1. Code Template
   1. -optimizacion de cin y cout
   2. -Java Template
2. Number Theory

2.1 -Formulas

-Euler’s formula, Number Catalan, Desarranjo

2.2. -Inverso modular de N!

2.3. -Modular Multiplication of big numbers

2.4. -Rabin-Miller

2.5. -Pollard-Rho

2.6. -Extended GCD(ax+by = d)

2.7. -Inverso Modular

2.8. -Teorema del Resto Chino

2.9. -Modular Equations(ax(n)=b(n))

2.10.-Find a primitive root of a prime number

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

2.12.-Shanks' Algorithm(a^x = b(m))

2.13.-FFT sin complex

2.14.-FFT con complex

2.15.-Phi

2.16.-Brent’s Algorithm (Cycle detection)

2.17.-Floyd’s Cycle-Finding algorithm

2.18.-Matrix Exponentiation

2.19.-Fast Square Testing

3. String

3.1. -Hashing

3.2. -Aho Corasick

3.3. -Manacher

3.4. -Suffix Array

3.5. -Z-Algorithm

3.6. -Lecomposition of lyndon

3.7. -LCS

3.8. -Edit\_Distant

3.9. -KMP

3.10. -Menor Rotación Lexicográfica

4. Graphs

4.1. -Struct edges

4.2. -Bellman Ford

4.3. -LCA

4.4. -Bridges y Punto de Articulacion

4.5. -Tarjan SCC

4.6. -Tarjan BCC

4.7. -vertex cover bipartite

4.8. -Edmons-Karp

4.9. -Dinic

4.10.-StoerWagner

4.11.-Max\_Flow\_Min\_Cost

4.12.-Hungarian

4.13.-Kuhn Bipartite Matching

4.14.-Hopcroft–Karp Bipartite Matching

4.15.-Havy light decomposition

4.16.-Estable Marriage

4.17.-Edmons

4.18.-Centroid descomposition

5. Data Structures

5.1. -Suma de intervalos con BIT

5.2. -AVL

5.3. -Misof Tree->the nth largest element

5.4. -Convex Hull Trick

5.5. -Monotonic Queue

5.6. -Splay Tree

5.7. -RMQ

6. Dynamic Programming

6.1. -Conquer and Divide Optimizations

6.2. -LIS-LDS

6.3. -Towers of Hanoi

7. Geometry

7.1. -Base element

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

7.3. -Intersection

7.4. -Distance.

7.5. -End point

7.6. -Polygon inclusion decision point

7.7. -Area of a polygon

7.8. -Perturbative deformation of the polygon

7.9. -triangulation

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

7.11.-Convexity determination

7.12.-Cutting of a convex polygon

7.13.-Diameter of a convex polygon

7.14.-End point of a convex polygon

7.15.-Convex polygon inclusion decision point

7.16.-Incircle

7.17-Closest Pair Point

7.18.-Intriangle

7.19.-Three Point Circle

7.20.-Circle\_circle\_intersect

8. Solution Ideas

9. Debugging Tips

# Code Templates

## Optimization of cin and cout

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

## Java Template

1. **import** java.util.\*;
2. **import** java.math.\*;
3. **import** java.io.\*;
4. **public** **class** Main {
5. **public** **static** **void** main(String[] args) **throws** IOException {
6. Scanner cin = **new** Scanner(System.*in*);
7. **int** a, b;
8. b = cin.nextInt(); a = cin.nextInt();
9. System.*out*.printf("%d", a+b);
10. cin.close();
11. }
12. }

# 2. Number Theory

## Formulas

*Number Catalan:*

C[n] => FOR(k=0,n-1) C[k] \* C[n-1-k]

C[n] => Comb(2\*n,n) / (n + 1)

C[n] => C[n-1]\*(4\*n-2)/(n+1)

*Euler’s formula:*

A + C = V + 2

*Desarranjo:*

d(1) = 0, d(2) = 1

d(n) = (n-1)\*(d(n-1) + d(n-2))

## Inverso modular de N!

ifact[n+1] = fact[n+1]^(mod-2)

ifact[n] = (ifact[n+a]\*(i+1))%mod;

## Modular Multiplication of big numbers

1. **inline** ll **mulmod**(ll a, ll b, ll m) {
2. ll x = 0, y = a % m;
3. **while** (b > 0) {
4. **if** (b % 2 == 1) x = (x + y) % m;
5. y = (y \* 2) % m;
6. b /= 2;
7. }
8. **return** x;
9. }

## Rabin-Miller

1. //using: mulmod( ), powmod()
2. **bool** **suspect**(ll a, **int** s, ll d, ll n) {
3. ll x = powMod(a, d, n);
4. **if** (x == 1) **return** **true**;
5. **for** (**int** r = 0; r < s; ++r) {
6. **if** (x == n - 1) **return** **true**;
7. x = mulmod(x, x, n);
8. }
9. **return** **false**;
10. }
11. // {2,7,61,0} is for n < 4759123141 (= 2^32)
12. // {2,3,5,7,11,13,17,19,23,0} is for n < 10^16 (at least)
13. **unsigned** test[] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 0 };
14. **bool** **miller\_rabin**(ll n) {
15. **if** (n <= 1 || (n > 2 && n % 2 == 0)) **return** **false**;
16. ll d = n - 1; **int** s = 0;
17. **while** (d % 2 == 0) ++s, d /= 2;
18. **for** (**int** i = 0; test[i] < n && test[i] != 0; i++)
19. **if** (!suspect(test[i], s, d, n))
20. **return** **false**;
21. **return** **true**;
22. }

## Pollard-Rho

1. // Randomized Factorization Algorithm O(sqrt(s(n))) expected
2. // where s(n) is the smallest prime divisor of n
3. // use in conjuntion with miller\_rabin test for primality
4. //using: Rabin-Miller(), mulmod()
5. **#define** func(x)(mulmod(x, x+B, n)+ A )
6. ll **pollard\_rho**(ll n) {
7. **if**( n == 1 ) **return** 1;
8. **if**( miller\_rabin( n ) )
9. **return** n;
10. ll d = n;
11. **while**( d == n ){
12. ll A = 1 + **rand**()%(n-1), B = 1 + **rand**()%(n-1);
13. ll x = 2, y = 2;
14. d = -1;
15. **while**( d == 1 || d == -1 ){
16. x = func(x), y = func(func(y));
17. d = \_\_gcd( x-y, n );
18. }
19. }
20. **return** abs(d);
21. }

## Extended GCD(ax+by = d)

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. }

## Inverso Modular

1. //using: Extended GCD
2. **int** **inverso\_mod**(**int** n ,**int** m){
3. **int** s, t, d;
4. d = extended\_euclid( n, m, s, t );
5. **return** ((s % m)+m)% m;
6. }

## Teorema del Resto Chino

1. //using: Inverso Modular
2. **int** **resto\_chino** (**int** x[], **int** m[], **int** k){
3. **int** i, tmp, MOD = 1, RES = 0;
4. **for** (i=0; i <k ; i++) MOD \*= m[i];
5. **for** (i =0; i <k ; i++){
6. tmp = MOD/m[i];
7. tmp \*= inverso\_mod(tmp, m[i]);
8. RES += (tmp\*x[i]) % MOD;
9. }
10. **return** RES % MOD;
11. }

## Modular Equations (ax(n)=b(n))

1. //using: Extended GCD
2. **int** **modular\_equations**( **int** a, **int** b, **int** n ){
3. **int** s, t, x0;
4. **int** d = extended\_euclid( a, n, s, t );
5. **if**( b % d == 0 ){
6. **int** tmp = (b/d)\*s;
7. x0 = ((tmp%n)+n)%n; // x0 = s\*b
8. **return** x0; // x = x0+(n/d)\*k, 0 < k < d
9. }
10. **return** -1;
11. }

## Find a primitive root of a prime number

1. // O( log^6(p)\*sqrt(p) ), using: powmod()
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. }

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

1. //using: powmod()
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. }

## Shanks' Algorithm(a^x = b(m))

1. // O(sqrt(m)), 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. }

## FFT sin complex

1. ///----FFT-O(n\*log(n))-----
2. **const** **int** MOD = 167772161;
3. // so the algorithm works until n = 2 ^17 = 131072
4. **const** **int** g = 3; // primitive root
5. //const int MOD = 1073872897 = 2^30 + 2^17 + 1, g = 7
6. //another good choice is MOD=167772161=2^27+2^25+1, g = 3
7. //a bigger choice would be MOD=3221225473=2^31+2^30+1, g = 5
8. // but it requires unsigned long long for multiplications
9. **typedef** **long** **long** i64;
10. // n must be a power of two
11. // sign = 1, scale = 1 for DFT
12. // sign = -1, scale=(1/n)(MOD) or (MOD-(MOD-1)/n) for inverse
13. //using: powmod()
14. **void** **ifft**(**int** n, i64 a[],**int** sign, i64 scale) {
15. **int** k;
16. **for** (k = 0; (1 << k) < n; k++);
17. **for** (**int** i = 0; i < n; i++) {
18. **int** q = 0;
19. **for** (**int** j = 0; j < k; j++) {
20. q <<= 1;
21. **if** (i & 1 << j) q++;
22. }
23. **if** (i < q) swap(a[i], a[q]);
24. }
25. i64 x = powmod(g, (MOD - 1) / n, MOD);
26. **for** (**int** q = 2; q <= n; q <<= 1) {
27. **int** q2 = q / 2;
28. i64 wn = powmod(x, n + sign \* n / q, MOD);
29. i64 w = 1;
30. **for** (**int** i = 0; i < q2; i++) {
31. **for** (**int** j = i; j < n; j += q) {
32. **int** v = w \* a[j + q2] % MOD;
33. a[j + q2] = (a[j] - v + MOD) % MOD;
34. a[j] = (a[j] + v) % MOD;
35. }
36. w = i64(w) \* wn % MOD;
37. }
38. }
39. **for** (**int** i = 0; i < n; i++) a[i] = a[i] \* scale % MOD;
40. }

## FFT con complex

1. **#define** PI 2\*acos(0)
2. **typedef** complex<**double**> base;
3. **void** **fft** (vector<base> & a, **bool** invert) {
4. **int** n = (**int**) a.size();
5. **for** (**int** i=1, j=0; i<n; ++i) {
6. **int** bit = n >> 1;
7. **for** (; j>=bit; bit>>=1)
8. j -= bit;
9. j += bit;
10. **if** (i < j) swap (a[i], a[j]);
11. }
12. **for** (**int** len=2; len<=n; len<<=1) {
13. **double** ang = 2\*PI/len \* (invert ? -1 : 1);
14. base wlen (**cos**(ang), **sin**(ang));
15. **for** (**int** i=0; i<n; i+=len) {
16. base w (1);
17. **for** (**int** j=0; j<len/2; ++j) {
18. base u = a[i+j], v = a[i+j+len/2] \* w;
19. a[i+j] = u + v;
20. a[i+j+len/2] = u - v;
21. w \*= wlen;
22. } } }
23. **if** (invert){
24. **for** (**int** i=0; i<n; ++i)
25. a[i] /= n;}
26. }
27. **void** **multiply**(vector<**int**>&a, vector<**int**>&b,vector<**int**> &res){
28. vector<base> fa (a.begin(), a.end()), fb(b.begin(), b.end());
29. size\_t n = 1;
30. **while** (n < max (a.size(), b.size())) n <<= 1;
31. n <<= 1;
32. fa.resize (n), fb.resize (n);
33. fft (fa, **false**), fft (fb, **false**);
34. **for** (size\_t i=0; i<n; ++i)
35. fa[i] \*= fb[i];
36. fft (fa, **true**);
37. res.resize (n);
38. **for** (size\_t i=0; i<n; ++i)
39. res[i] = **int** (fa[i].real() + 0.5);
40. }

## Phi

1. // computes the number of coprimes of p^k, being p prime
2. //int phi(int p,int k){return pow(p,k)-pow(p,k-1);}//phi(p^k)
3. **int** **phi**(**int** p,**int** pk){**return** pk-(pk/p);}//phi(p^k)where pk=p^k
4. // computes the number of coprimes of n
5. // phi(n) = (p\_1-1)\*p\_1^(k\_1-1)\*(p\_2-1)\*p\_2^(k\_2-1)
6. **int** **phi**(**int** n){
7. **int** coprimes = (n != 1); // phi(1) = 0
8. **for** (**int** i = 2; i\*i <= n; i++)
9. **if** (n%i == 0){
10. **int** pk = 1;
11. **while** (n%i == 0)
12. 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. }

## Brent’s Algorithm (Cycle detection)

1. //O(u+l) Fµ(x0) = Fµ+l(x0)
2. par **cycle\_detaction**( ){
3. **int** p = 1, l = 1, t = x0, h = f(x0), u;
4. **while** (t != h){
5. **if** (p == l) t = h, p\*= 2, l = 0;
6. h = f(h), ++l;
7. }
8. u = 0, t = h = x0;
9. **for** (i = l; i != 0; --i) h = f(h);
10. **while** (t != h) t = f(t), h = f(h), ++u;
11. **return** par( u, l );
12. }

## Floyd’s 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. }

## Matrix Exponentiation

1. // O( n^3\*log(n) )
2. **typedef** vector <**int**> vect;
3. **typedef** vector < vect > matrix;
4. matrix **identity** (**int** n) {
5. matrix A(n, vect(n));
6. **for** (**int** i = 0; i <n; i++) A[i][i] = 1;
7. **return** A;
8. }
9. matrix **mul**(**const** matrix &A, **const** matrix &B) {
10. matrix C(A.size(), vect(B[0].size()));
11. **for** (**int** i = 0; i < C.size(); i++)
12. **for** (**int** j = 0; j < C[i].size(); j++)
13. **for** (**int** k = 0; k < A[i].size(); k++)
14. C[i][j] += A[i][k] \* B[k][j];
15. **return** C;
16. }
17. matrix **pow**(**const** matrix &A, **int** e) {
18. **return** ( e == 0 ) ? identity(A.size()) :
19. ( e%2 == 0 )? pow(mul(A,A),e/2) : mul(A,pow(A,e-1));
20. }

## Fast Square Testing

1. **long** **long** M;
2. **void** **init\_is\_square**(){
3. rep(i,0,64) M |= 1ULL << (63-(i \* i)%64);
4. }
5. **inline** **bool** **is\_square**(**long** **long** x) {
6. **if** ((M << x) >= 0) **return** **false**;
7. **int** c =\_\_builtin\_ctz(x);
8. **if** (c & 1) **return** **false**;
9. x >>= c;
10. **if** ((x&7) - 1) **return** **false**;
11. **long** **long** r = **sqrt**(x);
12. **return** r\*r == x;
13. }

# String

## Hashing

1. **unsigned** **long** **long** **calc\_hash**( **int** ptr, **int** in, **int** f ){
2. **return** Dp[f] - Dp[in-1]\*pot33[f-in+1];
3. }
4. **void** **buil\_hash**( ){
5. **for**( **int** i = 1; i <= ta; i ++ )
6. Dp[i] = Dp[i-1]\*33LL + ( A[i] - 'a' );
7. }

## Aho Corasick

1. **const** **int** alph = 26;
2. **struct** tree {
3. **int** parent, slink;
4. **bool** band;
5. **int** hij[30];
6. **tree**( **int** p ){
7. parent = p, slink = 0, band = **false**;
8. fill( hij, hij + 30, -1 );
9. }
10. };
11. vector<tree> trie;
12. **void** **addWord**( string s1 ){
13. **int** root = 0;
14. **for**( **int** i = 0; i < (**int**)s1.length(); i ++ ){
15. **if**( trie[root].hij[s1[i] - 'a'] == -1 ){
16. trie[root].hij[s1[i] - 'a'] = trie.size();
17. trie.push\_back( tree( root ) );
18. }
19. root = trie[root].hij[s1[i] - 'a'];
20. }
21. trie[root].band = **true**;
22. }
23. queue<**int**> Q;
24. **void** **buildSuffixLinks**( ){
25. **int** nod, nextC;
26. Q.push( 0 );
27. Q.push( 0 );
28. **while**( !Q.empty() ){
29. nod = Q.front(), Q.pop();
30. nextC = Q.front(), Q.pop();
31. **for**( **int** i = 0; i <= alph; i ++ )
32. **if**( trie[nod].hij[i] != -1 ){
33. Q.push( trie[nod].hij[i] );
34. Q.push( i );
35. }
36. **if**( nod == 0 || trie[0].hij[nextC] == nod )
37. **continue**;
38. **int**& link = trie[nod].slink;
39. link = trie[trie[nod].parent].slink;
40. **while**( link != 0 && trie[link].hij[nextC] == -1 )
41. link = trie[link].slink;
42. link = trie[link].hij[nextC];
43. **if**( link == -1 )
44. link ++;
45. **if**( trie[link].band )
46. trie[nod].band = **true**;
47. }
48. }
49. **int** **go**( **int** nod, **char** c ){
50. **if**( nod == 0 )
51. **return** trie[0].hij[c - 'a'];
52. **if**( trie[nod].hij[c - 'a'] != -1 )
53. **return** trie[nod].hij[c - 'a'];
54. **int** link = trie[nod].slink;
55. **while**( link != 0 && trie[link].hij[c-'a'] == -1 )
56. link = trie[link].slink;
57. **return** trie[link].hij[c-'a'];
58. }
59. **int** automata[10005][30], N, M;
60. **void** **Aho\_Corasick**( ){
61. string tmp;
62. cin >> N >> M;
63. trie.clear();
64. trie = vector<tree> ( 1, tree(0) );
65. **for**( **int** i = 1; i <= M; i ++ ){
66. cin>>tmp;
67. addWord( tmp );
68. }
69. buildSuffixLinks( );
70. **for**( **int** j = 0; j < (**int**)trie.size(); j ++ )
71. **for**( **int** h = 'a'; h <= 'z'; h ++ )
72. automata[j][h-'a'] = go( j, h );
73. }

## Manacher

1. **int** rad[ 2 \* MAXLEN ], n;
2. **char** s[MAXLEN];
3. **void** **manacher**( ){ /// i%2!=0 par, i%2==0 par
4. **int** i, j, k;
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. }
14. }

## Suffix Array O(nlog(n))

1. **#define** MN 200005
2. **int** N,in[305],prox[MN],sa[MN],k;
3. **int** cant[MN],pos[MN],lcp[MN],may,s1;
4. **char** A[MN];
5. **bool** b1[MN], b2[MN];
6. **void** **LCP**( ){
7. **for**( **int** p = 0, i = 0; i < N; i ++ )
8. **if**( pos[i] != N - 1 ){
9. **for**( **int** j = sa[pos[i]+1]; i + p <= N &&
10. j + p <= N && A[i+p] == A[j+p]; p ++ );
11. lcp[pos[i]] = p;
12. **if**( p ) p --;
13. }
14. }
15. **inline** **void** **upper**( **int** x ){
16. **int** p = pos[x];
17. pos[x] = p + cant[p];
18. cant[p] ++;
19. b2[pos[x]] = **true**;
20. }
21. **void** **Suffix\_Array**( ){
22. fill( in, in + 300, -1 );
23. **for**( **int** i = 0; i < N; i ++ )
24. prox[i] = in[(**int**)A[i]], in[(**int**)A[i]] = i;
25. **for**( **int** i = 'a'; i <= 'z'; i ++ )
26. **for**( **int** j = in[i]; j != -1; j = prox[j] ){
27. sa[k] = j;
28. **if**( j == in[i] ) b1[k] = **true**;
29. k ++;
30. }
31. **int** p;
32. **for**( **int** H = 1; H < N; H \*= 2 ){
33. fill( b2, b2 + N + 1, **false** );
34. **for**( **int** i = 0; i < N; i = k ){
35. **for**( k = i+1; k < N && !b1[k]; k ++ );
36. cant[i] = 0;
37. **for**( **int** j = i; j < k; j ++ )
38. pos[sa[j]] = i;
39. }
40. upper( N - H );
41. **for**( **int** i = 0; i < N; i = k ){
42. **for**( k = i+1; k < N && !b1[k]; k ++ );
43. **for**( **int** j = i; j < k; j ++ )
44. **if**( sa[j] - H >= 0 )
45. upper( sa[j] - H );
46. **for**( **int** j = i; j < k; j ++ )
47. **if**( sa[j]-H >= 0 && b2[pos[sa[j]-H]] ){
48. **for**( p = pos[sa[j] - H] + 1; p < N
49. && !b1[p] && b2[p]; p ++ )
50. b2[p] = **false**;
51. }
52. }
53. **for**( **int** i = 0; i < N; i ++ ){
54. sa[pos[i]] = i;
55. b1[i] = ( b1[i] || b2[i] );
56. }
57. }
58. LCP( );
59. }

## Z-Algorithm

1. **void** **Z\_algorithm**( ){
2. **int** L = 0, R = 0, k;
3. **for** (**int** i = 1; i < n; i++){
4. **if**( i <= R && z[i-L] < R-i+1 )
5. z[i] = z[i-L];
6. **else**{
7. L = i, R = max( R, i );
8. **while**( R < n && s[R-L] == s[R] )
9. R ++;
10. z[i] = R - L;
11. R --;
12. }
13. }
14. }

## Decomposition of lyndon

1. //decomposition of lyndon s= w1w2w3..wk, w1 >= w2 >=...>= wk
2. **void** **lyndon**( string s ){
3. **int** n = (**int**)s.length(), i = 0;
4. **while**( i < n ){
5. **int** j = i+1, k = i;
6. **while**( j < n && s[k] <= s[j] ){
7. **if**( s[k] < s[j] ) k = i;
8. **else** k ++;
9. j ++;
10. }
11. **while**( i <= k ){
12. cout << s.substr( i, j-k )<<endl;
13. i += j-k;
14. }
15. }
16. }

## LCS

1. **int** **lcs** ( ){
2. **for** ( **int** i = 1; i <= t1; i ++ )
3. **for** ( **int** j = 1; j <= t2; j ++ )
4. **if** ( cad1[i] == cad2[j] )
5. Dp[i][j] = Dp[i-1][j-1]+1, P[i][j] = 'D';
6. **else**
7. **if** ( Dp[i-1][j] > Dp[i][j-1] )
8. Dp[i][j] = Dp[i-1][j], P[i][j] = 'I';
9. **else**
10. Dp[i][j] = Dp[i][j-1], P[i][j] = 'S';
11. **return** Dp[t1][t2];
12. }

## Edit Distant

1. **int** **Edit\_Dist**( ){
2. **for**(**int** i = 0; i <= max(la, lb); i ++)
3. C[0][i] = i, C[i][0] = i;
4. **for**(**int** i = 1; i <= la; i ++)
5. **for**(**int** j = 1; j <= lb; j ++)
6. **if**(A[i] == B[j] && C[i - 1][j - 1] != 1 << 30)
7. C[i][j] = C[i - 1][j - 1];
8. **else**{
9. C[i][j] = 1 << 30;
10. C[i][j] = min(C[i][j], C[i - 1][j] + 1);
11. C[i][j] = min(C[i][j], C[i][j - 1] + 1);
12. C[i][j] = min(C[i][j], C[i - 1][j - 1] + 1);
13. }
14. **return** C[la][lb];
15. }

## KMP

1. **void** **pre\_kmp**( ){
2. **for**(**int** j = 0, i = 2; i <= tp; i ++){
3. **while**( j && P[j + 1] != P[i]) j = fall[j];
4. **if**(P[j + 1] == P[i]) j ++;
5. fall[i] = j;
6. }
7. }
8. **void** **kmp**( ){
9. **for**(**int** j = 0, i = 1; i <= tt; i ++){
10. **while**( j && P[j + 1] != T[i]) j = fall[j];
11. **if**(P[j + 1] == T[i]) j ++;
12. **if**(j == tp) **printf**("%d\n", i - tp + 1);
13. }
14. }

## Lex-Rot

1. **int** **lexRot**(string str){
2. **int** n = str.size(), ini=0, fim=1, rot=0;
3. str += str;
4. **while**(fim < n && rot+ini+1 < n)
5. **if** (str[ini+rot] == str[ini+fim]) ini++;
6. **else** **if**(str[ini+rot]<str[ini+fim])fim+=ini+1, ini = 0;
7. **else** rot = max(rot+ini+1, fim), fim = rot+1, ini = 0;
8. **return** rot;
9. }

# Graphs

## Struct edges

1. **int** pos, Index[10005];///index = -1
2. **struct** edges{
3. **int** nod, newn, cap, cost, next;
4. **bool** band;
5. **edges**(**int** a = 0, **int** b = 0, **int** c = 0, **int** d = 0, **int** e = 0){
6. nod = a, newn = b, cap = c, cost = d, next = e;
7. }
8. **int** **nextn** ( **int** a ){
9. **if**( nod == a )
10. **return** newn;
11. **return** nod;
12. }
13. }G[100005];
14. ///nod, newn, cap, cost
15. **void** **insertar**( **int** a, **int** b, **int** c, **int** d = 0 ){
16. G[pos] = edges( a, b, c, d, Index[a] );
17. Index[a] = pos ++;
18. G[pos] = edges( b, a, 0, -d, Index[b] );
19. Index[b] = pos ++;
20. }

## Bellman Ford

1. **double** **Bellman\_Ford**( ){
2. **double** newc;
3. **int** nod, newn;
4. fill( dist + 1, dist + 1 + N, maxint );
5. fill( parent + 1, parent + 1 + N, -1 );
6. dist[In] = D, parent[In] = 1 << 30;
7. **for**( **int** i = 1; i < N; i ++ )
8. **for**( **int** j = 1; j <= M; j ++ ){
9. nod = G[j].nod, newn = G[j].newn;
10. newc = dist[nod] \* G[j].cost;
11. **if**( dist[newn] > newc ){
12. dist[newn] = newc;
13. parent[newn] = nod;
14. }
15. }
16. **if**( parent[Fin] == -1 )
17. **return** 0;
18. **for**( **int** j = 1; j <= M; j ++ ){
19. nod = G[j].nod, newn = G[j].newn;
20. newc = dist[nod] \* G[j].cost;
21. **if**( dist[newn] > newc )
22. **return** 0;//se encontro un ciclo negativo
23. }
24. **return** dist[Fin];
25. }

## LCA

1. **void** **LCA**( ){
2. lv[1] = 1, Q.push( 1 );
3. **int** logg, nod, newn, t;
4. mark[1] = **true**;
5. **while**( !Q.empty() ){
6. nod = Q.front();
7. Q.pop();
8. t = V[nod].size();
9. **for**(**int** i = 0; i < t; i ++){
10. newn = V[nod][i];
11. **if**( mark[newn] ) **continue**;
12. Q.push( newn );
13. lv[newn] = lv[nod] + 1;
14. Dp[newn][0] = nod;
15. logg = log2( lv[newn] );
16. **for**(**int** j = 1; j <= logg; j ++)
17. **if**( Dp[newn][j - 1] )
18. Dp[newn][j]=Dp[Dp[newn][j-1]][j-1];
19. }
20. }
21. }
22. **int** **ancestro**( **int** a, **int** b ){
23. **if**( lv[a] < lv[b] ) swap( a, b );
24. **int** logg = log2( lv[a] );
25. **for**( **int** i = logg; i >= 0; i -- )
26. **if**(lv[a] - ( 1 << i) >= lv[b] && Dp[a][i])
27. a = Dp[a][i];
28. **if**( a == b ) **return** a;
29. **for**(**int** i = logg; i >= 0; i --)
30. **if**(Dp[a][i] != Dp[b][i] && Dp[a][i])
31. a = Dp[a][i], b = Dp[b][i];
32. **return** Dp[a][0];
33. }

## Bridges y Punto de Articulacion

1. **void** **bridges\_PtoArt** ( **int** nod ){
2. **int** newn, num;
3. vector<**int**>::iterator it;
4. Td[nod] = low[nod] = ++ k;
5. **for**(it = V[nod].begin(); it != V[nod].end(); it ++){
6. num = \*it;
7. newn = G[num].nextn( nod );
8. **if**( G[num].band ) **continue**;
9. G[num].band = **true**;
10. **if**( Td[newn] ){
11. low[nod] = min( low[nod], Td[newn] );
12. **continue**;
13. }
14. bridges\_PtoArt( newn );
15. low[nod] = min( low[nod], low[newn] );
16. **if**(Td[nod] < low[newn])
17. puente.push(par( nod, newn ));
18. **if**( (Td[nod] == 1 && Td[newn] > 2 ) ||
19. ( Td[nod] != 1 && Td[nod] <= low[newn] ) )
20. Punto\_art[nod] = **true**;
21. }
22. }

## Tarjan SCC

1. **void** **Tarjan\_SCC**( **int** nod ){
2. **int** newn;
3. vector<**int**>::iterator it;
4. Td[nod] = low[nod] = ++ k;
5. P.push( nod );
6. **for**(it = V[nod].begin(); it != V[nod].end(); it ++){
7. newn = \*it;
8. **if**( Td[newn] ){
9. **if**( !mark[newn] )
10. low[nod] = min( low[nod], Td[newn] );
11. **continue**;
12. }
13. Tarjan\_SCC( newn );
14. low[nod] = min( low[nod], low[newn] );
15. }
16. **if**( low[nod] == Td[nod] ){
17. sol ++;
18. **while**( !mark[nod] )
19. {
20. **printf**("%d ", (**int**)P.top());
21. mark[(**int**)P.top()] = **true**;
22. P.pop();
23. }
24. }
25. }

## Tarjan BCC

1. **void** **BCC**( **int** nod ){
2. Td[nod] = Low[nod] = ++ k;
3. **int** newn, id;
4. vector<**int**>::iterator it;
5. **for**( it = V[nod].begin(); it != V[nod].end(); it ++ ){
6. id = \*it;
7. newn = G[id].nextn( nod );
8. **if**( !mark[id] ){
9. P.push( id );
10. mark[id] = **true**;
11. }
12. **if**( Td[newn] ){
13. Low[nod] = min( Low[nod], Td[newn] );
14. **continue**;
15. }
16. BCC( newn );
17. Low[nod] = min( Low[newn], Low[nod] );
18. **if**( Td[nod] <= Low[newn] ){
19. num ++;
20. **while**( !CB[id] ){
21. CB[P.top()] = num;
22. P.pop();
23. }
24. }
25. }
26. }

## Vertex cover bipartite

1. // Running time: O(VE)
2. **#define** MAXV 5000
3. **int** X, Y, E;
4. **int** matched[MAXV];
5. **bool** mark[MAXV];
6. **bool** T[MAXV];
7. vector<**int**> ady[MAXV];
8. **typedef** pair<**int**, **bool**> par;
9. queue<par> Q;
10. **bool** **augment**( **int** nod ){
11. **if** ( nod == -1 ) **return** **true**;
12. **int** size = ady[nod].size();
13. **for** ( **int** i = 0; i < size; i++ ){
14. **int** newn = ady[nod][i];
15. **if** ( mark[newn] ) **continue** ;
16. mark[newn] = **true**;
17. **if** ( augment( matched[newn] ) ) {
18. matched[nod] = newn;
19. matched[newn] = nod;
20. **return** **true**;
21. }
22. }
23. **return** **false**;
24. }
25. **void** **Vertex\_Cover\_Bipartite**( ){ /// X->Y
26. /\* Find maximum matching \*/
27. **memset**( matched, -1, **sizeof**( matched ) );
28. memset( T, **false**, **sizeof**( T ) );
29. **int** cardinality = 0;
30. **for** ( **int** i = 0; i < X; i++ ){
31. **memset**( mark, 0, **sizeof**( mark ) );
32. **if** ( augment( i ) ) cardinality++;
33. }
34. /\* Find minimum vertex cover \*/
35. **for** ( **int** i = 0; i < X; i++ )
36. **if** ( matched[i] == -1 ){
37. T[i] = **true**;
38. Q.push( par( i, **true** ) );
39. }
40. **int** nod, newn; **bool** band;
41. **while** ( !Q.empty() ){
42. nod = Q.front().first;
43. band = Q.front().second; Q.pop();
44. **int** size = ady[nod].size();
45. **for** ( **int** i = 0; i < size; i++ ){
46. newn = ady[nod][i];
47. **if** ( T[newn] ) **continue** ;
48. **if** ( ( band && newn != matched[nod] ) ||
49. ( !band && newn == matched[nod] )){
50. T[newn] = **true**;
51. Q.push( par( newn, !band ) );
52. }
53. }
54. }
55. **printf**("Minimum Vertex Cover:%d\n", cardinality );
56. //**for** ( **int** i = X; i < X + Y; i++ ) **if**( T[i] )
57. // vline %d %d %d -> V[i-X+1].x, V[i-X+1].a, V[i-X+1].b
58. //**for** ( **int** i = 0; i < X; i++ ) **if** ( !T[i] )
59. //hline %d %d %d\n" -> H[i+1].x, H[i+1].a, H[i+1].b
60. }

## Edmons-Karp

1. **void** **Edmon\_Karp**( ){
2. **int** nod, newn, flow[10005], P[10005];
3. **bool** band;
4. **for**( ; ; ){
5. fill( flow, flow + 2 + 2\*N, 0 );
6. fill( P, P + 2 + 2\*N, -1 );
7. P[0] = 0, flow[0] = 1, band = **false**;
8. **while**( !Q.empty() ) Q.pop();
9. Q.push( 0 );
10. **while**( !band && !Q.empty() ){
11. nod = Q.front(); Q.pop();
12. **for**(**int** i=Index[nod];i != -1; i = G[i].next ){
13. newn = G[i].newn;
14. **if**( P[newn] != -1 || !G[i].cap )
15. **continue**;
16. flow[newn] = min( G[i].cap, flow[nod] );
17. P[newn] = i, Q.push( newn );
18. **if**( newn == fin ){
19. band = **true**;
20. **break**;
21. }
22. }
23. }
24. **if**( !flow[fin] ) **break**;
25. sol += flow[fin];
26. **for**( **int** i = fin; i != 0; i = G[P[i]].nod ){
27. G[P[i]].cap -= flow[fin];
28. G[P[i]^1].cap += flow[fin];
29. }
30. }
31. }

## Dinic O(NM)

1. **int** lv[2005], Id[2005];
2. **bool** **Bfs**( **int** limt ){
3. **while**( !Q.empty() ) Q.pop();
4. fill( lv, lv + 2001, 0);
5. lv[0] = 1;
6. Q.push( 0 );
7. **int** nod, newn;
8. **while**( !Q.empty() ) {
9. nod = Q.front();
10. Q.pop();
11. **for**( **int** i = Index[nod]; i != -1; i = G[i].next ){
12. newn = G[i].newn;
13. **if**( lv[newn]!=0 || G[i].cap<limt )**continue**;
14. lv[newn] = lv[nod] + 1;
15. Q.push( newn );
16. **if**( newn == fin ) **return** **true**;
17. }
18. }
19. **return** **false**;
20. }
21. **bool** **Dfs**( **int** nod, **int** limt ){
22. **if**( nod == fin ) **return** **true**;
23. **int** newn;
24. **for**( ; Id[nod] != -1; Id[nod] = G[Id[nod]].next ){
25. newn = G[Id[nod]].newn;
26. **if**(lv[nod]+1==lv[newn] && G[Id[nod]].cap>=limt
27. && Dfs( newn, limt ) ){
28. G[Id[nod]].cap -= limt;
29. G[Id[nod]^1].cap += limt;
30. **return** **true**;
31. }
32. }
33. **return** **false**;
34. }
35. **int** **Dinic**( ){
36. **int** flow = 0;
37. **for**( **int** limt = 4; limt > 0; ){
38. **if**( !Bfs( limt ) ){
39. limt >>= 1;
40. **continue**;
41. }
42. **for**( **int** i = 0; i <= fin; i ++ )
43. Id[i] = Index[i];
44. **while**( limt > 0 && Dfs( 0, limt ) )
45. flow += limt;
46. }
47. **return** flow;
48. }

## StoerWagner

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. // mezcla s y t
32. **for**(**int** i=0;i<n;++i){
33. G[i][prev] += G[next][i];
34. G[prev][i] += G[next][i];
35. }
36. merged[next] = **true**;
37. }
38. **return** best;
39. }

## Max\_Flow\_Min\_Cost

1. priority\_queue<par, vector<par>, greater<par> >Qp;
2. par **Max\_Flow\_Min\_Cost**( ){
3. **int** FlowF = 0, CostF = 0, F[1005], parent[1005];
4. **int** nod, newn, newc, flow, dist[1005], cost;
5. **for**( ; ; ){
6. fill( F + 1, F + 1 + Fin, 0 );
7. fill( dist + 1, dist + 1 + Fin, 1 << 30 );
8. F[In] = 1 << 30, dist[In] = 0;
9. Qp.push( par( 0, In ) );
10. **while**( !Qp.empty() ){
11. nod = Qp.top().second, cost = Qp.top().first;
12. Qp.pop();
13. flow = F[nod];
14. **for**(**int** i=Index[nod];i != -1; i = G[i].next ){
15. newn = G[i].newn;
16. newc=cost+G[i].cost+Phi[nod]-Phi[newn];
17. **if**( G[i].cap > 0 && dist[newn] > newc ){
18. dist[newn] = newc;
19. F[newn] = min( flow, G[i].cap );
20. parent[newn] = i;
21. Qp.push( par( newc, newn ) );
22. }
23. }
24. }
25. **if**( F[Fin] <= 0 ) **break**;
26. CostF += (( dist[Fin] + Phi[Fin] ) \* F[Fin] );
27. FlowF += F[Fin];
28. **for**( **int** i = 1; i <= N; i ++ )
29. **if**( F[i] ) Phi[i] += dist[i];
30. nod = Fin;
31. **while**( nod != In ){
32. G[parent[nod]].cap -= F[Fin];
33. G[parent[nod]^1].cap += F[Fin];
34. nod = G[parent[nod]].nod;
35. }
36. }
37. **return** par( CostF, FlowF );
38. }

## 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. }

## Kuhn Bipartite Matching O(NM)

1. **bool** **khun**( **int** nodo ){
2. **if**( mark[nodo] )
3. **return** **false**;
4. mark[nodo] = 1;
5. **int** tam = V[nodo].size();
6. **for**( **int** i = 0; i < tam; i++ ){
7. **int** ady = V[nodo][i];
8. **if**( ( match[ady] == -1 || khun(match[ady])) ){
9. match[ady] = nodo;
10. **return** **true**;
11. }
12. }
13. **return** **false**;
14. }
15. **void** **PreMatching**() {
16. **for**( **int** i = 1; i <= N; i++ ){
17. **for**( **int** j = 0; j < (**int**)V[i].size(); j++ ){
18. **int** ady = V[i][j];
19. **if**( match[ady] != -1 )
20. **continue**;
21. match[ady] = i;
22. used[i] = **true**;
23. **break**;
24. }
25. }
26. }
27. /// a -> N+b N|W
28. **int** **Bipartite\_matchin**( ){
29. memset(match,-1,**sizeof**(**int**)\*(N+W+1));
30. PreMatching();
31. **int** sol = 0;
32. **for**( **int** i = 1; i <= N; i++ ){
33. fill(mark,mark+N+1,**false**);
34. **if**( used[i] ){
35. sol++;
36. **continue**;
37. }
38. **if**( khun(i) ) sol++;
39. }
40. **return** sol;
41. }

## Hopcroft–Karp Bipartite Matching O(M\*sqrt(N))

1. **const** **int** MAXV = 1001;
2. **const** **int** MAXV1 = 2\*MAXV;
3. **int** N,M;
4. vector<**int**> ady[MAXV];
5. **int** D[MAXV1],Mx[MAXV], My[MAXV];
6. **bool** **BFS**(){
7. **int** u, v, i, e;
8. queue<**int**> cola;
9. **bool** f = 0;
10. **for** (i = 0; i < N+M; i++) D[i] = 0;
11. **for** (i = 0; i < N; i++)
12. **if** (Mx[i] == -1) cola.push(i);
13. **while** (!cola.empty()){
14. u = cola.front(); cola.pop();
15. **for** (e = ady[u].size()-1; e >= 0; e--) {
16. v = ady[u][e];
17. **if** (D[v + N]) **continue**;
18. D[v + N] = D[u] + 1;
19. **if** (My[v] != -1){
20. D[My[v]] = D[v + N] + 1;
21. cola.push(My[v]);
22. }
23. **else** f = 1;
24. }
25. }
26. **return** f;
27. }
28. **int** **DFS**(**int** u){
29. **for** (**int** v, e = ady[u].size()-1; e >=0; e--){
30. v = ady[u][e];
31. **if** (D[v+N] != D[u]+1) **continue**;
32. D[v+N] = 0;
33. **if** (My[v] == -1 || DFS(My[v])){
34. Mx[u] = v; My[v] = u; **return** 1;
35. }
36. }
37. **return** 0;
38. }
39. **int** **Hopcroft\_Karp**(){
40. **int** i, flow = 0;
41. **for** (i = max(N,M); i >=0; i--) Mx[i] = My[i] = -1;
42. **while** (BFS())
43. **for** (i = 0; i < N; i++)
44. **if** (Mx[i] == -1 && DFS(i))
45. ++flow;
46. **return** flow;
47. }

## Havy light decomposition

1. //Havy light decomposition
2. /// cant- la cantidad de nodos
3. /// pos- la pos. donde aparece
4. /// nn- el nod en el cual aparece
5. /// pd- el link con el padre full superior
6. /// G-Dp
7. /// L-lazy
8. vector<**int**> G[MN], V[MN];
9. vector<**bool**> L[MN];
10. **int** cant[MN], pos[MN], nn[MN], pd[MN];
11. **void** **Dfs**( **int** nod, **int** pad ){
12. **int** t = V[nod].size(), newn;
13. **if**( t == 1 && nod != 1 ){
14. pos[nod] = 0;
15. nn[nod] = nod;
16. cant[nod] = 1;
17. pd[nod] = pad;
18. **return**;
19. }
20. **int** mej = nod;
21. **for**( **int** i = 0; i < t; i ++ ){
22. newn = V[nod][i];
23. **if**( newn == pad )
24. **continue**;
25. Dfs( newn, nod );
26. **if**( cant[mej] < cant[nn[newn]] )
27. mej = nn[newn];
28. }
29. pos[nod] = cant[mej];
30. cant[mej] ++;
31. nn[nod] = mej;
32. pd[mej] = pad;
33. }
34. **typedef** pair<**int**, **int**> par;
35. **typedef** pair<**int**, par> tri;
36. **typedef** vector<tri> vt;
37. **typedef** vector<par> vp;
38. /// me da el recorrido desde a hasta b en vector<tri>
39. /// f posicion s.f in, s.f fin
40. vt **rec**( **int** a, **int** b ){
41. vp A1, B1;
42. A1.clear(), B1.clear();
43. **for**( **int** i = a; i != -1; i = pd[nn[i]] )
44. A1.push\_back( par( nn[i], pos[i] ) );
45. **for**( **int** i = b; i != -1; i = pd[nn[i]] )
46. B1.push\_back( par( nn[i], pos[i] ) );
47. vt C1;
48. C1.clear();
49. reverse( A1.begin(), A1.end() );
50. reverse( B1.begin(), B1.end() );
51. **int** t = 0;
52. **while**(t < (**int**)A1.size() && t <
53. (**int**)B1.size() && A1[t] == B1[t] ) t ++;
54. **if**( t >= (**int**)A1.size() || t >= (**int**)B1.size() ||
55. ( t < (**int**)B1.size() && t < (**int**)A1.size()
56. && A1[t].first != B1[t].first ) ) t --;
57. **if**( (t <(**int**) A1.size() && t < (**int**)B1.size())
58. && A1[t].first == B1[t].first ){
59. C1.push\_back( tri( A1[t].first,
60. par( min( A1[t].second, B1[t].second ),
61. max( A1[t].second, B1[t].second ) ) ) );
62. t ++;
63. }
64. **for**( **int** i = t; i <(**int**) A1.size(); i ++ )
65. C1.push\_back( tri( A1[i].first, par( A1[i].second,
66. cant[A1[i].first] - 1 ) ) );
67. **for**( **int** i = t; i < (**int**)B1.size(); i ++ )
68. C1.push\_back( tri( B1[i].first, par( B1[i].second,
69. cant[B1[i].first] - 1 ) ) );
70. **return** C1;
71. }
72. **void** **havy\_light**( ){
73. Dfs( 1, -1 ); // root
74. **for**( **int** i = 1; i <= N; i ++ )/// rellenar con 4\*cant
75. **if**( cant[i] ){
76. G[i] = vector<**int**> ( cant[i]\*4, 0 );
77. L[i] = vector<**bool**> ( cant[i]\*4, **false** );
78. }
79. }

## Estable Marriage

1. **typedef** vector<**int**> vi;
2. **typedef** vector<vi> vvi;
3. **#define** rep(i,a,b) **for** ( \_\_typeof(a) i=(a); i<(b); ++i)
4. vi **stable\_marriage**(**int** n, **int** \*\*m, **int** \*\*w){
5. queue<**int**> q;
6. vi at(n, 0), eng(n, -1), res(n, -1); vvi inv(n, vi(n));
7. rep(i,0,n) rep(j,0,n) inv[i][w[i][j]] = j;
8. rep(i,0,n) q.push(i);
9. **while** (!q.empty()) {
10. **int** curm = q.front(); q.pop();
11. **for** (**int** &i = at[curm]; i < n; i++) {
12. **int** curw = m[curm][i];
13. **if** (eng[curw] == -1) { }
14. **else** **if** (inv[curw][curm] < inv[curw][eng[curw]])
15. q.push(eng[curw]);
16. **else** **continue**;
17. res[eng[curw] = curm] = curw, ++i; **break**;
18. }
19. }
20. **return** res;
21. }

## Edmons.

1. **struct** MaxMatching {
2. **static** **const** **int** *MaxV* = 1001;
3. **int** V, E;
4. **int** match[*MaxV*];
5. **int** head, tail, Q[*MaxV*];
6. **int** start, finish;
7. **int** newbase;
8. **int** father[*MaxV*], base[*MaxV*];
9. **bool** graph[*MaxV*][*MaxV*];
10. **int** queue[*MaxV*];
11. **bool** inpath[*MaxV*];
12. **bool** inblossom[*MaxV*];
13. **bool** inqueue[*MaxV*];
14. **void** **initialize**(**int** \_\_nodes){
15. V = \_\_nodes;
16. memset(graph,**false**,**sizeof**(graph));
17. }
18. **void** **addEdge**(**int** u, **int** v){
19. graph[u][v] = **true**;
20. graph[v][u] = **true**;
21. }
22. **void** **push**(**int** u){
23. Q[tail++] = u;
24. inqueue[u] = **true**;
25. }
26. **int** **pop**(){ **return** Q[head++]; }
27. **int** **findCommonAncestor**(**int** u, **int** v){
28. **memset**(inpath,0,**sizeof**(inpath));
29. **while**(**true**){
30. u = base[u];
31. inpath[u] = **true**;
32. **if**(u == start) **break**;
33. u = father[match[u]];
34. }
35. **while**(**true**){
36. v = base[v];
37. **if**(inpath[v]) **break**;
38. v = father[match[v]];
39. }
40. **return** v;
41. }
42. **void** **resetTrace**(**int** u){
43. **while**(base[u] != newbase){
44. **int** v = match[u];
45. inblossom[base[u]]= **true**;
46. inblossom[base[v]]= **true**;
47. u = father[v];
48. **if**(base[u] != newbase) father[u] = v;
49. }
50. }
51. **void** **blossomContract**(**int** u, **int** v){
52. newbase = findCommonAncestor(u,v);
53. **memset**(inblossom,**false**,**sizeof**(inblossom));
54. resetTrace(u);
55. resetTrace(v);
56. **if**(base[u] != newbase) father[u]= v;
57. **if**(base[v] != newbase) father[v]= u;
58. **for**(**int** i = 1; i <= V; ++i)
59. **if**(inblossom[base[i]]){
60. base[i] = newbase;
61. **if**(!inqueue[i]) push(i);
62. }
63. }
64. **void** **find\_augmenting\_path**() {
65. **memset**(inqueue,**false**,**sizeof**(inqueue));
66. **memset**(father,0,**sizeof**(father));
67. **for**(**int** i = 1; i <= V; ++i) base[i] = i;
68. head = 0;
69. tail = 0;
70. push(start);
71. finish = 0;
72. **while**(head < tail){
73. **int** u = pop();
74. **for**(**int** v = 1; v <= V; ++v)
75. **if**(graph[u][v]&&(base[u] != base[v])
76. && (match[u] != v)){
77. **if**((v == start) || ((match[v] > 0)
78. && (father[match[v]] > 0))){
79. blossomContract(u,v);
80. **continue**;
81. }
82. **if**(father[v] == 0){
83. father[v] = u;
84. **if**(match[v] > 0)push(match[v]);
85. **else**{
86. finish = v;
87. **return**;
88. }
89. }
90. }
91. }
92. }
93. **void** **augment\_path**(){
94. **int** u = finish;
95. **while**(u > 0){
96. **int** v = father[u];
97. **int** w = match[v];
98. match[v] = u;
99. match[u] = v;
100. u = w;
101. }
102. }
103. **int** **edmonds**(){
104. **memset**(match,0,**sizeof**(match));
105. **for**(**int** i = 1; i <= V; ++i)
106. **if**(!match[i]){
107. start = i;
108. find\_augmenting\_path();
109. **if**(finish > 0) augment\_path();
110. }
111. **int** ans = 0;
112. **for**(**int** i = 1; i <= V; ++i)
113. **if**(match[i]) ++ans;
114. **return** ans / 2;
115. }
116. } edmond;

## Centroid descomposition

1. **#define** MAXN 100005
2. **bool** mark[MAXN];
3. **int** cant[MAXN], timer;
4. **void** **Dfs**( **int** nod, **int** pad ){
5. cant[nod] = 1;
6. **for**( **auto** i:V[nod] )
7. **if**( i.first != pad && !mark[i.first] ){
8. Dfs( i.first, nod );
9. cant[nod] += cant[i.first];
10. }
11. }
12. **int** **centroid**( **int** nod, **int** pad, **int** nn ){
13. **for**( **auto** i:V[nod] )
14. **if**(i.first!=pad && !mark[i.first] && cant[i.first]>nn/2)
15. **return** centroid( i.first, nod, nn );
16. **return** nod;
17. }
18. **long** **long** **solve**( **int** nod, **int** pad ){
19. Dfs( nod, pad );
20. **int** centr = centroid( nod, pad, cant[nod] );
21. **long** **long** sol = 0;
22. mark[centr] = **true**;
23. timer ++;
24. **for**( **auto** i : V[centr] )
25. **if**( i.first != pad && !mark[i.first] ){
26. sol += query( i.first, centr, i.second );
27. updater( i.first, centr, i.second );
28. }
29. **for**( **auto** i : V[centr] )
30. **if**( i.first != pad && !mark[i.first] )
31. sol += solve( i.first, centr );
32. **return** sol;
33. }

# Data Structures

## Suma de intervalos con BIT

1. **void** **updater**( **int** x, **int** v ){
2. **int** tmp = x-1;
3. **for**( ; x <= N; x += (x&-x) ){
4. Dp[1][x] += v, Dp[2][x] += v\*tmp;
5. }
6. }
7. **int** **sum**( **int** p, **int** x ){
8. **int** s = 0;
9. **for**( ; x >= 1; x -= (x&-x) )
10. s += Dp[p][x];
11. **return** s;
12. }
13. **int** **sumsum**( **int** a ){
14. **return** sum( 1, a )\*a - sum( 2, a );
15. }
16. **void** **updater\_interv**( **int** a, **int** b, **int** v ){
17. updater( a, v ), updater( b+1, -v );
18. }

## AVL

1. **template** <**class** **T**>
2. **struct** avl\_tree {
3. **struct** node {
4. **T** key;
5. **int** size, height;
6. node \*child[2];
7. **node**(**const** **T** &key) : key(key), size(1), height(1) {
8. child[0] = child[1] = 0; }
9. } \*root;
10. **typedef** node \*pointer;
11. **avl\_tree**() { root = NULL; }
12. pointer **find**(**const** **T** &key) { **return** find(root, key); }
13. node \***find**(node \*t, **const** **T** &key) {
14. **if** (t == NULL) **return** NULL;
15. **if** (key == t->key) **return** t;
16. **else** **if** (key < t->key) **return** find(t->child[0], key);
17. **else** **return** find(t->child[1], key);
18. }
19. **void** **insert**(**const** **T** &key){root=insert(root,**new** node(key));}
20. node \***insert**(node \*t, node \*x) {
21. **if** (t == NULL) **return** x;
22. **if** (x->key < t->key) t->child[0] = insert(t->child[0], x);
23. **else** t->child[1] = insert(t->child[1], x);
24. t->size += 1;
25. **return** balance(t);
26. }
27. **void** **erase**(**const** **T** &key) { root = erase(root, key); }
28. node \***erase**(node \*t, **const** **T** &x) {
29. **if** (t == NULL) **return** NULL;
30. **if** (x == t->key) {
31. **return** move\_down(t->child[0], t->child[1]);
32. } **else** {
33. **if** (x < t->key) t->child[0] = erase(t->child[0], x);
34. **else** t->child[1] = erase(t->child[1], x);
35. t->size -= 1;
36. **return** balance(t);
37. }
38. }
39. node \***move\_down**(node \*t, node \*rhs) {
40. **if** (t == NULL) **return** rhs;
41. t->child[1] = move\_down(t->child[1], rhs);
42. **return** balance(t);
43. }
44. **#define** sz(t) (t ? t->size : 0)
45. **#define** ht(t) (t ? t->height : 0)
46. node \***rotate**(node \*t, **int** l, **int** r) {
47. node \*s = t->child[r];
48. t->child[r] = s->child[l];
49. s->child[l] = balance(t);
50. **if** (t) t->size = sz(t->child[0]) + sz(t->child[1]) + 1;
51. **if** (s) s->size = sz(s->child[0]) + sz(s->child[1]) + 1;
52. **return** balance(s);
53. }
54. node \***balance**(node \*t) {
55. **for** (**int** i = 0; i < 2; ++i) {
56. **if** (ht(t->child[!i]) - ht(t->child[i]) < -1) {
57. **if**(ht(t->child[i]->child[!i])-ht(t->child[i]->child[i]) >0)
58. t->child[i] = rotate(t->child[i], i, !i);
59. **return** rotate(t, !i, i);
60. }
61. }
62. **if**(t)t->height = max(ht(t->child[0]), ht(t->child[1]))+1;
63. **if** (t) t->size = sz(t->child[0]) + sz(t->child[1]) + 1;
64. **return** t;
65. }
66. pointer **rank**(**int** k) **const** { **return** rank(root, k); }
67. pointer **rank**(node \*t, **int** k) **const** {
68. **if** (!t) **return** NULL;
69. **int** m = sz(t->child[0]);
70. **if** (k < m) **return** rank(t->child[0], k);
71. **if** (k == m) **return** t;
72. **if** (k > m) **return** rank(t->child[1], k - m - 1);
73. }
74. **void** **clear**( node \*x ){
75. **if**( !x ) **return**;
76. **if**( x->child[0] )
77. clear( x->child[0] );
78. **if**( x->child[1] )
79. clear( x->child[1] );
80. **delete** x;
81. }
82. **int** **solve**( **const** **T** v ){
83. node \*p = root;
84. **int** sol = 0;
85. **while**( p ){
86. **if** (v < p->key)
87. p = p->child[0];
88. **else**
89. sol+=(( !p->child[0] )?0:p->child[0]->size)+1, p=p->child[1];
90. }
91. **return** sol;
92. }
93. };

## Misof Tree->the nth largest element

1. // Misof Tree. A simple tree data structure for inserting,
2. erasing, and querying the nth largest element.
3. **#define** BITS 15
4. **struct** misof\_tree {
5. **int** cnt[BITS][1<<BITS];
6. **misof\_tree**() { memset(cnt, 0, **sizeof**(cnt)); }
7. **void** **insert**(**int** x) {
8. **for** (**int** i = 0; i < BITS; cnt[i++][x]++, x >>= 1); }
9. **void** **erase**(**int** x) {
10. **for** (**int** i = 0; i < BITS; cnt[i++][x]--, x >>= 1); }
11. **int** **nth**(**int** n) {
12. **int** res = 0;
13. **for** (**int** i = BITS-1; i >= 0; i--)
14. **if** (cnt[i][res <<= 1] <= n)n-=cnt[i][res], res|=1;
15. **return** res;
16. }
17. };

## Convex Hull Trick

1. **struct** convex\_hull\_trick {
2. vector< pair<**double**,**double**> > h;
3. **double** **intersect**(**int** i) {
4. **return** (h[i+1].second-h[i].second)/
5. (h[i].first-h[i+1].first);
6. }
7. **void** **add**(**double** m, **double** b) {
8. h.push\_back(make\_pair(m,b));
9. **while** (h.size() >= 3) {
10. **int** n = h.size();
11. **if** (intersect(n-3) < intersect(n-2)) **break**;
12. swap(h[n-2], h[n-1]);
13. h.pop\_back();
14. } }
15. **double** **get\_min**(**double** x) {
16. **int** lo = 0, hi = h.size() - 2, res = -1;
17. **while** (lo <= hi) {
18. **int** mid = lo + (hi - lo) / 2;
19. **if** (intersect(mid) <= x) res = mid, lo = mid + 1;
20. **else** hi = mid - 1;
21. }
22. **return** h[res+1].first\*x + h[res+1].second;
23. }};

## Monotonic Queue

1. **typedef** **long** **long** i64;
2. **struct** monotonic\_queue {
3. deque< pair<**int**, i64> > D;
4. **void** **add**( **int** p, i64 v ){
5. **while**( !D.empty() && D.front().second <= v )
6. D.pop\_front();
7. D.push\_front({p, v});
8. }
9. **void** **borrar**( **int** x ){
10. **while**( !D.empty() && D.back().first <= x )
11. D.pop\_back();
12. }
13. i64 **maximo**( ){ **return** D.back().second; }
14. };

## Splay Tree

1. **struct** splay\_tree{
2. **const** **int** inf = 1e9;
3. **struct** nodo {
4. **int** size, cant[30];
5. nodo \*l, \*r, \*p;
6. **bool** inv;
7. **int** laz, let;
8. **nodo**(nodo \*f=0, nodo \*i = 0, nodo \*d = 0){
9. l=i, p=f,r=d,size=1,let=0,laz = -1, inv = **false**;
10. **for**(**int** i=0; i<30; i++) cant[i]=0;
11. }
12. } \*root;
13. **splay\_tree**(){ root = NULL; }
14. **inline** **void** **zig**(nodo \*x) {
15. nodo \*y = x->p, \*z = y->p;
16. y->l = x->r;
17. **if**( x->r )
18. x->r->p = y;
19. x->p = z;
20. **if**( z )
21. **if** (z->l == y)z->l = x; **else** z->r = x;
22. y->p = x, x->r = y;
23. updata(y);
24. }
25. **inline** **void** **zag**(nodo \*x) {
26. nodo \*y = x->p, \*z = y->p;
27. y->r = x->l;
28. **if**( x->l )
29. x->l->p = y;
30. x->p = z;
31. **if**( z )
32. **if** (z->l == y)z->l = x; **else** z->r = x;
33. y->p = x, x->l = y;
34. updata(y);
35. }
36. **inline** **void** **splay**(nodo \*x) {
37. **for** (; x->p ;) {
38. nodo \*y = x->p, \*z = y->p;
39. **if** (!z) {
40. **if** (y->l == x) zig(x); **else** zag(x);
41. } **else** {
42. **if** (z->l == y){
43. **if** (y->l == x)zig(y), zig(x);
44. **else** zag(x), zig(x);
45. }
46. **else** **if** (y->r == x) zag(y), zag(x);
47. **else** zig(x), zag(x);
48. }
49. }
50. root = x, updata(root);
51. }
52. **void** **find**(**int** x) {
53. **if**(!root)**return**;
54. nodo \*p = root;
55. **for**( ; ; ) {
56. lazy( p );
57. **int** izq = (p->l)?p->l->size:0;
58. **if** (x == izq + 1) **break**;
59. **if** (x > izq + 1){
60. x -= izq + 1;
61. **if** ( p->r ) p = p->r; **else** **break**;
62. }
63. **else**
64. **if** ( p->l ) p = p->l; **else** **break**;
65. }
66. splay(p);
67. }
68. **inline** **void** **insertpos**( **int** a, **int** b ){
69. nodo \*nn = **new** nodo( 0, 0, 0 );
70. nn->let = b;
71. find( a );
72. **if**( !root ){ root = nn, updata(root); **return**; }
73. nodo \*p = root;
74. root = root->r;
75. **if**( root ) root->p = 0;
76. p->r = nn, nn->p = p;
77. find( -inf );
78. nn->r = root;
79. **if**( root )
80. root->p = nn;
81. root = p;
82. updata(nn), updata(root);
83. **int** ui = 0;
84. }
85. **inline** **void** **insert**(**int** a) {
86. nodo \*p = root, \*f=0;
87. **while**(p){ f=p; p = p->r; }
88. p = **new** nodo(f, 0, 0);
89. p->let = a;
90. **if**( f )
91. f->r = p;
92. splay(p);
93. }
94. **inline** splay\_tree **split**(**int** x){
95. **if**(!root)**return** splay\_tree();
96. splay\_tree L = splay\_tree();
97. find(x);
98. **if**( root->l )
99. root->l->p=0;
100. L.root = root->l, root->l=0;
101. updata(root);
102. **return** L;
103. }
104. **inline** **void** **join**(splay\_tree L){
105. **if**( !L.root ) **return**;
106. **if**(!root) root = L.root;
107. **else**{
108. find(-inf);
109. root->l = L.root, root->l->p = root;
110. updata(root);
111. }
112. L.root = NULL;
113. }
114. **void** **print**(nodo \*r){
115. **if**(r == NULL)**return**;
116. lazy(r);
117. print(r->l);
118. **printf**("%c ", r->let);
119. print(r->r);
120. }
121. **void** **erase**(**int** x) {
122. find(x);
123. **if**(!root)**return**;
124. **if** (!root->l) {
125. nodo \*tmp = root;
126. root = root->r;
127. **if**(root)
128. root->p = 0;
129. **delete** tmp;
130. } **else** {
131. nodo \*t = root->r, \*tmp = root;
132. root = root->l;
133. **if**(root)root->p = 0;
134. find(x);
135. **if**(root)root->r = t;
136. **if**( t ) t->p = root;
137. updata(root);
138. **delete** tmp;
139. }
140. }
141. **void** **clear**( nodo\*x ){
142. **if**( x ) **return**;
143. clear( x->l );
144. clear( x->r );
145. **delete** x;
146. }
147. **inline** **void** **updata**(nodo \*x) {
148. x->size=((x->l)?x->l->size:0)+
149. ((x->r)?x->r->size:0)+1;
150. **for**(**int** i = 0; i < 30; i ++)
151. x->cant[i] = ((x->l)?x->l->cant[i]:0) +
152. ((x->r)?x->r->cant[i]:0) +
153. (x->let == i );
154. }
155. **inline** **void** **lazy**(nodo \*p){
156. **if**(!p)**return**;
157. **if**(p->inv){
158. swap(p->r, p->l);
159. **if**( p->r ) p->r->inv = !p->r->inv;
160. **if**( p->l ) p->l->inv = !p->l->inv;
161. p->inv=0;
162. }
163. **if**(p->laz!=-1){
164. updlazy(p->l, p->laz);
165. updlazy(p->r, p->laz);
166. p->laz = -1;
167. }
168. }
169. **inline** **void** **updlazy**(nodo \*p, **int** laz){
170. **if**( !p ) **return**;
171. p->laz = laz;
172. **for**(**int** i=0; i<30; i++)
173. **if**(i==p->laz) p->cant[i] = p->size;
174. **else** p->cant[i] = 0;
175. p->let = laz;
176. }
177. **void** **solve**(**char** opt, **int** a, **int** b, **int** c = 0 ){
178. splay\_tree t1 = split( a );
179. splay\_tree t = split( b - a + 2 );
180. **if**(opt=='S') t.updlazy(t.root, c);
181. **else** **if**( opt == 'R' )t.root->inv=( !t.root->inv );
182. **else** **printf**("%d\n", t.root->cant[c]);
183. join(t);
184. join(t1);
185. }
186. }ST;

## RMQ

1. **void** **build\_rmq**( ){
2. **for**(**int** i = 0; i < N; i ++) M[i][0] = i;
3. **for**(**int** j = 1; ( 1 << j ) < N; j ++ )
4. **for**(**int** i = 0; i+(1<<(j-1)) < N; i ++ )
5. **if**(arr[M[i][j-1]] <= arr[ M[i+(1<<(j-1))][j-1]])
6. M[i][j] = M[i][j-1];
7. **else** M[i][j] = M[i+(1<<(j-1))][j-1];
8. }
9. **int** **query\_rmq**( **int** x, **int** y ){
10. **int** lg = **log2**( y - x + 1 );
11. **if**( arr[ M[x][lg] ] <= arr[M[y-(1<<lg)+1][lg] ])
12. **return** M[x][lg];
13. **else** **return** M[y-(1<<lg)+1][lg];
14. }

# Dynamic Programming

## Conquer and Divide 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. }

## LIS-LDS

1. **void** **write** ( **int** ID ){
2. **if**( !ID ) **return**;
3. write ( Last[ID] );
4. printf ("%d ", List[ID]);
5. }
6. **void** **LIS\_LDS**( ){
7. **for** ( **int** i = 1; i <= N; i ++ ){
8. **if** ( Sol[m] <= List[i] ){
9. Sol[++ m] = List[i];
10. Id[m] = i;
11. Last[i] = Id[m - 1];
12. }
13. **else**{
14. up = upper\_bound(Sol + 1, Sol + m+1, List[i])-Sol;
15. Sol[up] = List[i];
16. Id[up] = i;
17. Last[i] = Id[up - 1];
18. }
19. }
20. **printf** ("%d\n", m);
21. write ( Id[m] );
22. }

## Towers of Hanoi

1. **void** **move**( **int** n, **char** from, **char** to, **char** aux ) {
2. **if** ( n == 1 )
3. **printf**( "Move disk from %c to %c\n", from, to );
4. **else** {
5. move( n - 1, from, aux, to );
6. **printf**( "Move disk from %c to %c\n", from, to );
7. move( n - 1, aux, to, from );
8. }

## }

# Geometry

## Base element

1. **const** **double** EPS = 1e-8;
2. **const** **double** inf = 1e12;
3. **typedef** complex<**double**> P;
4. **typedef** vector<P> polygon;
5. **namespace** std {
6. **bool** **operator <** (**const** P& a, **const** P& b) {
7. **return** real(a)!=real(b)?real(a)<real(b):imag(a)<imag(b);
8. }
9. }
10. **double** **cross**(**const** P& a, **const** P& b) {
11. **return** imag(conj(a)\*b);
12. }
13. **double** **dot**(**const** P& a, **const** P& b) {
14. **return** real(conj(a)\*b);
15. }
16. **struct** L : **public** vector<P> {
17. **L**(**const** P &a, **const** P &b) {
18. push\_back(a); push\_back(b);
19. }
20. };
21. **struct** C {
22. P p; **double** r;
23. **C**(**const** P &p, **double** r) : p(p), r(r) { }
24. };
25. P **crosspoint**(**const** L &l, **const** L &m) {
26. **double** A = cross(l[1] - l[0], m[1] - m[0]);
27. **double** B = cross(l[1] - l[0], l[1] - m[0]);
28. **if** (**abs**(A) < EPS && **abs**(B) < EPS) **return** m[0]; // same line
29. **if** (**abs**(A) < EPS) assert(**false**); // NOT SATISFIED!!!
30. **return** m[0] + B / A \* (m[1] - m[0]);
31. }

## The traveling direction of the point

1. **int** **ccw**(P a, P b, P c) {
2. b -= a; c -= a;
3. **if** (cross(b, c) > 0) **return** +1; // counter clockwise
4. **if** (cross(b, c) < 0) **return** -1; // clockwise
5. **if** (dot(b, c) < 0) **return** +2; // c--a--b on line
6. **if** (norm(b) < norm(c)) **return** -2; // a--b--c on line
7. **return** 0;
8. }

## Intersection

1. **bool** **intersectLL**(**const** L &l, **const** L &m) {
2. **return** abs(cross(l[1]-l[0],m[1]-m[0]))>EPS ||//non-parallel
3. abs(cross(l[1]-l[0], m[0]-l[0]))<EPS;// same line
4. }
5. **bool** **intersectLS**(**const** L &l, **const** L &s) {
6. **return** cross(l[1]-l[0], s[0]-l[0])\*// s[0] is left of l
7. cross(l[1]-l[0], s[1]-l[0])<EPS;//s[1] is right of l
8. }
9. **bool** **intersectLP**(**const** L &l, **const** P &p) {
10. **return** **abs**(cross(l[1]-p, l[0]-p)) < EPS;
11. }
12. **bool** **intersectSS**(**const** L &s, **const** L &t) {
13. **return** ccw(s[0],s[1],t[0])\*ccw(s[0],s[1],t[1]) <= 0 &&
14. ccw(t[0],t[1],s[0])\*ccw(t[0],t[1],s[1]) <= 0;
15. }
16. **bool** **intersectSP**(**const** L &s, **const** P &p) {
17. **return** abs(s[0]-p)+abs(s[1]-p)-abs(s[1]-s[0]) < EPS;
18. // triangle inequality
19. }

## Distance

1. P **projection**(**const** L &l, **const** P &p) {
2. **double** t = dot(p-l[0], l[0]-l[1]) / norm(l[0]-l[1]);
3. **return** l[0] + t\*(l[0]-l[1]);
4. }
5. P **reflection**(**const** L &l, **const** P &p) {
6. **return** p + P(2,0)\*(projection(l, p) - p);
7. }
8. **double** **distanceLP**(**const** L &l, **const** P &p) {
9. **return** abs(p - projection(l, p));
10. }
11. **double** **distanceLL**(**const** L &l, **const** L &m) {
12. **return** intersectLL(l, m) ? 0 : distanceLP(l, m[0]);
13. }
14. **double** **distanceLS**(**const** L &l, **const** L &s) {
15. **if** (intersectLS(l, s)) **return** 0;
16. **return** min(distanceLP(l, s[0]), distanceLP(l, s[1]));
17. }
18. **double** **distanceSP**(**const** L &s, **const** P &p) {
19. **const** P r = projection(s, p);
20. **if** (intersectSP(s, r)) **return** abs(r - p);
21. **return** min(abs(s[0] - p), abs(s[1] - p));
22. }
23. **double** **distanceSS**(**const** L &s, **const** L &t) {
24. **if** (intersectSS(s, t)) **return** 0;
25. **return** min(min(distanceSP(s, t[0]), distanceSP(s, t[1])),
26. min(distanceSP(t, s[0]), distanceSP(t, s[1])));
27. }

## End point

1. **#define** d(G, k) (dot(G[k], l[1] - l[0]))
2. P **extreme**(**const** polygon &G, **const** L &l) {
3. **int** k = 0;
4. **for** (**int** i = 1; i < (**int**)G.size(); ++i)
5. **if** (d(G, i) > d(G, k)) k = i;
6. **return** G[k];
7. }

## Polygon inclusion decision point

1. **#define** curr(G, i) G[i]
2. **#define** next(G, i) G[(i+1)%G.size()]
3. **enum** { *OUT*, *ON*, *IN* };
4. **int** **contains**(**const** polygon &G, **const** P& p) {
5. **bool** in = **false**;
6. **for** (**int** i = 0; i < (**int**)G.size(); ++i) {
7. P a = curr(G,i) - p, b = next(G,i) - p;
8. **if** (imag(a) > imag(b)) swap(a, b);
9. **if** (imag(a) <= 0 && 0 < imag(b))
10. **if** (cross(a, b) < 0) in = !in;
11. **if** (cross(a, b) == 0 && dot(a, b) <= 0) **return** *ON*;
12. }
13. **return** in ? *IN* : *OUT*;
14. }

## 7.7. Area of a polygon

1. **double** **area2**(**const** polygon& G) {
2. **double** A = 0;
3. **for** (**int** i = 0; i < (**int**)G.size(); ++i)
4. A += cross(curr(G, i), next(G, i));
5. **return** A;
6. }

## Perturbative deformation of a polygon

1. **#define** prev(G,i) G[(i-1+G.size())%G.size()]
2. polygon **shrink\_polygon**(**const** polygon &G, **double** len) {
3. polygon res;
4. **for** (**int** i = 0; i < (**int**)G.size(); ++i) {
5. P a = prev(G,i), b = curr(G,i), c = next(G,i);
6. P u = (b - a) / abs(b - a);
7. **double** th = arg((c - b)/ u) \* 0.5;
8. res.push\_back(b+u \* P(-**sin**(th), **cos**(th))\*len / **cos**(th) );
9. }
10. **return** res;
11. }

## triangulation

1. polygon **make\_triangle**(**const** P& a, **const** P& b, **const** P& c) {
2. polygon ret(3);
3. ret[0] = a; ret[1] = b; ret[2] = c;
4. **return** ret;
5. }
6. **bool** **triangle\_contains**(**const** polygon& tri, **const** P& p) {
7. **return** ccw(tri[0], tri[1], p) >= 0 &&
8. ccw(tri[1], tri[2], p) >= 0 &&
9. ccw(tri[2], tri[0], p) >= 0;
10. }
11. **bool** **ear\_Q**(**int** i, **int** j, **int** k, **const** polygon& G) {
12. polygon tri = make\_triangle(G[i], G[j], G[k]);
13. **if** (ccw(tri[0], tri[1], tri[2]) <= 0) **return** **false**;
14. **for** (**int** m = 0; m < (**int**)G.size(); ++m)
15. **if** (m != i && m != j && m != k)
16. **if** (triangle\_contains(tri, G[m]))
17. **return** **false**;
18. **return** **true**;
19. }
20. **void** **triangulate**(**const** polygon& G, vector<polygon>& t) {
21. **const** **int** n = G.size();
22. vector<**int**> l, r;
23. **for** (**int** i = 0; i < n; ++i) {
24. l.push\_back( (i-1+n) % n );
25. r.push\_back( (i+1+n) % n );
26. }
27. **int** i = n-1;
28. **while** ((**int**)t.size() < n-2) {
29. i = r[i];
30. **if** (ear\_Q(l[i], i, r[i], G)) {
31. t.push\_back(make\_triangle(G[l[i]], G[i], G[r[i]]));
32. l[ r[i] ] = l[i];
33. r[ l[i] ] = r[i];
34. }
35. }
36. }

## Convex\_hull

1. vector<P> **convex\_hull**(vector<P> ps) {
2. **int** n = ps.size(), k = 0;
3. sort(ps.begin(), ps.end());
4. vector<P> ch(2\*n);
5. **for** (**int** i = 0; i < n; ch[k++] = ps[i++])
6. **while** (k >= 2 && ccw(ch[k-2], ch[k-1], ps[i]) <= 0) --k;
7. **for** (**int** i = n-2, t = k+1; i >= 0; ch[k++] = ps[i--])
8. **while** (k >= t && ccw(ch[k-2], ch[k-1], ps[i]) <= 0) --k;
9. ch.resize(k-1);
10. **return** ch;
11. }

## Convexity determination

1. **bool** **isconvex**(**const** polygon &G) {
2. **for** (**int** i = 0; i < (**int**)G.size(); ++i)
3. **if**(ccw(prev(G,i),curr(G,i),next(G, i)) > 0)**return** **false**;
4. **return** **true**;
5. }

## Cutting of a convex polygon

1. polygon **convex\_cut**(**const** polygon& G, **const** L& l) {
2. polygon Q;
3. **for** (**int** i = 0; i < (**int**)G.size(); ++i) {
4. P A = curr(G, i), B = next(G, i);
5. **if** (ccw(l[0], l[1], A) != -1) Q.push\_back(A);
6. **if** (ccw(l[0], l[1], A)\*ccw(l[0], l[1], B) < 0)
7. Q.push\_back(crosspoint(L(A, B), l));
8. }
9. **return** Q;
10. }

## Diameter of a convex polygon

1. **#define** diff(G, i) (next(G, i) - curr(G, i))
2. **double** **convex\_diameter**(**const** polygon &pt) {
3. **const** **int** n = pt.size();
4. **int** is = 0, js = 0;
5. **for** (**int** i = 1; i < n; ++i) {
6. **if** (imag(pt[i]) > imag(pt[is])) is = i;
7. **if** (imag(pt[i]) < imag(pt[js])) js = i;
8. }
9. **double** maxd = norm(pt[is]-pt[js]);
10. **int** i, maxi, j, maxj;
11. i = maxi = is;
12. j = maxj = js;
13. **do** {
14. **if** (cross(diff(pt,i), diff(pt,j)) >= 0) j = (j+1) % n;
15. **else** i = (i+1) % n;
16. **if** (norm(pt[i]-pt[j]) > maxd) {
17. maxd = norm(pt[i]-pt[j]);
18. maxi = i; maxj = j;
19. }
20. } **while** (i != is || j != js);
21. **return** maxd; /\* farthest pair is (maxi, maxj). \*/
22. }

## End point of a convex polygon

1. P **convex\_extreme**(**const** polygon &G, **const** L &l) {
2. **const** **int** n = G.size();
3. **int** a = 0, b = n;
4. **if** (d(G, 0) >= d(G, n-1) && d(G, 0) >= d(G, 1)) **return** G[0];
5. **while** (a < b) {
6. **int** c = (a + b) / 2;
7. **if** (d(G,c)>= d(G,c-1) && d(G,c)>=d(G, c+1))**return** G[c];
8. **if** (d(G, a+1) > d(G, a)) {
9. **if** (d(G, c+1) <= d(G, c) || d(G, a) > d(G, c)) b = c;
10. **else** a = c;
11. } **else** {
12. **if** (d(G, c+1) > d(G, c) || d(G, a) >= d(G, c)) a = c;
13. **else** b = c;
14. }
15. }
16. **return** G[0];
17. }

## Convex polygon inclusion decision point

1. **enum** { *OUT*, *ON*, *IN* };
2. **int** **convex\_contains**(**const** polygon &G, **const** P &p) {
3. **const** **int** n = G.size();
4. P g = (G[0] + G[n/3] + G[2\*n/3]) / 3.0; // inner-point
5. **int** a = 0, b = n;
6. **while** (a+1 < b) { // invariant: c is in fan g-P[a]-P[b]
7. **int** c = (a + b) / 2;
8. **if** (cross(G[a]-g, G[c]-g) > 0) { // angle < 180 deg
9. **if** (cross(G[a]-g,p-g)>0 && cross(G[c]-g,p-g) < 0) b=c;
10. **else** a = c;
11. } **else** {
12. **if** (cross(G[a]-g,p-g)<0 && cross(G[c]-g,p-g) > 0) a=c;
13. **else** b = c;
14. }
15. }
16. b %= n;
17. **if** (cross(G[a] - p, G[b] - p) < 0) **return** *OUT*;
18. **if** (cross(G[a] - p, G[b] - p) > 0) **return** *IN*;
19. **return** *ON*;
20. }

## Incircle

1. **bool** **incircle**(P a, P b, P c, P p) {
2. a -= p; b -= p; c -= p;
3. **return** norm(a) \* cross(b, c)
4. + norm(b) \* cross(c, a)
5. + norm(c) \* cross(a, b) >= 0;
6. // < : inside, = cocircular, > outside
7. }

## Closest Pair Points

1. pair<P,P> **closestPair**(polygon p) {
2. **int** n=p.size(), s=0, t=1, m=2, S[n]; S[0] = 0, S[1] = 1;
3. sort(p.begin(), p.end()); // "p < q" <=> "p.x < q.x"
4. **double** d = norm(p[s]-p[t]);
5. **for** (**int** i = 2; i < n; S[m++] = i++)
6. **for**(**int** j = 0; j < m; j ++){
7. **if** (norm(p[S[j]]-p[i])<d)d = norm(p[s=S[j]]-p[t=i]);
8. **if** (real(p[S[j]]) < real(p[i]) - d) S[j--] = S[--m];
9. }
10. **return** make\_pair( p[s], p[t] );
11. }

## Intriangle

1. **bool** **intriangle**(P a, P b, P c, P p) {
2. a -= p; b -= p; c -= p;
3. **return** cross(a, b) >= 0 &&
4. cross(b, c) >= 0 &&
5. cross(c, a) >= 0;
6. }

## Three Point Circle

1. P **three\_point\_circle**(**const** P& a, **const** P& b, **const** P& c) {
2. P x = 1.0/conj(b - a), y = 1.0/conj(c - a);
3. **return** (y-x)/(conj(x)\*y - x\*conj(y)) + a;
4. }

## Circle\_circle\_intersect

1. pair<P,P>**c\_c\_intersect**(**const** P& c1,**const** **double**& r1,
2. **const** P& c2, **const** **double**& r2) {
3. P A = conj(c2-c1);
4. P B = (r2\*r2-r1\*r1-(c2-c1)\*conj(c2-c1)), C = r1\*r1\*(c2-c1);
5. P D = B\*B-4.0\*A\*C;
6. P z1 = (-B+sqrt(D))/(2.0\*A)+c1;
7. P z2 = (-B-sqrt(D))/(2.0\*A)+c1;
8. **return** pair<P, P>(z1, z2);
9. }

# Solution Ideas

* Dynamic Programming

– Drop a parameter, recover from others

– Swap answer and a parameter

– Parsing CFGs: CYK Algorithm

– Optimizations

∗ Convex hull optimization

· dp[*i*] = minj < i {dp[*j*] + b[*j*]×a[*i*]}

· b[*j*] ≥ b[*j* + 1]

· optionally a[*i*] ≤ a[*i* + 1]

· O(n^2 ) to O(n)

∗ Divide and conquer optimization

· dp[*i*][*j*] = mink < j {dp[*i* − 1][k] + C[k][*j*]}

· A[*i*][*j*] ≤ A[*i*][*j* + 1]

· O(kn^2 ) to O(knlogn)

· sufficient:C[a][c]+C[b][d]≤C[a][d]+C[b][c],

a ≤ b ≤ c ≤ d (QI)

∗ Knuth optimization

· dp[*i*][*j*] = mini<k<j {dp[*i*][k]+dp[k][*j*]+C[*i*][*j*]}

· A[*i*][*j* − 1] ≤ A[*i*][*j*] ≤ A[*i* + 1][*j*]

· O(n^3 ) to O(n^2 )

· sufficient:QI and C[b][c] ≤ C[a][d],a≤b≤c≤d

• Greedy

• Randomized

• Optimizations

– Use bitset (/64)

– Switch order of loops (cache locality)

• Process queries offline

– Mo’s algorithm

• Square-root decomposition

• Precomputation

• Efficient simulation

– Mo’s algorithm

– Sqrt decomposition

– Store 2^k jump pointers

• Data structure techniques

– Sqrt buckets

– Store 2^k jump pointers

– 2^k merging trick

• Counting

– Inclusion-exclusion principle

– Generating functions

• Graphs

– Can we model the problem as a graph?

– Can we use any properties of the graph?

– Strongly connected components

– Cycles (or odd cycles)

– Bipartite (no odd cycles)

∗ Bipartite matching

∗ Hall’s marriage theorem

∗ Stable Marriage

– Cut vertex/bridge

– Biconnected components

– Degrees of vertices (odd/even)

– Trees

∗ Heavy-light decomposition

∗ Centroid decomposition

∗ Least common ancestor

∗ Centers of the tree

– Eulerian path/circuit

– Chinese postman problem

– Topological sort

– (Min-Cost) Max Flow

– Min Cut

∗ Maximum Density Subgraph

– Huffman Coding

– Min-Cost Arborescence

– Steiner Tree

– Kirchoff’s matrix tree theorem

– Prüfer sequences

– Lovász Toggle

– Look at the DFS tree (which has no cross-edges)

• Mathematics

– Is the function multiplicative?

– Look for a pattern

– Permutations

∗ Consider the cycles of the permutation

– Functions

∗ Sum of piecewise-linear functions is a

piecewise-linear function

∗ Sum of convex(concave)functions is convex (concave)

– Modular arithmetic

∗ Chinese Remainder Theorem

∗ Linear Congruence

– Sieve

– System of linear equations

– Values to big to represent?

∗ Compute using the logarithm

∗ Divide everything by some large value

– Linear programming

∗ Is the dual problem easier to solve?

• Logic

– 2-SAT

– XOR-SAT (Gauss elimination or Bipartite matching)

• Meet in the middle

• Only work with the smaller half (log(n))

• Strings

– Trie (maybe over something weird, like bits)

– Suffix array

– Suffix automaton (+DP?)

– Aho-Corasick

– eerTree

– Work with S + S

• Hashing

• Euler tour, tree to array

• Segment trees

– Lazy propagation

– Persistent

– Implicit

– Segment tree of X

• Geometry

– Minkowski sum (of convex sets)

– Rotating calipers

– Sweep line (horizontally or vertically?)

– Sweep angle

– Convex hull

• Fix a parameter (possibly the answer).

• Are there few distinct values?

• Binary search

• Sliding Window (+ Monotonic Queue)

• Computing a Convolution? Fast Fourier Transform

• Exact Cover (+ Algorithm X)

• Cycle-Finding

• What is the smallest set of values that identify the solution?

The cycle structure of the permutation? The powers of primes

in the factorization?

• Look at the complement problem

– Minimize something instead of maximizing

• Immediately enforce necessary conditions. (All values greater

than 0? Initialize them all to 1)

• Add large constant to negative numbers to make them positive

• Counting/Bucket sort

# Debugging Tips

• Stack overflow? Recursive DFS on tree that is actually a long

path?

• Rounding negative numbers?

• Double

• Wrong Answer?

– Quitar el freopen,

– no mezclar cin con scanf

– Ver si hay que imprimir fin de linea

– Leer nuevamente el problema.

– Ver si es multiple casos, repetir el mismo caso varias

veces.

– long long

– Posibles Casos:

∗ n = 0,n = −1,n = 1,n = 2^31 − 1 or n = −2^31

∗ La lista esta vacia o con un solo elemento

∗ n is even, n is odd

∗ El Grafo esta vacion o contiene un solo vertice

∗ El Grafo es un multigrafo (lazo o multiple aristas)

∗ El Polygono es convexo o no

–Hay condicion inicial para los casos pequeños

–Estas utilizando el algoritmo correcto

– Explique su solucion a algien

– ¿Usa usted algunas funciones que usted completamente no

comprende? ¿Puede que STL funcione?

– ¿Puede que usted (o alguien más) debiera reescribir la

solución?

• Run-Time Error?

–Verificar el tamaño de los arreglos

–Division por 0