **for** (**int** i=0;i<(**int**)dom[prev[w]].size();++i){

53. **int** v = dom[prev[w]][i];

54. **int** u = eval(v);

55. idom[v] = (rank[prev[w]] > rank[semi[u]]?

56. u : prev[w]);

57. }

58. dom[prev[w]].clear();

59. }

60. **for** (**int** i = 1; i < (**int**) ord.size(); ++i){

61. **int** w = ord[i];

62. **if** (idom[w] != semi[w])

63. idom[w] = idom[idom[w]];

64. }

65. }

66. **vector**<**int**> dominators(**int** u){

67. **vector**<**int**> S;

68. **for** (; u < n; u = idom[u])

69. S.push\_back(u);

70. **return** S;

71. }

72. **void** tree( ){

73. **for** (**int** i = 0; i < n; ++i){

74. **if** (idom[i] < n)

75. to[ idom[i] ].push\_back( i );

76. }

77. }

78. };

**5.3. DSU On Tree O(NlogN)**

1. **vector**<par> Q[MAXN];

2. **vector**<**int**> G[MAXN];

3. **bool** IMP[MAXN];

4. **int** cnt[MAXN], col[MAXN], lv[MAXN];

5. **int** in[MAXN], fin[MAXN], k, sz[MAXN], ord[MAXN];

6. **void** act\_color( **int** c, **int** v ){ cnt[v] ^= c; }

7. **void** Dfs( **int** nod, **int** pad ){

8. in[nod] = ++k, ord[k] = nod, sz[nod] = 1;

9. **for**( **auto** newn : G[nod] ){

10. **if**( pad == newn ) **continue**;

11. lv[newn] = lv[nod]+1;

12. Dfs( newn, nod );

13. sz[nod] += sz[newn];

14. }

15. fin[nod] = k;

16. }

17. **void** dsu\_on\_tree( **int** nod, **int** pad, **bool** keep){

18. **int** mx = -1, bigChild = -1;

19. **for**( **auto** newn : G[nod] )

20. **if**( newn != pad && sz[newn] > mx)

21. mx = sz[newn], bigChild = newn;

22. //run a dfs on small childs and clear them from cnt

23. **for**( **auto** newn : G[nod])

24. **if**(newn != pad && newn != bigChild)

25. dsu\_on\_tree(newn, nod, 0);

26. //bigChild marked as big and not cleared from cnt

27. **if**(bigChild != -1) dsu\_on\_tree(bigChild, nod, 1);

28. //update childs

29. **for**(**auto** newn : G[nod]){

30. **if**(newn == pad || newn == bigChild) **continue**;

31. **for**(**int** j = in[newn]; j <= fin[newn]; j++)

32. act\_color(col[ord[j]], lv[ord[j]] );

33. }

34. act\_color( col[nod], lv[nod] ); //update nod

35. //You can answer the queries easily.

36. //q.first -> id de la query

37. //q.second -> informacion de la query

38. **if**( keep == 1 ) **return**;

39. **for**(**int** j = in[nod]; j <= fin[nod]; j++)

40. act\_color( col[ ord[j] ], lv[ord[j]] );//clear

41. }

**5.4. Heavy Light Decomposition**

1. **vector**<**int**> V[MAXN];

2. **int** n, sz[MAXN], lv[MAXN], P[MAXN], A[MAXN], B[MAXN], C[MAXN];

3. // P: padre A: ult hoja B: pos C:cant

4. // G[i] = vector<int>( 4\*C[i], 0 );

5. // lv[1] = 1;

6. **void** Dfs( **int** nod = 1, **int** pad = 0 ){

7. **int** mej = nod;

8. A[nod] = nod;

9. **for**( **auto** i : V[nod] ){

10. **if**( i == pad ) **continue**;

11. lv[i] = lv[nod]+1;

12. Dfs( i, nod );

13. **if**( sz[i] > sz[mej] ) mej = i;

14. sz[nod] += sz[i];

15. }

16. mej = A[mej];

17. sz[nod] ++;

18. P[mej] = pad;

19. A[nod] = mej, B[nod] = C[mej];

20. C[mej] ++;

21. }

22. **int** sol;

23. **void** solve( **int** a, **int** b ){

24. **int** a1 = a, b1 = b, dist = 0;

25. **while**( A[a1] != A[b1] ){

26. **if**( lv[ P[ A[a1] ] ] > lv[ P[ A[b1] ] ] )

27. dist += lv[a1] - lv[ P[ A[a1] ] ], a1 = P[ A[a1] ];

28. **else**

29. dist += lv[b1] - lv[ P[ A[b1] ] ], b1 = P[ A[b1] ];

30. }

31. dist += abs( lv[ a1 ] - lv[ b1 ] );

32. **int** lca = ( lv[a1] > lv[b1] ) ? b1 : a1;

33.

34. sol = 0;

35. **while**( A[a] != A[lca] ){

36. sol =\_\_gcd(sol,query(A[a],0,C[A[a]]-1,1,B[a],C[A[a]]-1));

37. a = P[ A[a] ];

38. }

39.

40. sol =\_\_gcd(sol, query(A[a], 0, C[A[a]]-1, 1, B[a], B[lca]-1 ));

41.

42. **while**( A[b] != A[lca] ){

43. sol =\_\_gcd(sol,query(A[b],0,C[A[b]]-1,1,B[b],C[A[b]]-1));

44. b = P[ A[b] ];

45. }

46.

47. sol =\_\_gcd(sol, query(A[b], 0, C[A[b]]-1, 1, B[b], B[lca] ));

48. }

**5.5. Hopcroft-Karp Bipartite Matching O(Msqrt(N))**

1. **const** **int** MAXV = 1001;

2. **const** **int** MAXV1 = 2\*MAXV;

3. **vector**<**int**> ady[MAXV];

4. **int** D[MAXV1],Mx[MAXV], My[MAXV];

5. **bool** BFS(){

6. **int** u, v, i, e;

7. **queue**<**int**> cola;

8. **bool** f = 0;

9. **for** (i = 0; i < N+M; i++) D[i] = 0;

10. **for** (i = 0; i < N; i++)

11. **if** (Mx[i] == -1) cola.push(i);

12. **while** (!cola.empty()){

13. u = cola.front(); cola.pop();

14. **for** (e = ady[u].size()-1; e >= 0; e--) {

15. v = ady[u][e];

16. **if** (D[v + N]) **continue**;

17. D[v + N] = D[u] + 1;

18. **if** (My[v] != -1){

19. D[My[v]] = D[v + N] + 1;

20. cola.push(My[v]);

21. }**else** f = 1;

22. }

23. }

24. **return** f;

25. }

26. **int** DFS(**int** u){

27. **for** (**int** v, e = ady[u].size()-1; e >=0; e--){

28. v = ady[u][e];

29. **if** (D[v+N] != D[u]+1) **continue**;

30. D[v+N] = 0;

31. **if** (My[v] == -1 || DFS(My[v])){

32. Mx[u] = v; My[v] = u; **return** 1;

33. }

34. }

35. **return** 0;

36. }

37. **int** Hopcroft\_Karp(){

38. **int** i, flow = 0;

39. **for** (i = max(N,M); i >=0; i--) Mx[i] = My[i] = -1;

40. **while** (BFS())

41. **for** (i = 0; i < N; i++)

42. **if** (Mx[i] == -1 && DFS(i))

43. ++flow;

44. **return** flow;

45. }

**5.6. Hungarian O(N^3)**

1. #define MAXN 300

2. **int** N,A[MAXN+1][MAXN+1],p,q, oo = 1 <<30;

3. **int** fx[MAXN+1],fy[MAXN+1],x[MAXN+1],y[MAXN+1];

4. **int** hungarian(){

5. memset(fx,0,**sizeof**(fx));

6. memset(fy,0,**sizeof**(fy));

7. memset(x,-1,**sizeof**(x));

8. memset(y,-1,**sizeof**(y));

9. **for**(**int** i = 0; i < N; ++i)

10. **for**(**int** j = 0; j < N; ++j) fx[i] = max(fx[i],A[i][j]);

11. **for**(**int** i = 0; i < N; ){

12. **vector**<**int**> t(N,-1), s(N+1,i);

13. **for**(p = q = 0; p <= q && x[i]<0; ++p)

14. **for**(**int** k = s[p], j = 0; j < N && x[i]<0; ++j)

15. **if** (fx[k]+fy[j]==A[k][j] && t[j]<0)

16. {

17. s[++q]=y[j];

18. t[j]=k;

19. **if**(s[q]<0)

20. **for**(p=j; p>=0; j=p)

21. y[j]=k=t[j], p=x[k], x[k]=j;

22. }

23. **if** (x[i]<0){

24. **int** d = oo;

25. **for**(**int** k = 0; k < q+1; ++k)

26. **for**(**int** j = 0; j < N; ++j)

27. **if**(t[j]<0) d=min(d,fx[s[k]]+fy[j]-A[s[k]][j]);

28. **for**(**int** j = 0; j < N; ++j) fy[j]+=(t[j]<0?0:d);

29. **for**(**int** k = 0; k < q+1; ++k) fx[s[k]]-=d;

30. }

31. **else** ++i;

32. }

33. **int** ret = 0;

34. **for**(**int** i = 0; i < N; ++i) ret += A[i][x[i]];

35. **return** ret;

36. }

**5.7. Max Flow Min Cost**

1. **namespace** MaxFlowMinCost{

2. #define MAXE 1000005

3. #define MAXN 100010

4. #define oo 1e9

5. **int** pos, Index[MAXN], In, Fin, NN;///index = -1

6. **typedef** **int** type\_cost;

7. **typedef** pair<type\_cost, **int**> par;

8. type\_cost Phi[MAXN];

9. **struct** edges{

10. **int** nod, newn, cap, next;

11. type\_cost cost;

12. edges( **int** a=0,**int** b=0,**int** c=0,type\_cost d=0,**int** e=0 ){

13. nod = a, newn = b, cap = c, cost = d, next = e;

14. }

15. }G[MAXE];

16. **void** initialize( **int** cnod, **int** source, **int** sink ){

17. In = source, Fin = sink, NN = cnod;

18. memset( Index, -1, **sizeof**(Index) );

19. pos = 0;

20. }

21. ///nod, newn, cap, cost

22. **void** insertar( **int** a, **int** b, **int** c, type\_cost d ){

23. G[pos] = edges( a, b, c, d, Index[a] );

24. Index[a] = pos ++;

25. G[pos] = edges( b, a, 0, -d, Index[b] );

26. Index[b] = pos ++;

27. }

28. priority\_queue<par, **vector**<par>, greater<par> >Qp;

29. **int** F[MAXN], parent[MAXN];

30. type\_cost dist[MAXN];

31. par Max\_Flow\_Min\_Cost( ){

32. **int** FlowF = 0;

33. type\_cost CostF = 0;

34. **int** nod, newn, flow;

35. type\_cost newc, cost;

36. memset( Phi, 0, **sizeof**(Phi) );

37. **for**( ; ; ){

38. fill( F, F + 1 + NN, 0 );

39. fill( dist, dist + 1 + NN, oo );

40. F[In] = oo, dist[In] = 0;

41. Qp.push( par( 0, In ) );

42. **while**( !Qp.empty() ){

43. nod = Qp.top().second, cost = Qp.top().first;

44. Qp.pop();

45. flow = F[nod];

46. **for**( **int** i = Index[nod]; i != -1; i = G[i].next ){

47. newn = G[i].newn;

48. newc = cost + G[i].cost + Phi[nod] - Phi[newn];

49. **if**( G[i].cap > 0 && dist[newn] > newc ){

50. dist[newn] = newc;

51. F[newn] = min( flow, G[i].cap );

52. parent[newn] = i;

53. Qp.push( par( newc, newn ) );

54. }

55. }

56. }

57. **if**( F[Fin] <= 0 ) break;

58. CostF += (( dist[Fin] + Phi[Fin] ) \* F[Fin] );

59. FlowF += F[Fin];

60. **for**( **int** i = In; i <= Fin; i ++ )

61. **if**( F[i] ) Phi[i] += dist[i];

62. nod = Fin;

63. **while**( nod != In ){

64. G[parent[nod]].cap -= F[Fin];

65. G[parent[nod]^1].cap += F[Fin];

66. nod = G[parent[nod]].nod;

67. }

68. }

69. **return** par( CostF, FlowF );

70. }

71. }

**5.8. Minimum Arborescences O(MlogN)**

1. template<typename T>

2. **struct** minimum\_aborescense{

3. **struct** edge{

4. **int** src, dst;

5. T weight;

6. };

7. **vector**<edge> edges;

8. **void** add\_edge(**int** u, **int** v, T w){

9. edges.push\_back({ u, v, w });

10. }

11. T solve(**int** r){

12. **int** n = 0;

13. **for** (**auto** e : edges)

14. n = max(n, max(e.src, e.dst) + 1);

15. **int** N = n;

16. **if**( N == 0 ) **return** 0;

17. **for** (T res = 0;;){

18. **vector**<edge> in(N,{-1,-1,numeric\_limits<T>::max()});

19. **vector**<**int**> C(N, -1);

20. **for** (**auto** e : edges)

21. **if** (in[e.dst].weight > e.weight)

22. in[e.dst] = e;

23. in[r] = {r, r, 0};

24. **for** (**int** u = 0; u < N; ++u){

25. **if** (in[u].src < 0)

26. **return** numeric\_limits<T>::max();

27. res += in[u].weight;

28. }

29. **vector**<**int**> mark(N, -1);

30. **int** index = 0;

31. **for** (**int** i = 0; i < N; ++i) {

32. **if** (mark[i] != -1) **continue**;

33. **int** u = i;

34. **while** (mark[u] == -1){

35. mark[u] = i;

36. u = in[u].src;

37. }

38. **if** (mark[u] != i || u == r)

39. **continue**;

40. **for**(**int** v=in[u].src;u!=v;v=in[v].src)

41. C[v] = index;

42. C[u] = index++;

43. }

44. **if** (index == 0) **return** res;

45. **for** (**int** i = 0; i < N; ++i)

46. **if** (C[i] == -1) C[i] = index++;

47. **vector**<edge> next;

48. **for** (**auto** &e : edges)

49. **if**(C[e.src]!=C[e.dst]&&C[e.dst]!=C[r])

50. next.push\_back({C[e.src], C[e.dst],

51. e.weight-in[e.dst].weight});

52. edges.swap(next);

53. N = index;

54. r = C[r];

55. }

56. }

57. };

**5.9. Punto de Art. y Bridges O(N)**

1. **void** bridges\_PtoArt ( **int** nod ){

2. Td[nod] = low[nod] = ++ k;

3. **for**( **auto** num : V[nod] ){

4. **int** newn = G[num].nextn( nod );

5. **if**( G[num].band ) **continue**;

6. G[num].band = **true**;

7. **if**( Td[newn] ){

8. low[nod] = min( low[nod], Td[newn] );

9. **continue**;

10. }

11. bridges\_PtoArt( newn );

12. low[nod] = min( low[nod], low[newn] );

13. **if**(Td[nod] < low[newn])

14. puente.push(par( nod, newn ));

15. **if**( (Td[nod] == 1 && Td[newn] > 2 ) ||

16. ( Td[nod] != 1 && Td[nod] <= low[newn] ) )

17. Punto\_art[nod] = **true**;

18. }

19. }

**5.10. SQRT On Tree**

1. **void** Dfs( **int** nod, **int** pad ){

2. P[nod] = pad;

3. **if**( lv[nod] % 2 ) G[nod] = ++k;

4. **for**( **auto** i : V[nod] ){

5. **if**( pad == i ) **continue**;

6. lv[i] = lv[nod]+1;

7. Dfs( i, nod );

8. }

9. **if**( lv[nod] % 2 == 0 ) G[nod] = ++k;

10. }

11. **struct** r{ **int** f, s, id; } Q[MAXA]; // f <= s

12. **int** R, kk;

13. **bool** comp ( **const** r s1, **const** r s2 ){

14. **if**( G[s1.f] / R != G[s2.f] / R )

15. **return** G[s1.f] / R < G[s2.f] / R;

16. **return** G[s1.s] < G[s2.s];

17. }

18. **void** mov( **int** x, **int** y ){

19. **int** p, cant = 0;

20. **while**( x != y ){

21. kk ++;

22. **if**( lv[x] >= lv[y] ){

23. p = P[x];

24. **if**( mark[p] )

25. mark[x] = **false**, remover( A[x] );

26. **else**

27. mark[p] = **true**, add( A[p] );

28. x = p;

29. }**else**{

30. tmp[++cant] = y;

31. y = P[y];

32. }

33. }

34. **for**( **int** i = cant; i >= 1; i -- ){

35. p = tmp[i];

36. **if**( mark[p] )

37. mark[x] = **false**, remover( A[x] );

38. **else**

39. mark[p] = **true**, add( A[p] );

40. x = p;

41. }

42. }

**5.11. Stable Marriage**

1. **typedef** **vector**<**int**> vi; **typedef** **vector**<vi> vvi;

2. #define rep(i,a,b) for ( \_\_typeof(a) i=(a); i<(b); ++i)

3. vi stable\_marriage(**int** n, **int** \*\*m, **int** \*\*w){

4. **queue**<**int**> q;

5. vi at(n, 0), eng(n, -1), res(n, -1); vvi inv(n, vi(n));

6. rep(i,0,n) rep(j,0,n) inv[i][w[i][j]] = j;

7. rep(i,0,n) q.push(i);

8. **while** (!q.empty()) {

9. **int** curm = q.front(); q.pop();

10. **for** (**int** &i = at[curm]; i < n; i++) {

11. **int** curw = m[curm][i];

12. **if** (eng[curw] == -1) { }

13. **else** **if** (inv[curw][curm] < inv[curw][eng[curw]])

14. q.push(eng[curw]);

15. **else** **continue**;

16. res[eng[curw] = curm] = curw, ++i; break;

17. }

18. }

19. **return** res;

20. }

**5.12. StoerWagner O(N^3)**

1. //maximo flujo seleccionando la mejor fuente y mejor sumidero

2. **int** G[MAXN][MAXN], w[MAXN], N;

3. **bool** A[MAXN], merged[MAXN];

4. **int** StoerWagner(**int** n){

5. **int** best = 1e8;

6. **for**(**int** i=1;i<n;++i) merged[i] = 0;

7. merged[0] = 1;

8. **for**(**int** phase=1;phase<n;++phase){

9. A[0] = 1;

10. **for**(**int** i=1;i<n;++i){

11. **if**(merged[i]) **continue**;

12. A[i] = 0;

13. w[i] = G[0][i];

14. }

15. **int** prev = 0,next;

16. **for**(**int** i=n-1-phase;i>=0;--i){

17. // hallar siguiente vertice que no esta en A

18. next = -1;

19. **for**(**int** j=1;j<n;++j)

20. **if**(!A[j] && (next==-1 || w[j]>w[next]))

21. next = j;

22. A[next] = **true**;

23. **if**(i>0){

24. prev = next;

25. // actualiza los pesos

26. **for**(**int** j=1;j<n;++j) **if**(!A[j])

27. w[j] += G[next][j];

28. }

29. }

30. **if**(best>w[next]) best = w[next];

31. **for**(**int** i=0;i<n;++i){// mezcla s y t

32. G[i][prev] += G[next][i];

33. G[prev][i] += G[next][i];

34. }

35. merged[next] = **true**;

36. }

37. **return** best;

38. }

**5.13. Tree Isomorphism O(NlogN)**

1. #define all(c) (c).begin(), (c).end()

2. **struct** tree{

3. **int** n;

4. **vector**<**vector**<**int**>> adj;

5. tree(**int** n) : n(n), adj(n) {}

6. **void** add\_edge(**int** src, **int** dst){

7. adj[src].push\_back(dst);

8. adj[dst].push\_back(src);

9. }

10. **vector**<**int**> centers(){

11. **vector**<**int**> prev;

12. **int** u = 0;

13. **for** (**int** k = 0; k < 2; ++k) {

14. **queue**<**int**> q;

15. prev.assign(n, -1);

16. **for** (q.push(prev[u] = u); !q.empty(); q.pop()){

17. u = q.front();

18. **for** (**auto** v : adj[u]){

19. **if** (prev[v] >= 0) **continue**;

20. q.push(v);

21. prev[v] = u;

22. }

23. }

24. }

25. **vector**<**int**> path = { u };

26. **while** (u != prev[u])

27. path.push\_back(u = prev[u]);

28. **int** m = path.size();

29. **if** (m % 2 == 0)

30. **return** {path[m/2-1], path[m/2]};

31. **else**

32. **return** {path[m/2]};

33. }

34. **vector**<**vector**<**int**>> layer;

35. **vector**<**int**> prev;

36. **int** levelize(**int** r){

37. prev.assign(n, -1);

38. prev[r] = n;

39. layer = {{r}};

40. **while** (1){

41. **vector**<**int**> next;

42. **for** (**int** u : layer.back())

43. **for** (**int** v : adj[u]){

44. **if** (prev[v] >= 0)

45. **continue**;

46. prev[v] = u;

47. next.push\_back(v);

48. }

49. **if** (next.empty()) break;

50. layer.push\_back(next);

51. }

52. **return** layer.size();

53. }

54. };

55. **bool** isomorphic(tree S, **int** s, tree T, **int** t){

56. **if** (S.n != T.n) **return** **false**;

57. **if** (S.levelize(s) != T.levelize(t)) **return** **false**;

58.

59. **vector**<**vector**<**int**>> longcodeS(S.n + 1), longcodeT(T.n + 1);

60. **vector**<**int**> codeS(S.n), codeT(T.n);

61. **for** (**int** h = (**int**) S.layer.size() - 1; h >= 0; --h) {

62. map<**vector**<**int**>, **int**> bucket;

63. **for** (**int** u : S.layer[h]){

64. sort(all(longcodeS[u]));

65. bucket[longcodeS[u]] = 0;

66. }

67. **for** (**int** u : T.layer[h]){

68. sort(all(longcodeT[u]));

69. bucket[longcodeT[u]] = 0;

70. }

71. **int** id = 0;

72. **for** (**auto** &p : bucket) p.second = id++;

73. **for** (**int** u : S.layer[h]){

74. codeS[u] = bucket[longcodeS[u]];

75. longcodeS[S.prev[u]].push\_back(codeS[u]);

76. }

77. **for** (**int** u : T.layer[h]){

78. codeT[u] = bucket[longcodeT[u]];

79. longcodeT[T.prev[u]].push\_back(codeT[u]);

80. }

81. }

82. **return** codeS[s] == codeT[t];

83. }

84. **bool** isomorphic(tree S, tree T){

85. **auto** x = S.centers(), y = T.centers();

86. **if** (x.size() != y.size()) **return** **false**;

87. **if** (isomorphic(S, x[0], T, y[0])) **return** **true**;

88. **return** x.size() > 1 && isomorphic(S, x[1], T, y[0]);

89. }

*6. Number Theory*

**6.1. Algoritmo Shanka-Tonelli (x^2 = a(mod p) )**

1. //devuelve x (mod p) tal que x^2 = a (mod p)

2. **long** **long** solve\_quadratic( **long** **long** a, **int** p ){

3. **if**( a == 0 ) **return** 0;

4. **if**( p == 2 ) **return** a;

5. **if**( powMod(a,(p-1)/2, p) != 1 ) **return** -1;

6. **int** phi = p-1, n = 0, k = 0, q = 0;

7. **while**( phi%2 == 0 ) phi/=2, n ++;

8. k = phi;

9. **for**( **int** j = 2; j < p; j ++ )

10. **if**( powMod( j, (p-1)/2, p ) == p-1 ){

11. q = j; break;

12. }

13. **long** **long** t = powMod( a, (k+1)/2, p );

14. **long** **long** r = powMod( a, k, p );

15. **while**( r != 1 ){

16. **int** i = 0, v = 1;

17. **while**( powMod( r, v, p ) != 1 ) v \*= 2, i ++;

18. **long** **long** e = powMod( 2, n-i-1, p );

19. **long** **long** u = powMod( q, k\*e, p );

20. t = (t\*u)%p;

21. r = (r\*u\*u)%p;

22. }

23. **return** t;

24. }

**6.2. Extended GCD ( ax+by = gcd(a,b) )**

1. //devuelve x,y tal que ax+by = gcd(a,b)

2. int64 extended\_euclid( int64 a, int64 b, int64& x, int64& y ) {

3. int64 g = a;

4. x = 1, y = 0;

5. **if** ( b != 0 ) {

6. g = extended\_euclid( b, a % b, y, x );

7. y -= ( a / b ) \* x;

8. }

9. **return** g;

10. }

**6.3. Fast Modulo Transform O(NlogN)**

1. **const** **int** mod = 167772161;

2. // so the algorithm works until n = 2 ^17 = 131072

3. **const** **int** G = 3; // primitive root

4. //const int MOD = 1073872897 = 2 ^ 30 + 2 ^ 17 + 1, g = 7

5. // another good choice is MOD = 167772161 = 2^27+2^25+1, g = 3

6. // a bigger choice would be MOD = 3221225473 = 2^31+2^30+1, g = 5

7. // but it requires unsigned long long for multiplications

8. // n must be a power of two

9. // sign = 1

10. // sign = -1

11. // fast modulo transform

12. // (1) n = 2^k < 2^23

13. // (2) only predetermined mod can be used

14. // (3) Inverso Modular \*/

15. **void** fmt(**vector**<ll> &x, **int** sign = +1){

16. **int** n = x.size();

17. **for** (**int** i = 0, j = 1; j < n - 1; ++j){

18. **for** (**int** k = n >> 1; k > (i ^= k); k >>= 1);

19. **if** (j < i) swap(x[i], x[j]);

20. }

21. ll h = pow(G, (mod - 1) / n, mod);

22. **if** (sign < 0) h = inv(h, mod);

23. **for** (**int** m = 1; m < n; m \*= 2){

24. ll w = 1, wk = pow(h, n / (2 \* m), mod);

25. **for** (**int** i = 0; i < m; ++i){

26. **for** (**int** j = i; j < n; j += 2 \* m){

27. ll u = x[j], d = ( x[j + m] \* w ) % mod;

28. x[j] = (u + d)%mod;

29. x[j + m] = (u - d + mod)%mod;

30. }

31. w = w \* wk % mod;

32. }

33. }

34. **if** (sign < 0){

35. ll n\_inv = inv(n, mod);

36. **for** (**auto** &a : x) a = (a \* n\_inv) % mod;

37. }

38. }

**6.4. FFT O(NlogN)**

1. #define PI acos(-1)

2. **typedef** complex<**double**> base;

3. **void** fft (**vector**<base> & a, **int** invert){

4. **int** n = (**int**) a.size();

5. **for** (**int** i = 1, j = 0; i < n-1; ++i){

6. **for** (**int** k = n >> 1; (j ^= k) < k; k >>= 1);

7. **if** (i < j) swap(a[i], a[j]);

8. }

9. **for** (**int** len=2; len <= n; len<<=1) {

10. **double** ang = 2\*PI/len \* invert;

11. base wlen(cos(ang), sin(ang)), w(1);

12. **for** (**int** i=0; i < n; i += len, w = base(1) )

13. **for** (**int** j=0; j<len/2; ++j, w \*= wlen ){

14. base u = a[i+j], v = a[i+j+len/2] \* w;

15. a[i+j] = u + v;

16. a[i+j+len/2] = u - v;

17. } }

18. **if** (invert == -1){ **for** (**int** i=0; i<n; ++i) a[i] /= n; }

19. } //a la hora de conv. de complex a int real + o - 0.5

**6.5. Find a primitive root of a prime number**

1. // Assuming the Riemnan Hypothesis it runs in O(log^6(p)\*sqrt(p))

2. **int** generator (**int** p){

3. **vector**<**int**> fact;

4. **int** phi = p-1, n = phi;

5. **for** (**int** i=2; i\*i<=n; ++i)

6. **if** (n % i == 0){

7. fact.push\_back (i);

8. **while** (n % i == 0)

9. n /= i;

10. }

11. **if** (n > 1) fact.push\_back (n);

12. **for** (**int** res=2; res<=p; ++res){

13. **bool** ok = **true**;

14. **for** (size\_t i=0; i<fact.size() && ok; ++i)

15. ok &= powmod (res, phi / fact[i], p) != 1;

16. **if** (ok) **return** res;

17. }

18. **return** -1;

19. }

**6.6. Floyds Cycle-Finding algorithm**

1. par find\_cycle() {

2. **int** t = f(x0), h = f(t), mu = 0, lam = 1;

3. **while** (t != h) t = f(t), h = f(f(h));

4. h = x0;

5. **while** (t != h) t = f(t), h = f(h), mu++;

6. h = f(t);

7. **while** (t != h) h = f(h), lam++;

8. **return** par(mu, lam);

9. }

**6.7. Gauss O(N^3)**

1. **const** **int** oo = 0x3f3f3f3f;

2. **const** **double** eps = 1e-9;

3. **int** gauss(**vector**<**vector**<**double**>> a, **vector**<**double**> &ans){

4. **int** n = (**int**) a.size();

5. **int** m = (**int**) a[0].size() - 1;

6. **vector**<**int**> where(m, -1);

7. **for** (**int** col = 0, row = 0; col < m && row < n; ++col){

8. **int** sel = row;

9. **for** (**int** i = row; i < n; ++i)

10. **if** (abs(a[i][col]) > abs(a[sel][col])) sel = i;

11. **if** (abs(a[sel][col]) < eps) **continue**;

12. **for** (**int** i = col; i <= m; ++i)

13. swap(a[sel][i], a[row][i]);

14. where[col] = row;

15. **for** (**int** i = 0; i < n; ++i)

16. **if** (i != row){

17. **double** c = a[i][col] / a[row][col];

18. **for** (**int** j = col; j <= m; ++j)

19. a[i][j] -= a[row][j] \* c;

20. }

21. ++row;

22. }

23. ans.assign(m, 0);

24. **for** (**int** i = 0; i < m; ++i)

25. **if** (where[i] != -1)

26. ans[i] = a[where[i]][m] / a[where[i]][i];

27. **for** (**int** i = 0; i < n; ++i) {

28. **double** sum = 0;

29. **for** (**int** j = 0; j < m; ++j)

30. sum += ans[j] \* a[i][j];

31. **if** (abs(sum - a[i][m]) > eps)

32. **return** 0;

33. }

34. **for** (**int** i = 0; i < m; ++i)

35. **if** (where[i] == -1) **return** oo;

36. **return** 1;

37. }

**6.8. Inverso Modular para factorial**

1. ifact[n+1] = fact[n+1]^(mod-2)

2. ifact[n] = (ifact[n+a]\*(i+1))%mod;

**6.9. Inverso Modular**

1. ll inv(ll b, ll M){ //mcd(b,m)==1

2. ll u = 1, x = 0, s = b, t = M;

3. **while**( s ){

4. ll q = t / s;

5. swap(x -= u \* q, u);

6. swap(t -= s \* q, s);

7. }

8. **return** (x %= M) >= 0 ? x : x + M;

9. }

**6.10. Josephus**

1. // n-cantidad de personas, m es la longitud del salto.

2. // comienza en la k-esima persona.

3. ll josephus(ll n, ll m, ll k) {

4. ll x = -1;

5. **for** (ll i = n - k + 1; i <= n; ++i) x = (x + m) % i;

6. **return** x;

7. }

8. ll josephus\_inv(ll n, ll m, ll x){

9. **for** (ll i = n;; i--){

10. **if** (x == i) **return** n - i;

11. x = (x - m % i + i) % i;

12. }

13. **return** -1;

14. }

**6.11. Linear Recurrence Solver O( N^2logK )**

1. /\* x[i+n] = a[0] x[i] + a[1] x[i+1] + ... + a[n-1] x[i+n-1]

2. with initial solution x[0], x[1], ..., x[n-1]

3. Complexity: O(n^2 log k) time, O(n log k) space \*/

4. ll linear\_recurrence(**vector**<ll> a, **vector**<ll> x, ll k){

5. **int** n = a.size();

6. **vector**<ll> t(2 \* n + 1);

7. function<**vector**<ll>(ll)> rec = [&](ll k){

8. **vector**<ll> c(n);

9. **if** (k < n) c[k] = 1;

10. **else**{

11. **vector**<ll> b = rec(k / 2);

12. fill(t.begin(), t.end(), 0);

13. **for** (**int** i = 0; i < n; ++i)

14. **for** (**int** j = 0; j < n; ++j){

15. t[i+j+(k&1)] += (b[i]\*b[j])%mod;

16. t[i+j+(k&1)] %= mod;

17. }

18. **for** (**int** i = 2\*n-1; i >= n; --i)

19. **for** (**int** j = 0; j < n; ++j){

20. t[i-n+j] += (a[j]\*t[i])%mod;

21. t[i-n+j] %= mod;

22. }

23. **for** (**int** i = 0; i < n; ++i)

24. c[i] = t[i];

25. }

26. **return** c;

27. };

28. **vector**<ll> c = rec(k);

29. ll ans = 0;

30. **for** (**int** i = 0; i < x.size(); ++i){

31. ans += (c[i] \* x[i])%mod;

32. ans %= mod;

33. }

34. **return** ans;

35. }

**6.12. Matrix Exponentiation O( N^3log(N) )**

1. **typedef** **vector** <ll> vect;

2. **typedef** **vector** < vect > matrix;

3. matrix identity (**int** n) {

4. matrix A(n, vect(n));

5. **for** (**int** i = 0; i <n; i++) A[i][i] = 1;

6. **return** A;

7. }

8. matrix mul(**const** matrix &A, **const** matrix &B) {

9. matrix C(A.size(), vect(B[0].size()));

10. **for** (**int** i = 0; i < C.size(); i++)

11. **for** (**int** j = 0; j < C[i].size(); j++)

12. **for** (**int** k = 0; k < A[i].size(); k++){

13. C[i][j] += (A[i][k] \* B[k][j])%mod;

14. C[i][j] %= mod;

15. }

16. **return** C;

17. }

18. matrix powm(**const** matrix &A, ll e) {

19. **return** ( e == 0 ) ? identity(A.size()) :

20. ( e % 2 == 0 ) ? powm(mul(A, A), e/2) :

21. mul(A, powm(A, e-1));

22. }

**6.13. Miller-Rabin is prime ( probability test )**

1. **bool** suspect(ll a, **int** s, ll d, ll n) {

2. ll x = powMod(a, d, n);

3. **if** (x == 1) **return** **true**;

4. **for** (**int** r = 0; r < s; ++r) {

5. **if** (x == n - 1) **return** **true**;

6. x = mulmod(x, x, n);

7. }

8. **return** **false**;

9. }

10. // {2,7,61,0} is for n < 4759123141 (= 2^32)

11. // {2,3,5,7,11,13,17,19,23,0} is for n < 10^16 (at least)

12. **unsigned** test[] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 0 };

13. **bool** miller\_rabin(ll n) {

14. **if** (n <= 1 || (n > 2 && n % 2 == 0)) **return** **false**;

15. ll d = n - 1; **int** s = 0;

16. **while** (d % 2 == 0) ++s, d /= 2;

17. **for** (**int** i = 0; test[i] < n && test[i] != 0; i++)

18. **if** (!suspect(test[i], s, d, n))

19. **return** **false**;

20. **return** **true**;

21. }

**6.14. Modular Equations ( ax = b(n) )**

1. /\* Modular Linear Equation Solver O(log(n))

2. \* Given a, b and n, solves the equation ax = b(n)

3. \* for x. Returns the vector of solutions, all smaller

4. \* than n and sorted in increasing order. \*/

5. **vector**< **int** > msolve( **int** a, **int** b, **int** n ){

6. **if**( n < 0 ) n = -n;

7. **int** d, x, y;

8. d = extended\_euclid( a, n, x, y );

9. **vector**< **int** > r;

10. **if**( b % d ) **return** r;

11. **int** x0 = ( b / d \* x ) % n;

12. **if**( x0 < 0 ) x0 += n;

13. x0 = x0 % (n / d);

14. **for**( **int** i = 0; i < d; i++ )

15. r.push\_back( ( x0 + i \* n / d ) % n );

16. **return** r;

17. }

**6.15. Modular Multiplication of big numbers**

1. **inline** ll mulmod(ll a, ll b, ll m) {

2. ll x = 0, y = a % m;

3. **for**( ; b ; b >>= 1 ){

4. **if**( b & 1 ) x = (x + y) % m;

5. y = (y \* 2) % m;

6. }

7. **return** x;

8. }

**6.16. Newton Raphston**

1. **double** eval(**double** P[],**int** n, **double** x){

2. **double** r = 0;

3. **for**(**int** i = n - 1; i >=0; i--){

4. r\*=x;

5. r+=P[i];

6. }

7. **return** r;

8. }

9. **int** main() {

10. **int** test = 1, n;

11. **while**(scanf("%d", &n) && n) {

12. **double** a[10] = {};

13. **for**(**int** i = n; i >= 0; i--) scanf("%lf", &a[i]);

14. **double** ret[10];

15. **int** m = n;

16. **for**(**int** i = 0; i < m; i++) {

17. **double** b[10] = {}; // f'(x)

18. **for**(**int** j = 0; j <= n; j++)

19. b[j] = a[j+1]\*(j+1);

20. **double** x = 25, tx; //max\_value

21. **if**(i) x = ret[i-1];

22. **while**(**true**) {

23. **double** fx =eval(a,n+1,x),ffx =eval(b,n,x);

24. tx = x - fx/ffx;

25. **if**(fabs(fx) < 1e-8)

26. break;

27. x = tx;

28. }

29. ret[i] = x;

30. **for**(**int** j = n; j >= 0; j--)

31. a[j] = a[j] + a[j+1]\*x;

32. **for**(**int** j = 0; j <= n; j++)

33. a[j] = a[j+1];

34. n--;

35. }

36. printf("Equation %d:", test++);

37. n = m;

38. sort(ret, ret+n);

39. **for**(**int** i = 0; i < n; i++) printf(" %.4lf", ret[i]);

40. printf("\n");

41. }

42. }

43.

**6.17. Newton's Method**

1. template<**class** F, **class** G>

2. **double** find\_root(F f, G df, **double** x){

3. **for** (**int** iter = 0; iter < 100; ++iter){

4. **double** fx = f(x), dfx = df(x);

5. x -= fx / dfx;

6. **if** (fabs(fx) < 1e-12)

7. break;

8. }

9. **return** x;

10. }

**6.18. Parametric Self-Dual Simplex method O(n+m)**

1. /\* - Solve a canonical LP:

2. min. c x

3. s.t. A x <= b

4. x >= 0 \*/

5. **const** **double** eps = 1e-9, oo = numeric\_limits<**double**>::infinity();

6. **typedef** **vector**<**double**> vec;

7. **typedef** **vector**<vec> mat;

8. **double** simplexMethodPD(mat &A, vec &b, vec &c){

9. **int** n = c.size(), m = b.size();

10. mat T(m + 1, vec(n + m + 1));

11. **vector**<**int**> base(n + m), row(m);

12. **for**(**int** j = 0; j < m; ++j){

13. **for** (**int** i = 0; i < n; ++i)

14. T[j][i] = A[j][i];

15. T[j][n + j] = 1;

16. base[row[j] = n + j] = 1;

17. T[j][n + m] = b[j];

18. }

19. **for** (**int** i = 0; i < n; ++i) T[m][i] = c[i];

20. **while** (1){

21. **int** p = 0, q = 0;

22. **for** (**int** i = 0; i < n + m; ++i)

23. **if** (T[m][i] <= T[m][p]) p = i;

24. **for** (**int** j = 0; j < m; ++j)

25. **if** (T[j][n + m] <= T[q][n + m]) q = j;

26. **double** t = min(T[m][p], T[q][n + m]);

27. **if** (t >= -eps) {

28. vec x(n);

29. **for** (**int** i = 0; i < m; ++i)

30. **if** (row[i] < n) x[row[i]] = T[i][n + m];

31. // x is the solution

32. **return** -T[m][n + m]; // optimal

33. }

34. **if** (t < T[q][n + m]){

35. // tight on c -> primal update

36. **for** (**int** j = 0; j < m; ++j)

37. **if** (T[j][p] >= eps)

38. **if** (T[j][p] \* (T[q][n + m] - t) >=

39. T[q][p] \* (T[j][n + m] - t))

40. q = j;

41.

42. **if** (T[q][p] <= eps)

43. **return** oo; // primal infeasible

44. }**else**{

45. // tight on b -> dual update

46. **for** (**int** i = 0; i < n + m + 1; ++i)

47. T[q][i] = -T[q][i];

48. **for** (**int** i = 0; i < n + m; ++i)

49. **if** (T[q][i] >= eps)

50. **if** (T[q][i] \* (T[m][p] - t) >=

51. T[q][p] \* (T[m][i] - t))

52. p = i;

53. **if** (T[q][p] <= eps)

54. **return** -oo; // dual infeasible

55. }

56. **for** (**int** i = 0; i < m + n + 1; ++i)

57. **if** (i != p) T[q][i] /= T[q][p];

58. T[q][p] = 1; // pivot(q, p)

59. base[p] = 1;

60. base[row[q]] = 0;

61. row[q] = p;

62. **for** (**int** j = 0; j < m + 1; ++j)

63. **if** (j != q){

64. **double** alpha = T[j][p];

65. **for** (**int** i = 0; i < n + m + 1; ++i)

66. T[j][i] -= T[q][i] \* alpha;

67. }

68. }

69. **return** oo;

70. }

**6.19. Phi**

1. ll phi(ll p, ll pk) { **return** pk - (pk/p); }

2. ll phi(ll n){

3. ll coprimes = (n != 1); // phi(1) = 0

4. **if** (n%2 == 0){

5. ll pk = 1;

6. **while** (n%2 == 0) n /= 2, pk \*= 2;

7. coprimes \*= phi(2, pk);

8. }

9. **for** (ll i = 3; i\*i <= n; i+=2)

10. **if** (n%i == 0){

11. ll pk = 1;

12. **while** (n%i == 0) n /= i, pk \*= i;

13. coprimes \*= phi(i, pk);

14. }

15. **if** (n > 1) coprimes \*= phi(n, n); // n is prime

16. **return** coprimes;

17. }

**6.20. Pollard Rho O(sqrt(s(n))) expected**

1. #define func(x)(mulmod(x, x+B, n)+ A )

2. ll pollard\_rho(ll n) {

3. **if**( n == 1 ) **return** 1;

4. **if**( miller\_rabin( n ) )

5. **return** n;

6. ll d = n;

7. **while**( d == n ){

8. ll A = 1 + rand()%(n-1), B = 1 + rand()%(n-1);

9. ll x = 2, y = 2;

10. d = -1;

11. **while**( d == 1 || d == -1 ){

12. x = func(x), y = func(func(y));

13. d = \_\_gcd( x-y, n );

14. }

15. }

16. **return** abs(d);

17. }

**6.21. Shanks' Algorithm O( sqrt(N) ) ( a^x = b(mod m) )**

1. //return x such that a^x = b(mod m)

2. **int** solve ( **int** a, **int** b, **int** m ){

3. **int** n = (**int**)sqrt( m + .0 )+1, an = 1;

4. **for** ( **int** i = 0; i < n; i++ )

5. an = (an \* a)%m;

6. map<**int**, **int**>vals;

7. **for** ( **int** i = 1, cur = an; i <= n; ++ i ){

8. **if** ( ! vals. count ( cur ) )

9. vals [ cur ] = i ;

10. cur = (cur \* an)%m;

11. }

12. **for** ( **int** i = 0, cur = b; i <= n; ++ i ){

13. **if** ( vals. count ( cur ) ){

14. **int** ans = vals [ cur ] \* n - i ;

15. **if** ( ans < m )**return** ans;

16. }

17. cur = (cur \* a)%m;

18. }

19. **return** -1;

20. }

**6.22. Simpson Rule**

1. // Error = O( (delta x)^4 )

2. **const** **int** ITR = 1e4; //must be an even number

3. **double** Simpson(**double** a,**double** b, **double** f(**double**)){

4. **double** s = f(a) + f(b), h = (b - a) / ITR;

5. **for** (**int** i = 1; i < ITR; ++i) {

6. **double** x = a + h \* i;

7. s += f(x)\*( i&1 ? 4 : 2);

8. }

9. **return** s \* h/3;

10. }

**6.23. Teorema Chino del Resto**

1. **int** resto\_chino (**vector**<**int**> x, **vector**<**int**> m, **int** k){

2. **int** i, tmp, MOD = 1, RES = 0;

3. **for** (i=0; i <k ; i++) MOD \*= m[i];

4. **for** (i =0; i <k ; i++){

5. tmp = MOD/m[i];

6. tmp \*= inverso\_mod(tmp, m[i]);

7. RES += (tmp\*x[i]) % MOD;

8. }

9. **return** RES % MOD;

10. }

*7. String*

**7.1. Aho Corasick**

1. **int** tree[MAXN][26], fail[MAXN];

2. **int** termina[MAXN], size = 1;

3. **void** addWord( **string** pal ){

4. **int** p = 0;

5. **for**(**char** c : pal){

6. **if**( !tree[p][c-'a'] )

7. tree[p][c-'a'] = size++;

8. p = tree[p][c-'a'];

9. }

10. //termina[p].push\_back( pal\_id );

11. termina[p] = pal.size();

12. }

13. **void** buildersuffix(){

14. **queue**<**int**> Q;

15. **for**(**int** i = 0; i < 26; i++)

16. **if**( tree[0][i] ) Q.push(tree[0][i]);

17. **while**( !Q.empty() ){

18. **int** u, v = Q.front(); Q.pop();

19. //for( auto i : termina[fail[v]] )

20. // termina[v].push\_back( i );

21. termina[v] = max(termina[v], termina[fail[v]]);

22. **for**( **int** i = 0; i < 26; i++ )

23. **if**(u = tree[v][i]){

24. fail[u] = tree[fail[v]][i];

25. Q.push( u );

26. }**else**

27. tree[v][i] = tree[fail[v]][i];

28. }

29. }

**7.2. Lyndon Decomposition O( N )**

1. /\*s = w1w2w3..wk, w1 >= w2 >=...>= wk.

2. > Menor RotaciÃ³n LexicogrÃ¡fica:Es el mayor valor

3. de i, tal que i < n, en la descomposicion de lyndon

4. de la cadena s+s, n = |s| \*/

5. **void** lyndon( **string** s ){

6. **int** n = (**int**)s.length(), i = 0;

7. **while**( i < n ){

8. **int** j = i+1, k = i;

9. **while**( j < n && s[k] <= s[j] ){

10. **if**( s[k] < s[j] ) k = i;

11. **else** k ++;

12. j ++;

13. }

14. **while**( i <= k ){

15. cout << s.substr( i, j-k )<<endl;

16. i += j-k;

17. } } }

**7.3. Manacher O( N )**

1. **int** rad[ 2 \* MAXLEN ], n;

2. **char** s[MAXLEN];

3. **void** manacher( ){ /// i%2!=0 par, i%2==0 impar

4. **int** i, j, k; /// i -> 2\*i o 2\*i+1

5. **for** ( i = 0, j = 0; i < 2 \* n - 1; i += k ) {

6. **while** ( i - j >= 0 && i + j + 1 < 2 \* n &&

7. s[(i - j)/2] == s[(i + j + 1)/2] )

8. j++;

9. rad[i] = j;

10. **for**(k = 1;k <= rad[i] && rad[i-k] != rad[i]-k;k++ )

11. rad[ i + k ] = min( rad[ i - k ], rad[i] - k );

12. j = max( j - k, 0 );

13. } }

**7.4. Palindrome Tree O( N )**

1. **struct** PalindromicTree{

2. **int** tree[MAXN][30], link[MAXN], length[MAXN], sz, ult;

3. **int** diff[MAXN], slink[MAXN], ans[MAXN], sans[MAXN];

4. **string** s;

5. **void** ini( ){

6. memset( tree, 0, **sizeof**(tree) );

7. memset( link, 0, **sizeof**(link) );

8. memset( length, 0, **sizeof**(length) );

9. length[1] = -1, link[1] = 1;

10. length[2] = 0, link[2] = 1;

11. sz = ult = 2, s.clear();

12. }

13. **int** find\_x( **int** suff, **int** p ){

14. **int** len = length[suff];

15. **while**( p - len < 1 || s[p] != s[p-len-1] )

16. suff = link[suff], len = length[suff];

17. **return** suff;

18. }

19. **void** insertar( **char** c ){

20. **int** p = s.size();

21. s.push\_back( c );

22. **int** suff = find\_x( ult, p );

23. **if**( tree[suff][c-'a'] == 0 ){

24. tree[suff][c-'a'] = ++sz;

25. length[sz] = length[suff] + 2;

26. link[sz] = ( length[sz] == 1 )? 2 :

27. tree[find\_x( link[suff], p )][c-'a'];

28. diff[sz] = length[sz]-length[link[sz]];

29. slink[sz] = ( diff[sz]!=diff[link[sz]] )?

30. link[sz] : slink[link[sz]];

31. }

32. ult = tree[suff][c-'a'];

33. }

34. **void** descomponer( **int** i ){

35. ans[i] = 1 << 30;

36. **for**(**int** v = ult; length[v]>0; v = slink[v]){

37. sans[v]= ans[i -(length[slink[v]] + diff[v])];

38. **if**(diff[v] == diff[link[v]])

39. sans[v] = min(sans[v], sans[link[v]]);

40. ans[i] = min(ans[i], sans[v] + 1);

41. }

42. }

43. }palin;

**7.5. Suffix Array O( NlogN )**

1. **int** n, \_sa[LEN], \_b[LEN], top[LEN], \_tmp[LEN];

2. **int** LCP[LEN], \*SA = \_sa, \*B = \_b, \*tmp = \_tmp;

3. **char** s[LEN];

4. **void** build\_lcp (){

5. **for**(**int** i = 0, k = 0; i < n; ++i){

6. **if**(B[i] == n - 1)

7. **continue**;

8. **for**(**int** j = SA[B[i] + 1]; i + k < n &&

9. j + k < n && s[i+k] == s[j + k]; k++);

10. LCP[B[i]] = k;

11. **if**( k ) k--;

12. }

13. }

14. **void** build\_sa (){

15. //memset 0 -> \_sa, \_b, \_tmp, top, LCP

16. s[n] = '\0', n ++;

17. **int** na = (n < 256 ? 256 : n);

18. **for** (**int** i = 0; i < n ; i++)

19. top[B[i] = s[i]]++;

20. **for** (**int** i = 1; i < na; i++)

21. top[i] += top[i - 1];

22. **for** (**int** i = 0; i < n ; i++)

23. SA[--top[B[i]]] = i;

24. **for** (**int** ok = 1,j = 0;ok < n && j < n-1;ok <<= 1){

25. **for** (**int** i = 0; i < n; i++){

26. j = SA[i] - ok;

27. **if** (j < 0)

28. j += n;

29. tmp[top[B[j]]++] = j;

30. }

31. SA[tmp[top[0] = 0]] = j = 0;

32. **for** (**int** i = 1; i < n; i++){

33. **if** (B[tmp[i]] != B[tmp[i - 1]] ||

34. B[tmp[i]+ok] != B[tmp[i-1] + ok])

35. top[++j] = i;

36. SA[tmp[i]] = j;

37. }

38. swap(B, SA), swap(SA, tmp);

39. }

40. build\_lcp();

41. n --, s[n] = '\0';

42. }

**7.6. Suffix Automata O( N )**

1. // Construct:

2. // Automaton sa; for(char c : s) sa.extend(c);

3. // 1. Number of distinct substr O( N ):

4. // - Find number of different paths --> DFS on SA

5. // - f[u] = 1 + sum( f[v] for v in s[u].next

6. // 2. Number of occurrences of a substr O( N ):

7. // - Initially, in extend: s[cur].cnt = 1; s[clone].cnt = 0;

8. // - for( auto it = base.rbegin(); it != base.rend(); it ++ ){

9. // int p = st[it->second].link;

10. // cnt[p] += cnt[it->second]; }

11. // 3. Find total length of different substrings O( N ):

12. // - We have f[u] = number of strings starting from node u

13. // - ans[u] = sum(ans[v] + d[v] for v in next[u])

14. // 4. Lexicographically k-th substring O(N)

15. // - Based on number of different substring

16. // 5. Find first occurrence O(N)

17. // - firstpos[cur] = len[cur] - 1, firstpos[clone] = firstpos[q]

18. // 6. Longest common substring of two strings s, t O(N).

19. **struct** state {

20. **int** len, link;

21. **int** fpos;///

22. map<**char**,**int**>next;

23. state( ){

24. len = 0, link = -1, fpos = 0;

25. next.clear();

26. }

27. };

28. **const** **int** MAXLEN = 100002;

29. state st[MAXLEN\*2];

30. **int** sz, last;

31. set<pair<**int**,**int**>> base ;///

32. **int** cnt[MAXLEN\*2];///

33. **void** sa\_init() {

34. sz = last = 0;

35. st[0] = state();

36. cnt[0] = 0;

37. sz++;

38. base.clear();

39. }

40. **void** sa\_extend (**char** c) {

41. **int** cur = sz++;

42. st[cur] = state();

43. st[cur].len = st[last].len + 1;

44. st[cur].fpos = st[cur].len - 1;///

45. cnt[cur] = 1 ; ///

46. base.insert(make\_pair(st[cur].len, cur));///

47. **int** p;

48. **for** (p=last; p!=-1 && !st[p].next.count(c); p=st[p].link)

49. st[p].next[c] = cur;

50. **if** (p == -1)

51. st[cur].link = 0;

52. **else** {

53. **int** q = st[p].next[c];

54. **if** (st[p].len + 1 == st[q].len)

55. st[cur].link = q;

56. **else** {

57. **int** clone = sz++;

58. st[clone] = state();

59. st[clone].len = st[p].len + 1;

60. st[clone].next = st[q].next;

61. st[clone].link = st[q].link;

62. st[clone].fpos = st[q].fpos;///

63. cnt[clone]=0;///

64. base.insert(make\_pair(st[clone].len,clone)); ///

65. **for** (; p!=-1 && st[p].next[c]==q; p=st[p].link)

66. st[p].next[c] = clone;

67. st[q].link = st[cur].link = clone;

68. }

69. }

70. last = cur;

71. }

72. //6. Longest common substring of two strings s, t.

73. **string** lcs (**string** s, **string** t) {

74. sa\_init();

75. **for** (**int** i=0; i<(**int**)s.length(); i++)

76. sa\_extend (s[i]);

77. **int** v = 0, l = 0, best = 0, bestpos = 0;

78. **for** (**int** i=0; i<(**int**)t.length(); i++) {

79. **while** (v && !st[v].next.count(t[i])) {

80. v = st[v].link;

81. l = st[v].len;

82. }

83. **if** (st[v].next.count(t[i])) {

84. v = st[v].next[t[i]];

85. l++;

86. }

87. **if** (l > best) best = l, bestpos = i;

88. }

89. **return** t.substr (bestpos-best+1, best);

90. }

**7.7. Tandems O( NlogN )**

1. **void** output\_tandem (**const** **string** & s, **int** shift,

2. **bool** left, **int** cntr, **int** l, **int** l1, **int** l2){

3. **int** pos;

4. **if** (left) pos = cntr-l1;

5. **else** pos = cntr-l1-l2-l1+1;

6. cout << "[" << shift + pos << ".."; // ini

7. cout << shift + pos+2\*l-1 << "] = "; // fin

8. cout << s.substr (pos, 2\*l) << endl;

9. }

10. **void** output\_tandems (**const** **string** & s, **int** shift,

11. **bool** left, **int** cntr, **int** l, **int** k1, **int** k2){

12. **for** (**int** l1=1; l1<=l; l1++) {

13. **if** (left && l1 == l) break;

14. **if** (l1 <= k1 && l-l1 <= k2)

15. output\_tandem(s,shift,left,cntr, l, l1, l-l1);

16. }

17. }

18. **inline** **int** get\_z (**const** **vector**<**int**> & z, **int** i) {

19. **return** 0<=i && i<(**int**)z.size() ? z[i] : 0;

20. }

21. **void** find\_tandems (**string** s, **int** shift = 0) {

22. **int** n = (**int**) s.length();

23. **if** (n == 1) **return**;

24. **int** nu = n/2, nv = n-nu;

25. **string** u = s.substr (0, nu),

26. v = s.substr (nu);

27. **string** ru = **string** (u.rbegin(), u.rend()),

28. rv = **string** (v.rbegin(), v.rend());

29. find\_tandems (u, shift);

30. find\_tandems (v, shift + nu);

31. **vector**<**int**> z1 = z\_function (ru),

32. z2 = z\_function (v + '#' + u),

33. z3 = z\_function (ru + '#' + rv),

34. z4 = z\_function (v);

35. **for** (**int** cntr=0; cntr<n; cntr++) {

36. **int** l, k1, k2;

37. **if** (cntr < nu) {

38. l = nu - cntr;

39. k1 = get\_z (z1, nu-cntr);

40. k2 = get\_z (z2, nv+1+cntr);

41. }**else** {

42. l = cntr - nu + 1;

43. k1 = get\_z (z3, nu+1 + nv-1-(cntr-nu));

44. k2 = get\_z (z4, (cntr-nu)+1);

45. }

46. **if**(k1 + k2 >= l) // longitud 2\*l

47. output\_tandems(s,shift,cntr<nu,cntr,l,k1,k2);

48. }

49. }

**7.8. Z Algorithm O( N )**

1. **vector**<**int**> z\_function (**const** **string** & s){

2. **int** n = (**int**) s.length();

3. **vector**<**int**> z (n);

4. **for** (**int** i=1, l=0, r=0; i<n; i++) {

5. **if** (i <= r) z[i] = min (r-i+1, z[i-l]);

6. **while** (i+z[i] < n && s[z[i]] == s[i+z[i]])

7. z[i]++;

8. **if** (i+z[i]-1 > r) l = i, r = i+z[i]-1;

9. }

10. **return** z;

11. }