

El Bicho

DondeEstasCR7



06/02/2026

Índice

1. Algos	3
1.1. Binary Search	3
1.2. Fast Io	3
1.3. Sliding Window	4
1.4. Tablas Y Cotas	5
1.5. Two Pointers	7
2. Bit Manipulation	8
2.1. Bits	8
3. Combinatory	9
3.1. Combi Brute Sin Mod	9
3.2. Combinatory	9
4. Data Structures	10
4.1. Fenwick Tree	10
4.2. Find Two Numbers	11
4.3. Segment Tree	11
4.4. Sparse Table	12
5. Dp	12
5.1. Digit Dp Pattern	12
5.2. Digit Dp	13
5.3. Edit Distance	14
5.4. Knapsack	14
5.5. Lcs	14
6. Geometry	15
6.1. 2D Geometry	15
6.2. Convex Hull	22
7. Graph	23
7.1. Bellman Ford	23
7.2. Bfs	23
7.3. Bipartite	24

7.4. Dfs	24
7.5. Dfs 2D	25
7.6. Dijkstra	25
7.7. Disjoint Set Union Dsu	26
7.8. Floyd Warshall	26
7.9. Kruskal	26
7.10. Lowest Common Ancestor Lca	27
7.11. Prim	28
7.12. Scc	29
7.13. Topo Sort Dfs	30
7.14. Topo Sort Kahns Bfs	30
8. Manhattan Distance	31
8.1. Farthest Pair Of Points	31
8.2. Nearest Neighbor In Each Octant	31
9. Number Theory	32
9.1. Euler Toliente	32
9.2. Gcd Lcm	32
9.3. Number Theory	33
9.4. Phi Euler	33
9.5. Potenciacion Binaria	34
9.6. Sieve	34
9.7. Sieve Bitset	34
9.8. Sum Of Divisors	35
10. Numerical Methods	35
10.1. Calculating The Determinant Of A Matrix By Gauss	35
10.2. Gauss Method For Solving System Of Linear Equations	36
10.3. Gauss Solving Modular Slae	37
10.4. Integration By Simpson'S Formula	37
11. String	37
11.1. Kmp	37
11.2. Trie	38
11.3. Z Algorithm	39

1. Algos

1.1. Binary Search

```
1 // binary search en array ordenado
2 int n, target; cin >> n >> target;
3 vector<int> a(n);
4 for (int i = 0; i < n; i++) cin >> a[i];
5
6 // encontrar primera posición >= target
7 int l = 0, r = n - 1, first_pos = n;
8 while (l <= r) {
9     int mid = (l + r) / 2;
10    if (a[mid] >= target) {
11        first_pos = mid;
12        r = mid - 1;
13    } else {
14        l = mid + 1;
15    }
16}
17
18 // encontrar última posición <= target
19 l = 0, r = n - 1;
20 int last_pos = -1;
21 while (l <= r) {
22     int mid = (l + r) / 2;
23     if (a[mid] <= target) {
24         last_pos = mid;
25         l = mid + 1;
26     } else {
27         r = mid - 1;
28     }
29}
30
31 // binary search en función monótona
32 function<bool(int)> check = [&](int x) {
33     return true; // condición
34};
35 l = 0, r = 1e9;
36 int ans = -1;
37 while (l <= r) {
38     int mid = (l + r) / 2;
39     if (check(mid)) {
40         ans = mid;
41         l = mid + 1; // o r = mid - 1 dependiendo del problema
42     } else {
43         r = mid - 1; // o l = mid + 1
44     }
45}
```

1.2. Fast Io

```
1 #include <bits/stdc++.h>
2 #include <ext/pb_ds/assoc_container.hpp>
3 #include <ext/pb_ds/tree_policy.hpp>
4
5 #define cpu() ios::sync_with_stdio(false);cin.tie(nullptr);
6
7 using namespace std;
```

```

8  using namespace __gnu_pbds;
9  template <class T>
10 using ordered_set = tree<T, null_type, less_equal<T>, rb_tree_tag,
11     tree_order_statistics_node_update>;
12
13 #define pb push_back
14 #define sz(a) ((int)(a).size())
15 #define ff first
16 #define ss second
17 #define all(a) (a).begin(), (a).end()
18 #define allr(a) (a).rbegin(), (a).rend()
19 #define approx(a) fixed << setprecision(a)
20
21 template <class T> void read(vector<T> &v);
22 template <class F, class S> void read(pair<F, S> &p);
23 template <class T, size_t Z> void read(array<T, Z> &a);
24 template <class T> void read(T &x) {cin >> x;}
25 template <class R, class... T> void read(R& r, T&... t){read(r); read(t...);}
26 template <class T> void read(vector<T> &v) {for(auto& x : v) read(x);}
27 template <class F, class S> void read(pair<F, S> &p) {read(p.ff, p.ss);}
28 template <class T, size_t Z> void read(array<T, Z> &a) { for(auto &x : a) read(x); }
29
30 template <class F, class S> void pr(const pair<F, S> &x);
31 template <class T> void pr(const T &x) {cout << x;}
32 template <class R, class... T> void pr(const R& r, const T&... t) {pr(r); pr(t...);}
33 template <class F, class S> void pr(const pair<F, S> &x) {pr("[", x.ff, ",\"", x.ss, "\"]\n");}
34 void ps() {pr("\n");}
35 template <class T> void ps(const T &x) {pr(x); ps();}
36 template <class T> void ps(vector<T> &v) {for(auto& x : v) pr(x, ' '); ps();}
37 template <class T, size_t Z> void ps(const array<T, Z> &a) { for(auto &x : a) pr(x, ' '); ps();
38   (); }
39
40 using ll = long long;
41 const double PI = 3.141592653589793;
42 const ll MX = 1e9 + 1;
43
44 void solve() {
45
46 }
47
48 int main() {
49   cpu();
50
51   int t = 1;
52   //cin >> t;
53   while (t--) {
54     solve();
55   }
56
57   return 0;
58 }
```

1.3. Sliding Window

```

1 int n, k; cin >> n >> k;
2 vector<int> a(n);
3 for (int i = 0; i < n; i++) cin >> a[i];
```

```

4 // ventana deslizante de tamaño k
5 deque<int> dq;
6 for (int i = 0; i < n; i++) {
7     while (!dq.empty() && dq.front() <= i - k) dq.pop_front();
8     while (!dq.empty() && a[dq.back()] <= a[i]) dq.pop_back();
9     dq.push_back(i);
10    if (i >= k - 1) {
11        // a[dq.front()] es el máximo en ventana [i-k+1, i]
12    }
13}
14
15
16 // mínimo en ventana de tamaño k
17 dq.clear();
18 for (int i = 0; i < n; i++) {
19     while (!dq.empty() && dq.front() <= i - k) dq.pop_front();
20     while (!dq.empty() && a[dq.back()] >= a[i]) dq.pop_back();
21     dq.push_back(i);
22    if (i >= k - 1) {
23        // a[dq.front()] es el mínimo en ventana [i-k+1, i]
24    }
25}

```

1.4. Tablas Y Cotas

```

1 // Primeros 180 Primos:
2 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89
3 97 101 103 107 109 113 127 131 137 139 149 151 157 163 167 173 179
4 181 191 193 197 199 211 223 227 229 233 239 241 251 257 263 269 271
5 277 281 283 293 307 311 313 317 331 337 347 349 353 359 367 373 379
6 383 389 397 401 409 419 421 421 431 433 439 443 449 457 461 463 467 479
7 487 491 499 503 509 521 523 541 547 557 563 569 571 577 587 593 599
8 601 607 613 617 619 631 641 643 647 653 659 661 673 677 683 691 701
9 709 719 727 733 739 743 751 757 761 769 773 787 797 809 811 821 823
10 827 829 839 853 857 859 863 877 881 883 887 907 911 919 929 937 941
11 947 953 967 971 977 983 991 997 1009 1013 1019 1021 1031 1033 1039
12 1049 1051 1061 1063 1069
13
14 // Primos cercanos a 10^n
15 9941 9949 9967 9973 10007 10009 10037 10039 10061 10067 10069 10079
16 99961 99971 99989 99991 100003 100019 100043 100049 100057 100069
17 999959 999961 999979 999983 1000003 1000033 1000037 1000039 9999943
18 9999971 9999973 9999991 10000019 10000079 10000103 10000121 99999941
19 99999959 99999971 99999989 100000007 100000037 100000039 100000049
20 999999893 999999929 999999937 100000007 100000009 100000021
21 100000033
22
23 // Cantidad de primos menores que 10^n
24 pi(10^1) = 4 -> [2, 3, 5, 7]
25 pi(10^2) = 25 -> [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67,
   71, 73, 79, 83, 89]
26 pi(10^3) = 168
27 pi(10^4) = 1.229
28 pi(10^5) = 9.592
29 pi(10^6) = 78.498
30 pi(10^7) = 664.579
31 pi(10^8) = 5.761.455
32 pi(10^9) = 50.847.534
33 pi(10^10) = 455.052.511

```

```
34 pi(10^11) = 4.118.054.813
35 pi(10^12) = 37.607.912.018
36
37 // Cantidad de divisores
38 sigma0(60) = 12 -> [1, 2, 3, 4, 6, 10, 12, 15, 20, 30, 60]
39 sigma0(120) = 16 -> [1, 2, 3, 4, 6, 8, 10, 12, 15, 20, 24, 30, 40, 60, 120]
40 sigma0(180) = 18 -> [1, 2, 3, 4, 5, 6, 9, 10, 12, 15, 18, 20, 30, 36, 60, 90, 180]
41 sigma0(240) = 20 -> [1, 2, 3, 4, 5, 6, 8, 10, 12, 15, 16, 20, 24, 30, 40, 60, 80, 120, 240]
42 sigma0(360) = 24
43 sigma0(720) = 30
44 sigma0(840) = 32
45 sigma0(1.260) = 36
46 sigma0(1.680) = 40
47 sigma0(10.080) = 72
48 sigma0(15.120) = 80
49 sigma0(50.400) = 108
50 sigma0(83.160) = 128
51 sigma0(110.880) = 144
52 sigma0(498.960) = 200
53 sigma0(554.400) = 216
54 sigma0(1.081.080) = 256
55 sigma0(1.441.440) = 288
56 sigma0(4.324.320) = 384
57 sigma0(8.648.640) = 448
58
59 // Suma de divisores
60 sigma1(96) = 252 -> [1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48, 96]
61 sigma1(108) = 280 -> [1, 2, 3, 4, 6, 9, 12, 18, 27, 36, 54, 108]
62 sigma1(120) = 360 -> [1, 2, 3, 4, 5, 6, 8, 10, 12, 15, 20, 24, 30, 40, 60, 120]
63 sigma1(144) = 403 -> [1, 2, 3, 4, 6, 8, 9, 12, 16, 18, 24, 36, 48, 72, 144]
64 sigma1(168) = 480
65 sigma1(960) = 3.048
66 sigma1(1.008) = 3.224
67 sigma1(1.080) = 3.600
68 sigma1(1.200) = 3.844
69 sigma1(4.620) = 16.128
70 sigma1(4.680) = 16.380
71 sigma1(5.040) = 19.344
72 sigma1(5.760) = 19.890
73 sigma1(8.820) = 31.122
74 sigma1(9.240) = 34.560
75 sigma1(10.080) = 39.312
76 sigma1(10.920) = 40.320
77 sigma1(32.760) = 131.040
78 sigma1(35.280) = 137.826
79 sigma1(36.960) = 145.152
80 sigma1(37.800) = 148.800
81 sigma1(60.480) = 243.840
82 sigma1(64.680) = 246.240
83 sigma1(65.520) = 270.816
84 sigma1(70.560) = 280.098
85 sigma1(95.760) = 386.880
86 sigma1(98.280) = 403.200
87 sigma1(100.800) = 409.448
88 sigma1(491.400) = 2.083.200
89 sigma1(498.960) = 2.160.576
90 sigma1(514.080) = 2.177.280
91 sigma1(982.800) = 4.305.280
92 sigma1(997.920) = 4.390.848
```

```
93 sigma1(1.048.320) = 4.464.096
94 sigma1(4.979.520) = 22.189.440
95 sigma1(4.989.600) = 22.686.048
96 sigma1(5.045.040) = 23.154.768
97 sigma1(9.896.040) = 44.323.200
98 sigma1(9.959.040) = 44.553.600
99 sigma1(9.979.200) = 45.732.192
100
101 // Factoriales
102 0! = 1 (int)
103 1! = 1
104 2! = 2
105 3! = 6
106 4! = 24
107 5! = 120
108 6! = 720
109 7! = 5.040
110 8! = 40.320
111 9! = 362.880
112 10! = 3.628.800
113 11! = 39.916.800
114 12! = 479.001.600 (int)
115 13! = 6.227.020.800 (11)
116 14! = 87.178.291.200
117 15! = 1.307.674.368.000
118 16! = 20.922.789.888.000
119 17! = 355.687.428.096.000
120 18! = 6.402.373.705.728.000
121 19! = 121.645.100.408.832.000
122 20! = 2.432.902.008.176.640.000 (11)
123 21! = 51.090.942.171.709.400.000 (__int128_t)
124
125 // Límites de enteros
126 max signed char = 127
127 max unsigned char = 255
128 max signed int = 2.147.483.647
129 max unsigned int = 4.294.967.295
130 max signed long long = 9.223.372.036.854.775.807
131 max unsigned long long = 18.446.744.073.709.551.615
132 max signed __int128_t = 170.141.183.460.469.231.731.687.303.715.884.105.727
133 max unsigned __int128_t = 340.282.366.920.938.463.463.374.607.431.768.211.456
```

1.5. Two Pointers

```
1 int n, target; cin >> n >> target;
2 vector<int> a(n);
3 for (int i = 0; i < n; i++) cin >> a[i];
4
5 // encontrar subarray con suma = target
6 int l = 0, sum = 0;
7 for (int r = 0; r < n; r++) {
8     sum += a[r];
9     while (sum > target && l <= r) {
10         sum -= a[l++];
11     }
12     if (sum == target) {
13         // subarray [l, r] tiene suma = target
14     }
15 }
```

```

16 // encontrar número de subarrays con suma <= target
17 l = 0, sum = 0;
18 long long count = 0;
19 for (int r = 0; r < n; r++) {
20     sum += a[r];
21     while (sum > target && l <= r) {
22         sum -= a[l++];
23     }
24     count += r - l + 1;
25 }
26 // count = número de subarrays con suma <= target
27

```

2. Bit Manipulation

Técnicas para manipular bits individuales y operaciones a nivel de bit. Incluye macros útiles para competencias de programación.

2.1. Bits

Macros esenciales para manipulación de bits: verificar potencias de 2, establecer/limpiar bits, contar bits, y operaciones con LSB/MSB.

```

1 using ull = unsigned long long;
2 const ull UNSIGNED_LL_MAX = 18'446'744'073'709'551'615;
3 // Verifica si S es potencia de dos (y distinto de cero)
4 #define isPowerOfTwo(S) ((S) && !((S) & ((S) - 1)))
5 // Retorna la potencia de dos más cercana a S
6 #define nearestPowerOfTwo(S) (1LL << lround(log2(S)))
7 // Calcula S % N cuando N es potencia de dos
8 #define modulo(S, N) ((S) & ((N) - 1))
9
10 // Verifica si el bit está encendido (bit en 1)
11 #define isOn(S, i) ((S) & (1LL<<(i)))
12 // Enciende el bit (Lo pone en 1)
13 #define setBit(S, i) ((S) |= (1LL<<(i)))
14 // Apaga el bit (Lo pone en 0)
15 #define clearBit(S, i) ((S) &= ~(1LL<<(i)))
16 // Invierte el estado del bit (0 <-> 1)
17 #define toggleBit(S, i) ((S) ^= (1LL<<(i)))
18 // Enciende los primeros 'n' bits (idx-0)
19 #define setAll(S, n) ((S) = ((n)>=64 ? ~0LL : (1LL << (n))-1))
20
21 // Extrae el bit menos significativo 0100 (Least Significant Bit)
22 #define lsb(S) ((S) & -(S))
23 // Número de ceros a la derecha (Posición del LSB, idx-0)
24 #define idxLastBit(x) __builtin_ctzll(x)
25 // Extrae el bit más significativo 0100 (Most Significant Bit)
26 #define msb(S) (1LL << (63 - __builtin_clzll(S)))
27 // Posición del MSB (63 - ceros a la izquierda, idx-0)
28 #define idxFirstBit(x) (63 - __builtin_clzll(x))
29
30 #define countAllOnes(x) __builtin_popcountll(x)
31 // Apaga el último bit encendido (el menos significativo)
32 #define turnOffLastBit(S) ((S) & ((S) - 1))
33 // Enciende el último cero menos significativo
34 #define turnOnLastZero(S) ((S) | ((S) + 1))
35 // Apaga todos los bits encendidos más a la derecha consecutivos
36 #define turnOffLastConsecutiveBits(S) ((S) & ((S) + 1))
37 // Enciende los ceros consecutivos más a la derecha

```

```

38 #define turnOnLastConsecutiveZeroes(S) ((S) | ((S) - 1))
39
40 // Máscara de bits (mask -> subconjunto) 0(2^N)
41 for (int mask = 0; mask < (1 << N); mask++)
42
43 // Recorrer subconjuntos de un superconjunto (menos el vacío)
44 int b = 0b1011; // Representación binaria de un decimal en int
45 for (int i = b; i; i = (i - 1) & b) {
46     cout << bitset<4>(i) << "\n";
47 }
48
49 void printBin(ll x) {
50     // 63 -> unsigned ll, 62 -> ll, 31 -> unsigned int, 30 -> int
51     for (ll i = 63; i >= 0; i--)
52         cout << ((x >> i) & 1);
53     cout << '\n';
54 }
```

3. Combinatory

3.1. Combi Brute Sin Mod

```

1 // nCk brute force sin MOD n <= 20
2 long long nCk_bruteforce(long long n, long long k) {
3     if (k < 0 || k > n) return 0;
4     long long res = 1;
5     for (long long i = 1; i <= k; i++) {
6         res = res * (n - i + 1) / i; // aquí la división es exacta
7     }
8     return res;
9 }
10
11 // nPk brute force sin MOD n <= 20
12 long long nPk_bruteforce(long long n, long long k) {
13     if (k < 0 || k > n) return 0;
14     long long res = 1;
15     for (long long i = 0; i < k; i++) {
16         res *= (n - i);
17     }
18     return res;
19 }
```

3.2. Combinatory

OJO: Es necesario usar binpow con MOD primo

```

1 // Devuelve el inverso modular de a mod MOD
2 // Usa el Teorema Pequeño de Fermat: a^(MOD-2) === a^(-1) (mod MOD)
3 // (válido solo si MOD es primo)
4 ll inv(ll a, ll p = MOD) {
5     return binpow(a, p - 2, p);
6 }
7
8 // Factoriales e inversos factoriales precomputados
9 // fact[n] = n! mod MOD
10 // invf[n] = (n!)^(-1) mod MOD
11 // Precomputa en O(n)
12 vector<ll> fact(MAXN + 1), invf(MAXN + 1);
13 
```

```

14 void precompute_factorials() {
15     fact[0] = 1;
16     for (int i = 1; i <= MAXN; i++) {
17         fact[i] = fact[i - 1] * i % MOD;
18     }
19     invf[MAXN] = inv(fact[MAXN]);
20     for (int i = MAXN; i > 0; i--) {
21         invf[i - 1] = invf[i] * i % MOD;
22     }
23 }
24
25 // Combinatoria de n en k: nCk(n, k) para n <= 10^6
26 // "n choose k" = n! / (k! * (n-k)!) mod MOD
27 // Retorna 0 si k > n
28 ll nCk(ll n, ll k) {
29     if (k < 0 || k > n) return 0;
30     return fact[n] * invf[k] % MOD * invf[n - k] % MOD;
31 }
32
33 // Permutación de n en k: nPk(n, k) para n <= 10^6
34 // Calcula permutaciones: "n permute k" = n! / (n-k)! mod MOD
35 // Retorna 0 si k > n
36 ll nPk(ll n, ll k) {
37     if (k < 0 || k > n) return 0;
38     return fact[n] * invf[n - k] % MOD;
39 }

```

4. Data Structures

4.1. Fenwick Tree

```

1 struct FenwickTree {
2     vector<long long> tree;
3     int n;
4
5     FenwickTree(int size) : n(size + 1) {
6         tree.assign(n, 0);
7     }
8
9     void update(int idx, long long delta) {
10        for (idx++; idx < n; idx += idx & -idx) {
11            tree[idx] += delta;
12        }
13    }
14
15    long long query(int idx) {
16        long long sum = 0;
17        for (idx++; idx > 0; idx -= idx & -idx) {
18            sum += tree[idx];
19        }
20        return sum;
21    }
22
23    long long range_query(int l, int r) {
24        return query(r) - query(l - 1);
25    }
26 };
27

```

```

28 // uso:
29 // int n; cin >> n;
30 // FenwickTree ft(n);
31 // for (int i = 0; i < n; i++) {
32 //   long long x; cin >> x;
33 //   ft.update(i, x);
34 }
35 // ft.update(idx, delta); // actualizar elemento en idx
36 // long long sum = ft.range_query(l, r); // suma en rango [l, r]

```

4.2. Find Two Numbers

```

1 // "find two number where the sum is x, and gcd(a, b) > 1" b
2 auto find = [&](ll x){
3     for(int d = 2; d <= x / 2; d++){
4         if(x % d == 0){
5             ll m = 1, n = (x / d) - 1;
6             ll a = d * m, b = d * n;
7             if(__gcd(a, b) > 1){
8                 cout << a << ' ' << b;
9                 ps();
10                return;
11            }
12        }
13    }
14};

```

4.3. Segment Tree

```

1 // "This segment_tree I understand better how it works"
2 template<typename T>
3 struct seg_tree {
4     int N;
5     T Z = 0;
6     vector<T> tree;
7
8     seg_tree(int N) : N(N) {
9         tree.resize(4 * N);
10    }
11
12    seg_tree(vector<T>& A) {
13        N = (int)A.size();
14        tree.resize(4 * N);
15        build(A, 1, 0, N-1);
16    }
17
18 private:
19    T op(T a, T b) {
20        return a + b;
21    }
22
23    void build(vector<T>& a, int node, int left, int right) {
24        if(left == right) {
25            tree[node] = a[left];
26            return;
27        }
28        int mid = (left + right) >> 1;
29        build(a, 2 * node, left, mid);
30        build(a, 2 * node + 1, mid + 1, right);
31        tree[node] = op(tree[2 * node], tree[2 * node + 1]);

```

```

32     }
33
34     void modify(int pos, T value, int node, int left, int right) {
35         if(left == right) {
36             tree[node] = value;
37             return;
38         }
39         int mid = (left + right) >> 1;
40         if(pos <= mid)
41             modify(pos, value, 2 * node, left, mid);
42         else
43             modify(pos, value, 2 * node + 1, mid + 1, right);
44         tree[node] = op(tree[2 * node], tree[2 * node + 1]);
45     }
46
47     T query(int l, int r, int node, int left, int right) {
48         if(r < left || l > right) return Z;
49         if(l <= left && right <= r) return tree[node];
50         int mid = (left + right) >> 1;
51         T leftSum = query(l, r, 2 * node, left, mid);
52         T rightSum = query(l, r, 2 * node + 1, mid + 1, right);
53         return op(leftSum, rightSum);
54     }
55
56 public:
57     void build(vector<T>& a) { build(a, 1, 0, N-1); }
58     void modify(int pos, T value) { modify(pos, value, 1, 0, N-1); }
59     T query(int l, int r) { return query(l, r, 1, 0, N-1); }
60 };

```

4.4. Sparse Table

```

1 int n; cin >> n;
2 vector<long long> a(n);
3 for (int i = 0; i < n; i++) cin >> a[i];
4 int k = log2(n) + 1;
5 vector<vector<long long>> st(n, vector<long long>(k));
6 for (int i = 0; i < n; i++) st[i][0] = a[i];
7 for (int j = 1; j < k; j++) {
8     for (int i = 0; i + (1 << j) <= n; i++) {
9         st[i][j] = min(st[i][j - 1], st[i + (1 << (j - 1))][j - 1]);
10    }
11 }
12 function<long long(int, int)> query = [&](int l, int r) {
13     int j = log2(r - l + 1);
14     return min(st[l][j], st[r - (1 << j) + 1][j]);
15 };
16 // query(l, r) = mínimo en rango [l, r] en O(1)
17 // cambiar min por max para máximo
18 // cambiar min por gcd para GCD en rango

```

5. Dp

5.1. Digit Dp Pattern

```

1 string pattern; cin >> pattern; // ejemplo: "xxxxx3xxxx" donde x = dígito libre
2 int n = pattern.size();
3 long long k; cin >> k; // modulo
4

```

```

5 vector<vector<vector<long long>>> dp(n, vector<vector<long long>>(k, vector<long long>(2,
6   -1)));
7
8 function<long long(int, int, bool, bool)> solve = [&](int pos, int rem, bool tight, bool
9   started) {
10  if (pos == n) {
11    return (started && rem == 0) ? 1LL : 0LL;
12  }
13  if (started && !tight && dp[pos][rem][tight ? 1 : 0] != -1) {
14    return dp[pos][rem][tight ? 1 : 0];
15  }
16  long long res = 0;
17  if (pattern[pos] != 'x' && pattern[pos] != 'X') {
18    int fixed_digit = pattern[pos] - '0';
19    bool new_tight = tight && (fixed_digit == 9);
20    bool new_started = started || (fixed_digit > 0);
21    int new_rem = (rem * 10 + fixed_digit) % k;
22    res += solve(pos + 1, new_rem, new_tight, new_started);
23  } else {
24    int limit = tight ? 9 : 9;
25    int start_digit = (pos == 0) ? 1 : 0; // primer dígito no puede ser 0
26    for (int d = start_digit; d <= limit; d++) {
27      bool new_tight = tight && (d == limit);
28      bool new_started = started || (d > 0);
29      int new_rem = (rem * 10 + d) % k;
30      res += solve(pos + 1, new_rem, new_tight, new_started);
31    }
32  }
33  if (started && !tight) {
34    dp[pos][rem][tight ? 1 : 0] = res;
35  }
36  return res;
37};
38
39 long long result = solve(0, 0, true, false);
40 // result = cantidad de números que siguen el patrón y son divisibles por k
41 // ejemplo: pattern = "xxxxx3xxxx", k = 7
42 // cuenta números tipo 1234534567 que son divisibles por 7
43 // x o X = dígito libre, cualquier otro carácter = dígito fijo

```

5.2. Digit Dp

```

1 string s; cin >> s; // número como string (puede ser muy grande, tipo 10^100)
2 int n = s.size();
3 long long k; cin >> k; // modulo
4
5 vector<vector<vector<long long>>> dp(n, vector<vector<long long>>(k, vector<long long>(2,
6   -1)));
7
8 function<long long(int, int, bool, bool)> solve = [&](int pos, int rem, bool tight, bool
9   started) {
10  if (pos == n) {
11    return (started && rem == 0) ? 1LL : 0LL;
12  }
13  if (started && !tight && dp[pos][rem][tight ? 1 : 0] != -1) {
14    return dp[pos][rem][tight ? 1 : 0];
15  }
16  long long res = 0;
17  int limit = tight ? (s[pos] - '0') : 9;

```

```

16    for (int d = 0; d <= limit; d++) {
17        bool new_tight = tight && (d == limit);
18        bool new_started = started || (d > 0);
19        int new_rem = (rem * 10 + d) % k;
20        res += solve(pos + 1, new_rem, new_tight, new_started);
21    }
22    if (started && !tight) {
23        dp[pos][rem][tight ? 1 : 0] = res;
24    }
25    return res;
26};
27
28 long long result = solve(0, 0, true, false);
29 // result = cantidad de números <= s que son divisibles por k
30 // para contar en rango [a, b]: result_b - result_a-1
31 // ejemplo: s = "1000000", k = 7 -> contar números de 0 a 1000000 divisibles por 7

```

5.3. Edit Distance

```

1 string s1, s2; cin >> s1 >> s2;
2 int n = s1.size(), m = s2.size();
3 vector<vector<int>> dp(n + 1, vector<int>(m + 1));
4 for (int i = 0; i <= n; i++) dp[i][0] = i;
5 for (int j = 0; j <= m; j++) dp[0][j] = j;
6 for (int i = 1; i <= n; i++) {
7     for (int j = 1; j <= m; j++) {
8         if (s1[i - 1] == s2[j - 1]) {
9             dp[i][j] = dp[i - 1][j - 1];
10        } else {
11            dp[i][j] = 1 + min({dp[i - 1][j], dp[i][j - 1], dp[i - 1][j - 1]});
12        }
13    }
14}
15 // dp[n][m] = edit distance (mínimo número de operaciones: insertar, eliminar, reemplazar)
16 // para convertir s1 en s2

```

5.4. Knapsack

```

1 int n, capacity; cin >> n >> capacity;
2 vector<int> weight(n), value(n);
3 for (int i = 0; i < n; i++) {
4     cin >> weight[i] >> value[i];
5 }
6 vector<long long> dp(capacity + 1, 0);
7 for (int i = 0; i < n; i++) {
8     for (int w = capacity; w >= weight[i]; w--) {
9         dp[w] = max(dp[w], dp[w - weight[i]] + value[i]);
10    }
11}
12 // dp[capacity] = valor máximo que se puede obtener con capacidad máxima
13 // para version con items ilimitados, cambiar el loop: for (int w = weight[i]; w <= capacity
14     ; w++)

```

5.5. Lcs

```

1 string s1, s2; cin >> s1 >> s2;
2 int n = s1.size(), m = s2.size();
3 vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));
4 for (int i = 1; i <= n; i++) {
5     for (int j = 1; j <= m; j++) {

```

```

6     if (s1[i - 1] == s2[j - 1]) {
7         dp[i][j] = dp[i - 1][j - 1] + 1;
8     } else {
9         dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
10    }
11 }
12 }
13 // dp[n][m] = longitud de LCS (Longest Common Subsequence)
14
15 string reconstruct_lcs() {
16     string lcs = "";
17     int i = n, j = m;
18     while (i > 0 && j > 0) {
19         if (s1[i - 1] == s2[j - 1]) {
20             lcs += s1[i - 1];
21             i--;
22             j--;
23         } else if (dp[i - 1][j] > dp[i][j - 1]) {
24             i--;
25         } else {
26             j--;
27         }
28     }
29     reverse(lcs.begin(), lcs.end());
30     return lcs;
31 }
31 // lcs = string de la LCS

```

6. Geometry

6.1. 2D Geometry

Cookbook Geometría 2D - Operaciones con Puntos, Vectores, Líneas y Polígonos

PROBLEMAS TÍPICOS Y SUS SOLUCIONES

- **¿Un segmento toca un rectángulo?**

Construir 4 lados del rectángulo como Line, usar segIntersect(seg, lado_i) para $i = 1..4$, y verificar con between/extremos dentro para cubrir el caso “segmento completamente dentro”.

Funciones clave: segIntersect, between, Point/Line.

- **¿Dos segmentos se cruzan?**

Usar segIntersect(l1, l2) si contar colineales y tocar endpoints, o segStrictlyIntersect(l1, l2) si solo quieres cruce estricto.

- **Distancia mínima entre dos segmentos**

Real d = segDist(l1, l2); Si $d = 0$ entonces se tocan o cruzan.

- **Distancia mínima de un punto a un segmento**

Real d = pointToSegDist(p, seg);

- **Distancia mínima de un punto a una recta infinita**

Real d = pointToLineDist(p, line);

- **¿Punto dentro de un polígono cualquiera?**

bool inside = pointInPoly(p, poly);

- **Área de un polígono (ordenado CCW o CW)**

T twice = area(poly); Real A = fabs((Real)twice) / 2.0;

- **¿Cuándo un triángulo es degenerado?**

Un triángulo con vértices a, b, c es degenerado (área = 0) si: (1) sus puntos son colineales `sign(cross(b-a, c-a)) == 0`, o (2) con lados a, b, c , falla la desigualdad triangular estricta ($a+b > c, a+c > b, b+c > a$), i.e., $a + b = c$ (se aplana).

- **¿Punto sobre el borde de un polígono?**

Recorrer lados $[i, i + 1]$, verificar si `pointOnSeg(p, Line(ai, aj))`.

- **¿Dos rectas infinitas se cruzan? Dame el punto**

Primero verificar que no sean paralelas: `sign(cross(direction(11), direction(12))) != 0`, luego `P<Real>inter = lineIntersection(11, 12);`

- **Ordenar vectores/puntos por ángulo alrededor del origen**

`sort(v.begin(), v.end(), polar<i64>);` o usar lambda con `up()` y `cross()`.

- **Vector normal a un segmento**

`P<T>d = direction(seg); P<T>n = rotate90(d);` (normal 90° CCW) o `P<Real>nu = normal(d);` (normal unitaria).

- **Detectar si un punto está a la izquierda/derecha de un borde**

`int s = side(p, a, b);` donde $s > 0$ izquierda (CCW), $s < 0$ derecha (CW), $s = 0$ colineal.

- **Clasificar orientación de un polígono**

`T twice = area(poly);` Si `twice > 0` es CCW, si `twice < 0` es CW.

- **Reflexión de un punto respecto a una recta**

`P<Real>pr = reflection(p, line);`

- **Proyección de un punto sobre una recta**

`P<Real>proj = projection(p, line);`

- **Intersección de rayo con segmento**

`rayIntersect(rayo1, rayo2)` si ambos modelados como `Line` con origen en `1.1[0]` y dirección `1.1[1]-1.1[0]`. Para rayo vs segmento, combinar con `segIntersect` o rediseñar con `side()` y `dot()`.

- **Rotar un punto/vector alrededor del origen**

`P<Real>rotado = rotate(p, angulo_en_radianes);` Para 90° CCW rápido: `rotate90(p);`

- **Calcular ángulo de un vector respecto al eje X**

`Real ang = angle(p);` devuelve `atan2(p.y, p.x)` en radianes.

TIPS IMPORTANTES

- Usa `Point = P<i64>` para coords enteras, evita overflow en `cross/dot`
- Usa `P<Real>` (long double) para resultados con decimales (distancias, intersecciones)
- `EPS = 1e-9` para comparaciones de flotantes
- `sign()` y `cmp()` manejan tolerancia automáticamente
- Para convex hull y polígonos, mantén puntos en sentido CCW
- Verifica casos especiales: colineales, segmentos degenerados (mismo punto)
- Al leer entrada: `Point p; cin >>p;` (operador sobrecargado)

```
1 // ===== Tipos base =====
2 // - i64: reemplazo al long long
3 // - Real: para cálculos con flotantes de más precisión (distancias, intersecciones, etc.).
4 using i64 = long long;
5 using Real = long double;
6 constexpr Real EPS = 1e-9;
7
8 // ===== sign / cmp =====
9 // - sign(x): devuelve -1, 0, 1 (según x < 0, x == 0, x > 0)
10 // - cmp(a, b): compara a y b con tolerancia (para Real).
11 //   cmp(a, b) == 0 -> a "igual" a b, cmp(a, b) < 0 -> a < b, etc.
12 template <typename T>
13 int sign(T x) {
14     return (x > 0) - (x < 0);
15 }
16 int sign(Real x) {
17     return (x > EPS) - (x < -EPS);
18 }
19
20 template <typename T>
21 int cmp(T a, T b) {
22     return sign(a - b);
23 }
24
25 // ===== Punto y Línea =====
26 // - con T = i64 es usual en problemas de coordenadas enteras.
27 // - P<T>: punto/vector en 2D con componentes de tipo T.
28 // - L<T>: línea o segmento definido por dos puntos (l[0], l[1]).
29 template <typename T>
30 struct P {
31     T x = 0, y = 0;
32     P(T x = 0, T y = 0) : x(x), y(y) {}
33     friend istream& operator>>(istream &is, P &p) { return is >> p.x >> p.y; }
34     friend ostream& operator<<(ostream &os, P p) { return os << p.x << ' ' << p.y; }
35     friend bool operator==(P a, P b) { return cmp(a.x, b.x) == 0 && cmp(a.y, b.y) == 0; }
36     friend bool operator!=(P a, P b) { return !(a == b); }
37     P operator-() const { return P(-x, -y); }
38     P& operator+=(P a) {
39         x += a.x; y += a.y;
40         return *this;
41     }
42     P& operator-=(P a) {
43         x -= a.x; y -= a.y;
44         return *this;
45     }
46     P& operator*=(T d) {
47         x *= d; y *= d;
48         return *this;
49     }
50     P& operator/=(T d) {
51         x /= d; y /= d;
52         return *this;
53     }
54     friend P operator+(P a, P b) { return P(a) += b; }
55     friend P operator-(P a, P b) { return P(a) -= b; }
56     friend P operator*(P a, T d) { return P(a) *= d; }
57     friend P operator/(P a, T d) { return P(a) /= d; }
58     friend bool operator<(P a, P b) {
59         int sx = cmp(a.x, b.x);
```

```
60     return (sx != 0 ? sx == -1 : cmp(a.y, b.y) == -1);
61 }
62 };
63
64 template <typename T>
65 struct L {
66     array<P<T>, 2> l;
67     L(P<T> a = {}, P<T> b = {}) : l{a, b} {}
68 };
69
70 // ===== Operaciones vectoriales básicas =====
71 // - dot(a, b): producto escalar.
72 // - cross(a, b): producto cruzado escalar (a.x * b.y - a.y * b.x).
73 // - cross(p, a, b): cross(a - p, b - p) orientación de p respecto a ab.
74 // - square(a): |a|^2.
75 // - dist2(a, b): |a-b|^2, sin sqrt.
76 // - length(a): |a|.
77 // - dist(a, b): distancia euclíadiana entre a y b.
78 template <typename T>
79 T dot(P<T> a, P<T> b) { return a.x * b.x + a.y * b.y; }
80 template <typename T>
81 T cross(P<T> a, P<T> b) { return a.x * b.y - a.y * b.x; }
82 template <typename T>
83 T cross(P<T> p, P<T> a, P<T> b) { return cross(a - p, b - p); }
84 template <typename T>
85 T square(P<T> a) { return dot(a, a); }
86 template <typename T>
87 T dist2(P<T> a, P<T> b) { return square(a - b); }
88 template <typename T>
89 Real length(P<T> a) { return sqrtl(square(a)); }
90 template <typename T>
91 Real dist(P<T> a, P<T> b) { return length(a - b); }
92
93 // ===== Direcciones, ángulos, normales =====
94 // - normal(a): vector unitario en dirección de a.
95 // - up(a): true si el vector está en el semiplano "de arriba" (para ordenar por ángulo).
96 // - polar(a, b): criterio de orden por ángulo polar (para sort).
97 // - parallel(a, b): vectores paralelos.
98 // - sameDirection(a, b): vectores paralelos y apuntando en misma dirección.
99 // - angle(p): atan2(y, x).
100 // - rotate90(p): rota 90° CCW (útil para una normal rápida).
101 // - rotate(p, ang): rota un vector por un ángulo ang en radianes.
102 template <typename T>
103 P<Real> normal(P<T> a) {
104     Real len = length(a);
105     return P<Real>(a.x / len, a.y / len);
106 }
107 template <typename T>
108 bool up(P<T> a) {
109     return sign(a.y) > 0 || (sign(a.y) == 0 && sign(a.x) > 0);
110 }
111 // 3 colinear? recuerda remover (0,0) si lo usas en ordenamientos polares
112 template <typename T>
113 bool polar(P<T> a, P<T> b) {
114     bool ua = up(a), ub = up(b);
115     return ua != ub ? ua : sign(cross(a, b)) == 1;
116 }
117 template <typename T>
118 bool parallel(P<T> a, P<T> b) {
```

```
119     return sign(cross(a, b)) == 0;
120 }
121 template <typename T>
122 bool sameDirection(P<T> a, P<T> b) {
123     return sign(cross(a, b)) == 0 && sign(dot(a, b)) == 1;
124 }
125 template <typename T>
126 Real angle(P<T> p) {
127     return atan2((Real)p.y, (Real)p.x);
128 }
129 template <typename T>
130 P<T> rotate90(P<T> p) {
131     return P<T>(-p.y, p.x);
132 }
133 P<Real> rotate(P<Real> p, Real ang) {
134     return P<Real>(p.x * cosl(ang) - p.y * sinl(ang),
135         p.x * sinl(ang) + p.y * cosl(ang));
136 }
137
138 // ====== Dirección de una línea ======
139 // - direction(l): vector l.l[1] - l.l[0] (dirección del segmento/recta).
140 // - parallel(l1, l2) / sameDirection(l1, l2): igual que para vectores pero con líneas.
141 template <typename T>
142 P<T> direction(L<T> l1) {
143     return l.l[1] - l.l[0];
144 }
145 template <typename T>
146 bool parallel(L<T> l1, L<T> l2) {
147     return sameDirection(direction(l1), direction(l2));
148 }
149 template <typename T>
150 bool sameDirection(L<T> l1, L<T> l2) {
151     return sameDirection(direction(l1), direction(l2));
152 }
153
154 // ====== Proyección, reflexión, distancias a recta ======
155 // - projection(p, l): proyección ortogonal de p sobre la recta (infinita) l.
156 // - reflection(p, l): reflejo de p respecto a la recta l.
157 // - pointToLineDist(p, l): distancia mínima de p a la recta infinita que pasa por l.
158 P<Real> projection(P<Real> p, L<Real> l) {
159     auto d = direction(l);
160     return l.l[0] + d * (dot(p - l.l[0], d) / (Real)square(d));
161 }
162 P<Real> reflection(P<Real> p, L<Real> l) {
163     return projection(p, l) * 2 - p;
164 }
165 template <typename T>
166 Real pointToLineDist(P<T> p, L<T> l) {
167     if (l.l[0] == l.l[1]) return dist(p, l.l[0]);
168     return fabsl(cross(l.l[0] - l.l[1], l.l[0] - p)) / length(direction(l));
169 }
170
171 // ====== Intersección de líneas (rectas infinitas) ======
172 // - lineIntersection(l1, l2): punto de intersección de las rectas infinitas
173 //   definidas por l1 y l2.
174 // - OJO: no chequea si son paralelas; debes verificar antes que cross != 0.
175 template <typename T>
176 P<Real> lineIntersection(L<T> l1, L<T> l2) {
177     auto d1 = direction(l1);
```

```
178     auto d2 = direction(l2);
179     auto num = (Real)cross(d2, l1.l[0] - l2.l[0]);
180     auto den = (Real)cross(d2, d1);
181     return P<Real>(l1.l[0]) - d1 * (num / den);
182 }
183
184 // ===== Side / Between =====
185 // - side(p, a, b): orientación de p respecto al vector ab.
186 //   > 0: izquierda (CCW), < 0: derecha (CW), 0: colineal.
187 // - side(p, l): igual que antes pero con línea l.
188 // - between(m, l, r): true si m está entre l y r (incluyendo bordes).
189 template <typename T>
190 int side(P<T> p, P<T> a, P<T> b) {
191     return sign(cross(p, a, b));
192 }
193 template <typename T>
194 int side(P<T> p, L<T> l) {
195     return side(p, l.l[0], l.l[1]);
196 }
197 template <typename T>
198 bool between(T m, T l, T r) {
199     return cmp(l, m) == 0 || cmp(m, r) == 0 || (l < m) != (r < m);
200 }
201
202 // ===== Puntos sobre segmento =====
203 // - pointOnSeg(p, l): true si p está sobre el segmento l (incluye endpoints).
204 // - pointStrictlyOnSeg(p, l): true si p está sobre el segmento pero no en los endpoints.
205 template <typename T>
206 bool pointOnSeg(P<T> p, L<T> l) {
207     return side(p, l) == 0 &&
208         between(p.x, l.l[0].x, l.l[1].x) &&
209         between(p.y, l.l[0].y, l.l[1].y);
210 }
211 template <typename T>
212 bool pointStrictlyOnSeg(P<T> p, L<T> l) {
213     if (side(p, l) != 0) return false;
214     auto d = direction(l);
215     return sign(dot(p - l.l[0], d)) * sign(dot(p - l.l[1], d)) < 0;
216 }
217
218 // ===== Solapamiento de intervalos =====
219 // - overlap(l1, r1, l2, r2): true si [l1, r1] y [l2, r2] se solapan (1D).
220 template <typename T>
221 bool overlap(T l1, T r1, T l2, T r2) {
222     if (l1 > r1) swap(l1, r1);
223     if (l2 > r2) swap(l2, r2);
224     return cmp(r1, l2) != -1 && cmp(r2, l1) != -1;
225 }
226
227 // ===== Intersección de segmentos / rayos =====
228 // - segIntersect(l1, l2): true si los segmentos se tocan o cortan
229 //   (incluye colineales solapados y tocar en vértices).
230 // - segStrictlyIntersect(l1, l2): true si se cortan estrictamente
231 //   (no cuenta tocar solo en un endpoint).
232 // - rayIntersect(l1, l2): considera l1 y l2 como rayos, intersectan "hacia adelante"
233 //   (no cuenta si solo coincide el origen).
234 template <typename T>
235 bool segIntersect(L<T> l1, L<T> l2) {
236     auto [p1, p2] = l1.l;
```

```
237     auto [q1, q2] = 12.1;
238     return overlap(p1.x, p2.x, q1.x, q2.x) &&
239         overlap(p1.y, p2.y, q1.y, q2.y) &&
240         side(p1, 12) * side(p2, 12) <= 0 &&
241         side(q1, 11) * side(q2, 11) <= 0;
242 }
243 template <typename T>
244 bool segStrictlyIntersect(L<T> l1, L<T> l2) {
245     auto [p1, p2] = l1.1;
246     auto [q1, q2] = l2.1;
247     return side(p1, l2) * side(p2, l2) < 0 &&
248         side(q1, l1) * side(q2, l1) < 0;
249 }
250 template <typename T>
251 bool rayIntersect(L<T> l1, L<T> l2) {
252     auto v1 = direction(l1);
253     auto v2 = direction(l2);
254     int x = sign(cross(v1, v2));
255     return x != 0 && side(l1.1[0], l2) == x && side(l2.1[0], l1) == -x;
256 }
257 // ===== Distancias punto-segmento / segmento-segmento =====
258 // - pointToSegDist(p, l): distancia mínima de p al segmento l.
259 // - segDist(l1, l2): distancia mínima entre dos segmentos (0 si se intersectan).
260 template <typename T>
261 Real pointToSegDist(P<T> p, L<T> l) {
262     auto d = direction(l);
263     if (sign(dot(p - l.1[0], d)) >= 0 && sign(dot(p - l.1[1], d)) <= 0) {
264         return pointToLineDist(p, l);
265     } else {
266         return min(dist(p, l.1[0]), dist(p, l.1[1]));
267     }
268 }
269 template <typename T>
270 Real segDist(L<T> l1, L<T> l2) {
271     if (segIntersect(l1, l2)) return 0;
272     return min({
273         pointToSegDist(l1.1[0], l2),
274         pointToSegDist(l1.1[1], l2),
275         pointToSegDist(l2.1[0], l1),
276         pointToSegDist(l2.1[1], l1)
277     });
278 }
279 // ===== Área de polígono y punto en polígono =====
280 // - area(a): devuelve 2 * área con signo del polígono a (ordenado).
281 //      >0 CCW, <0 CW, abs(area)/2.0 área real.
282 // - pointInPoly(p, a): true si p está dentro o sobre el borde del polígono a
283 //      (no necesariamente convexo).
284 template <typename T>
285 T area(vector<P<T>> a) {
286     T res = 0;
287     int n = (int)a.size();
288     for (int i = 0; i < n; i++) {
289         res += cross(a[i], a[(i + 1) % n]);
290     }
291     return res;
292 }
293 template <typename T>
```

```

296 bool pointInPoly(P<T> p, vector<P<T>> a) {
297     int n = (int)a.size(), res = 0;
298     for (int i = 0; i < n; i++) {
299         P<T> u = a[i], v = a[(i + 1) % n];
300         if (pointOnSeg(p, L<T>(u, v))) return true;
301         if (cmp(u.y, v.y) <= 0) swap(u, v);
302         if (cmp(p.y, u.y) > 0 || cmp(p.y, v.y) <= 0) continue;
303         res += cross(p, u, v) > 0;
304     }
305     return res;
306 }
307
308 // ===== Aliases finales =====
309 // - Point = P<i64>    puntos con coordenadas enteras.
310 // - Line   = L<i64>    segmentos/líneas con endpoints enteros.
311 // - Usa Real (long double) para distancias si necesitas precisión extra.
312 using Point = P<i64>;
313 using Line = L<i64>;
314
315 // ejemplo de uso rápido:
316 // Point a, b; cin >> a >> b;
317 // Line seg(a, b);
318 // if (segIntersect(seg, Line(Point(0,0), Point(10,0)))) { ... }

```

6.2. Convex Hull

```

1 struct Point {
2     long long x, y;
3     Point(long long x = 0, long long y = 0) : x(x), y(y) {}
4     Point operator-(const Point& p) const { return Point(x - p.x, y - p.y); }
5     long long cross(const Point& p) const { return x * p.y - y * p.x; }
6     long long cross(const Point& a, const Point& b) const { return (a - *this).cross(b - *this) }
7     bool operator<(const Point& p) const { return x < p.x || (x == p.x && y < p.y); }
8     bool operator==(const Point& p) const { return x == p.x && y == p.y; }
9 };
10
11 vector<Point> convex_hull(vector<Point>& points) {
12     int n = points.size();
13     if (n <= 1) return points;
14     sort(points.begin(), points.end());
15     vector<Point> hull;
16     for (int i = 0; i < n; i++) {
17         // Si quiero incluir coords colineales (< 0), si no lo quiero (<= 0)
18         while (hull.size() >= 2 && hull[hull.size() - 2].cross(hull.back(), points[i]) <= 0) {
19             hull.pop_back();
20         }
21         hull.push_back(points[i]);
22     }
23     int lower = hull.size();
24     for (int i = n - 2; i >= 0; i--) {
25         // Si quiero incluir coords colineales (< 0), si no lo quiero (<= 0)
26         while (hull.size() > lower && hull[hull.size() - 2].cross(hull.back(), points[i]) <= 0)
27             {
28                 hull.pop_back();
29             }
30             hull.push_back(points[i]);
}

```

```

31     hull.pop_back();
32     return hull;
33 }
34
35 // uso:
36 // int n; cin >> n;
37 // vector<Point> points(n);
38 // for (int i = 0; i < n; i++) {
39 //   cin >> points[i].x >> points[i].y;
40 // }
41 // vector<Point> hull = convex_hull(points);
42 // hull contiene los puntos del convex hull en orden counter-clockwise (CCW / Antihorario)
43 // reverse(hull) para obtenerlo en orden CW / horario

```

7. Graph

Algoritmos de grafos: DFS, BFS, componentes fuertemente conexas, y otras estructuras de datos para problemas de grafos.

7.1. Bellman Ford

```

1 int n, m; cin >> n >> m;
2 vector<tuple<int, int, long long>> edges;
3 for (int i = 0; i < m; i++) {
4     int u, v; cin >> u >> v;
5     long long w; cin >> w;
6     edges.emplace_back(u, v, w);
7 }
8 vector<long long> dist(n + 1, inf);
9 dist[1] = 0;
10 for (int i = 0; i < n - 1; i++) {
11     for (auto [u, v, w] : edges) {
12         if (dist[u] != inf) {
13             dist[v] = min(dist[v], dist[u] + w);
14         }
15     }
16 }
17 bool has_negative_cycle = false;
18 for (auto [u, v, w] : edges) {
19     if (dist[u] != inf && dist[u] + w < dist[v]) {
20         has_negative_cycle = true;
21         break;
22     }
23 }
24 // dist[u] = distancia minima desde nodo 1 hasta u
25 // has_negative_cycle = true si hay ciclo negativo alcanzable desde nodo 1

```

7.2. Bfs

```

1 vector<bool> vis(n+1);
2 queue<int> q;
3 function<void(int)> bfs = [&](int start) {
4     vis[start] = true;
5     q.push(start);
6     int levels = 1;
7     while (!q.empty()) {
8         int sz = q.size();
9         levels++;
10        while (sz--) {

```

```

11     int u = q.front();
12     q.pop();
13     for (int& v : adj[u]) {
14         if (vis[v]) continue;
15         vis[v] = true;
16         q.push(v);
17     }
18 }
19 }
20 };
21
22 for (int u = 1; u <= n; u++) {
23     if (vis[u]) continue;
24     bfs(u);
25 }
```

7.3. Bipartite

```

1 int N, M; cin >> N >> M;
2 vector<vector<int>> adj(N + 1);
3 while (M--) {
4     int u, v; cin >> u >> v;
5     adj[u].push_back(v);
6     adj[v].push_back(u);
7 }
8
9 vector<bool> vis(N + 1);
10 vector<int> col(N + 1, 0);
11 // bipartite graph
12 function<bool(int, int)> dfs = [&](int u, int c) {
13     vis[u] = 1;
14     col[u] = c;
15
16     for (auto v : adj[u]) {
17         if (vis[v] && col[u] == col[v]) return false;
18         else if (!vis[v] && !dfs(v, c ^ 1)) return false;
19     }
20     return true;
21 };
22
23 for (int i = 1; i <= N; i++) {
24     if (vis[i]) continue;
25     if (dfs(i, 1) == false) {
26         cout << "IMPOSSIBLE";
27         return;
28     }
29 }
30
31 for (int i = 1; i <= N; i++) cout << (col[i] ? 1 : 2) << ' ';
```

7.4. Dfs

```

1 vector<bool> vis(n+1);
2 function<void(int)> dfs = [&](int u) {
3     vis[u] = true;
4     for (int& v : adj[u]) {
5         if (vis[v]) continue;
6         dfs(v);
7     }
8 };
```

```

9
10 | for (int u = 1; u <= n; u++) {
11 |   if (vis[u]) continue;
12 |   dfs(u);
13 |

```

7.5. Dfs 2D

```

1 int N, M; cin >> N >> M;
2 vector<vector<char>> grid(N, vector<char>(M));
3 for (int i = 0; i < N; i++) {
4   for (int j = 0; j < M; j++) {
5     cin >> grid[i][j];
6   }
7 }
8
9 vector<vector<bool>> vis(N, vector<bool>(M));
10 int dx[4] = {-1, 1, 0, 0}, dy[4] = {0, 0, -1, 1};
11 function<void(int, int)> dfs = [&](int x, int y) {
12   vis[x][y] = 1;
13
14   for (int d = 0; d < 4; d++) {
15     int nx = x + dx[d], ny = y + dy[d];
16     if (0 <= nx && 0 <= ny && nx < N && ny < M && grid[nx][ny] == '.' && !vis[nx][ny]) {
17       dfs(nx, ny);
18     }
19   }
20 };
21
22 int comp = 0;
23 for (int i = 0; i < N; i++) {
24   for (int j = 0; j < M; j++) {
25     if (vis[i][j] || grid[i][j] == '#') continue;
26     dfs(i, j);
27     comp++;
28   }
29 }
30
31 cout << comp;

```

7.6. Dijkstra

```

1 int N, M; cin >> N >> M;
2 vector<vector<pair<int, long long>>> adj(N + 1);
3 for (int i = 0; i < M; i++) {
4   int u, v; cin >> u >> v;
5   long long w; cin >> w;
6   adj[u].emplace_back(v, w);
7 }
8 vector<long long> dis(N + 1, inf);
9 pqg<pair<long long, int>> pq;
10 dis[1] = 0;
11 pq.push({0LL, 1});
12 while (!pq.empty()) {
13   auto [d, node] = pq.top(); pq.pop();
14   if (dis[node] != d) continue;
15   for (auto [v, w] : adj[node]) {
16     if (d + w < dis[v]) {
17       dis[v] = d + w;
18       pq.push({dis[v], v});

```

```

19     }
20 }
21 }
22 for (int u = 1; u <= N; u++) {
23     cout << dis[u] << " ";
24 }
```

7.7. Disjoint Set Union Dsu

```

1 struct DSU {
2     vector<int> p, size;
3     DSU(int n){
4         p.resize(n + 1), size.resize(n + 1, 1);
5         for(int i = 1; i <= n; i++) p[i] = i;
6     }
7
8     int find(int x){
9         if(p[x] != x) p[x] = find(p[x]);
10        return p[x];
11    }
12
13    void merge(int x, int y){
14        x = find(x), y = find(y);
15        if(x == y) return;
16        if(size[x] < size[y]) swap(x, y);
17        size[x] += size[y];
18        p[y] = x;
19    }
20};
```

7.8. Floyd Warshall

```

1 int n; cin >> n;
2 vector<vector<long long>> dist(n + 1, vector<long long>(n + 1, inf));
3 for (int i = 1; i <= n; i++) dist[i][i] = 0;
4 int m; cin >> m;
5 for (int i = 0; i < m; i++) {
6     int u, v; cin >> u >> v;
7     long long w; cin >> w;
8     dist[u][v] = min(dist[u][v], w);
9     dist[v][u] = min(dist[v][u], w);
10}
11for (int k = 1; k <= n; k++) {
12    for (int i = 1; i <= n; i++) {
13        for (int j = 1; j <= n; j++) {
14            if (dist[i][k] != inf && dist[k][j] != inf) {
15                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
16            }
17        }
18    }
19}
20 // dist[i][j] = distancia minima entre nodo i y nodo j
```

7.9. Kruskal

```

1 struct DSU {
2     vector<int> p, size;
3     DSU(int n) {
4         p.resize(n + 1), size.resize(n + 1, 1);
5         for (int i = 1; i <= n; i++) p[i] = i;
```

```

6 }
7     int find(int x) {
8         if (p[x] != x) p[x] = find(p[x]);
9         return p[x];
10    }
11    void merge(int x, int y) {
12        x = find(x), y = find(y);
13        if (x == y) return;
14        if (size[x] < size[y]) swap(x, y);
15        size[x] += size[y];
16        p[y] = x;
17    }
18 };
19
20 int n, m; cin >> n >> m;
21 vector<tuple<long long, int, int>> edges;
22 for (int i = 0; i < m; i++) {
23     int u, v; cin >> u >> v;
24     long long w; cin >> w;
25     edges.emplace_back(w, u, v);
26 }
27 sort(edges.begin(), edges.end());
28 DSU dsu(n);
29 long long mst_cost = 0;
30 for (auto [w, u, v] : edges) {
31     if (dsu.find(u) != dsu.find(v)) {
32         dsu.merge(u, v);
33         mst_cost += w;
34     }
35 }
36 // mst_cost = costo del MST (Minimum Spanning Tree)

```

7.10. Lowest Common Ancestor Lca

```

1 struct LCA{
2     int n, l, timer = 0;
3     vector<vector<int>> up, adj;
4     vector<int> depth, in, out;
5
6     LCA(int _n) {
7         n = _n + 1;
8         l = ceil(log2(n));
9         up.resize(n, vector<int>(l + 1));
10        adj.resize(n);
11        depth.resize(n);
12        in.resize(n);
13        out.resize(n);
14    }
15
16    void add_edge(int p, int u){
17        adj[p].push_back(u);
18        adj[u].push_back(p);
19    }
20
21    void dfs(int u = 1, int p = 1){
22        up[u][0] = p;
23        depth[u] = depth[p] + 1;
24        in[u] = ++timer;
25        for(int level = 1; level <= l; level++){

```

```

26     up[u][level] = up[up[u][level - 1]][level - 1];
27 }
28 for(int v : adj[u]){
29     if(v == p) continue;
30     dfs(v, u);
31 }
32 out[u] = ++timer;
33 }

34
35 bool is_ancestor(int p, int u){
36     return in[p] <= in[u] && out[p] >= out[u];
37 }
38
39 int query(int u, int v){
40     if(is_ancestor(u, v)) return u;
41     if(is_ancestor(v, u)) return v;
42
43     for(int bit = 1; bit >= 0; bit--){
44         if(is_ancestor(up[u][bit], v)) continue;
45         u = up[u][bit];
46     }
47     return up[u][0];
48 }
49
50 int ancestor(int u, int k){
51     if(depth[u] <= k) return -1;
52     for(int bit = 0; bit <= 1; bit++){
53         if(k >> bit & 1) u = up[u][bit];
54     }
55     return u;
56 }
57
58 int distance(int u, int v){
59     return depth[u] + depth[v] - 2 * depth[query(u, v)];
60 }
61 };

```

7.11. Prim

```

1 int n, m; cin >> n >> m;
2 vector<vector<pair<int, long long>>> adj(n + 1);
3 for (int i = 0; i < m; i++) {
4     int u, v; cin >> u >> v;
5     long long w; cin >> w;
6     adj[u].emplace_back(v, w);
7     adj[v].emplace_back(u, w);
8 }
9 vector<bool> vis(n + 1);
10 pqg<pair<long long, int>> pq;
11 pq.push({0LL, 1});
12 long long mst_cost = 0;
13 while (!pq.empty()) {
14     auto [w, u] = pq.top(); pq.pop();
15     if (vis[u]) continue;
16     vis[u] = true;
17     mst_cost += w;
18     for (auto [v, weight] : adj[u]) {
19         if (!vis[v]) {
20             pq.push({weight, v});

```

```
21     }
22 }
23 }
24 // mst_cost = costo del MST (Minimum Spanning Tree)
```

7.12. Scc

Algoritmo de Tarjan para encontrar componentes fuertemente conexas (SCC) en un grafo dirigido.

```
1 // "These works to find a componente fuertemente conexa that it's in directed graph"
2 struct SCC {
3     int N = 0, id;
4     vector<vector<int>> adj;
5     vector<int> ind, low;
6     stack<int> s;
7     vector<bool> in_stack;
8     vector<vector<int>> components;
9     vector<int> component_id;
10
11 //1-indexed
12 SCC(int n = 0){ N = n + 1, adj.assign(N, {}); }
13 SCC(const vector<vector<int>> &_adj){ adj = _adj, N = adj.size(); }
14
15 void add_edge(int from, int to){
16     adj[from].push_back(to);
17 }
18
19 void dfs(int u){
20     low[u] = ind[u] = id++;
21     s.push(u);
22     in_stack[u] = true;
23     for(int v : adj[u]){
24         if(ind[v] == -1){
25             dfs(v);
26             low[u] = min(low[u], low[v]);
27         }else if(in_stack[v]){
28             low[u] = min(low[u], ind[v]);
29         }
30     }
31     if(low[u] == ind[u]){
32         components.emplace_back();
33         vector<int> & comp = components.back();
34         while(true){
35             assert(!s.empty());
36             int x = s.top(); s.pop();
37             in_stack[x] = false;
38             component_id[x] = components.size() - 1;
39             comp.push_back(x);
40             if(x == u) break;
41         }
42     }
43 }
44
45 vector<vector<int>> get(){
46     ind.assign(N, -1); low.assign(N, -1); component_id.assign(N, -1);
47     s = stack<int>();
48     in_stack.assign(N, false);
49     id = 0;
50     components = {};
51     for(int i = 1; i < N; i++)
```

```

52     if(ind[i] == -1) dfs(i);
53
54     // reverse(components.begin(), components.end()); return components; // SCC in
55     // topological order
55     return components; // SCC in reverse topological order
56 }
57 };

```

7.13. Topo Sort Dfs

```

1 // O(N + M). 0-indexed. Retorna cualquier toposort válido (no necesariamente
2   lexicográficamente mínima)
3 vector<int> topo_sort(vector<vector<int>>& adj){
4     int n = adj.size();
5     bool cycle = false;
6     vector<int> topo, color(n); // 0 = no visitado, 1 = visitando, 2 = terminado
7
8     function<void(int)> dfs = [&](int u){
9       color[u] = 1;
10      for (int v : adj[u]){
11        if (color[v] == 0 && !cycle) dfs(v);
12        else if (color[v] == 1) cycle = true; // ciclo detectado
13      }
14      color[u] = 2;
15      topo.push_back(u);
16    };
17
18    for (int i = 0; i < n; i++){
19      if (color[i] == 0 && !cycle) dfs(i);
20    }
21    if (cycle) return {};
22    reverse(topo.begin(), topo.end());
23    return topo;
24  };

```

7.14. Topo Sort Kahns Bfs

```

1 // O((N + M)*logN). 0-indexed. Topological sort (Kahn BFS) con min-heap (lexicográficamente
2   mínimo)
3 vector<int> topo_sort(int n, const vector<vector<int>>& adj) {
4   vector<int> indeg(n, 0); // in-degree de cada nodo
5   for (int u = 0; u < n; u++) {
6     for (int v : adj[u]) {
7       indeg[v]++;
8     }
9   }
10  // min-heap para siempre sacar el nodo de menor índice
11  priority_queue<int, vector<int>, greater<int>> pq;
12  for (int u = 0; u < n; u++) {
13    if (indeg[u] == 0) pq.push(u);
14  }
15
16  vector<int> order;
17  order.reserve(n);
18  while (!pq.empty()) {
19    int u = pq.top();
20    pq.pop();
    order.push_back(u);
  }

```

```

21     for (int v : adj[u]) {
22         if (--indeg[v] == 0) {
23             pq.push(v);
24         }
25     }
26 }
27 if ((int)order.size() != n) { return {}; }
28 return order;
29 }
```

8. Manhattan Distance

8.1. Farthest Pair Of Points

```

1 long long ans = 0;
2 for (int msk = 0; msk < (1 << d); msk++) {
3     long long mx = LLONG_MIN, mn = LLONG_MAX;
4     for (int i = 0; i < n; i++) {
5         long long cur = 0;
6         for (int j = 0; j < d; j++) {
7             if (msk & (1 << j)) cur += p[i][j];
8             else cur -= p[i][j];
9         }
10        mx = max(mx, cur);
11        mn = min(mn, cur);
12    }
13    ans = max(ans, mx - mn);
14 }
```

8.2. Nearest Neighbor In Each Octant

```

1 // Nearest Neighbor in each Octant in O(n log n)
2 struct point {
3     long long x, y;
4 };
5
6 // Returns a list of edges in the format (weight, u, v).
7 // Passing this list to Kruskal algorithm will give the Manhattan MST.
8 vector<tuple<long long, int, int>> manhattan_mst_edges(vector<point> ps) {
9     vector<int> ids(ps.size());
10    iota(ids.begin(), ids.end(), 0);
11    vector<tuple<long long, int, int>> edges;
12    for (int rot = 0; rot < 4; rot++) { // for every rotation
13        sort(ids.begin(), ids.end(), [&](int i, int j){
14            return (ps[i].x + ps[i].y) < (ps[j].x + ps[j].y);
15        });
16        map<int, int, greater<int>> active; // (xs, id)
17        for (auto i : ids) {
18            for (auto it = active.lower_bound(ps[i].x); it != active.end();
19                 active.erase(it++)) {
20                int j = it->second;
21                if (ps[i].x - ps[i].y > ps[j].x - ps[j].y) break;
22                assert(ps[i].x >= ps[j].x && ps[i].y >= ps[j].y);
23                edges.push_back({(ps[i].x - ps[j].x) + (ps[i].y - ps[j].y), i, j});
24            }
25            active[ps[i].x] = i;
26        }
27        for (auto &p : ps) { // rotate
28            if (rot & 1) p.x *= -1;
```

```

29         else swap(p.x, p.y);
30     }
31 }
32 return edges;
33 }
```

9. Number Theory

9.1. Euler Toliente

```

1 class EulerTotiente {
2     public:
3     /* metodo en O(sqrt(n))
4     template <typename T>
5     T euler_classic(T n) {
6         T result = n;
7         for(T i = 2; i * i <= n; i++) {
8             if(n % i == 0) {
9                 while(n % i == 0) n /= i;
10                result -= result / i;
11            }
12        }
13        if(n > 1) {
14            result -= result / n;
15        }
16        return result;
17    }
18
19    /* metodo en O(nlog(log(n)))
20    void euler_faster(int n) {
21        vector<int> phi(n + 1);
22        for(int i = 0; i <= n; i++) {
23            phi[i] = i;
24        }
25        for(int i = 2; i <= n; i++) {
26            if(phi[i] == i) {
27                for(int j = i; j <= n; j += i) {
28                    phi[j] -= phi[j] / i;
29                }
30            }
31        }
32        for(int i = 1; i <= n; i++) {
33            cout << i << ' ' << phi[i] << '\n';
34        }
35    }
36 };
```

9.2. Gcd Lcm

```

1 // Maximo comun divisor (GCD): Algoritmo de Euclides
2 int gcd(int a, int b) {
3     if (a > b) swap(a, b);
4     if (a == 0) return b;
5     return gcd(b % a, a);
6 }
7
8 // Minimo comun multiplo (LCM): Calculado con GCD
9 int lcm(int a, int b) {
10    return (a * b) / gcd(a, b);
```

11 | }

9.3. Number Theory

```

1 // Divisores de N: Hasta N = 10^6
2 vector<int> divisores(int N) {
3     vector<int> divs;
4     for (int d = 1; d * d <= N; d++) {
5         if (N % d == 0) {
6             divs.push_back(d);
7             if (N / d != d) divs.push_back(N / d);
8         }
9     }
10    return divs;
11 }
12
13 // Factorizacion de N: Hasta N = 10^6
14 vector<pair<int, int>> factorizar(int N) {
15     vector<pair<int, int>> facts;
16     for (int p = 2; p * p <= N; p++) {
17         if (N % p == 0) {
18             int exp = 0;
19             while (N % p == 0) {
20                 exp++;
21                 N /= p;
22             }
23             facts.push_back({ p, exp });
24         }
25     }
26     if (N > 1) facts.push_back({ N, 1 });
27     return facts;
28 }
29
30 // Primalidad: Hasta N = 10^6 - O(sqrt(N))
31 bool isPrime(int N) {
32     if (N < 2) return false;
33     for (int d = 2; d * d <= N; d++) {
34         if (N % d == 0) return false;
35     }
36     return true;
37 }

```

9.4. Phi Euler

$\Phi(n)$ = contar la cantidad de numero coprimos entre 1 a n

```

1 int phi(int n) {
2     int ans = n;
3     for(int i = 2; i * i <= n; i++) {
4         if(n % i == 0) {
5             while (n % i == 0) {
6                 n /= i;
7             }
8             ans -= ans / i;
9         }
10    }
11    if(n > 1) {
12        ans -= ans / n;
13    }
14    return ans;
15 }

```

```

16  /*
17   * phi(n) -> complex: O(log(log(n)))
18   void phi_1_to_n(int n) {
19     vector<int> phi(n + 1);
20     for (int i = 0; i <= n; i++)
21       phi[i] = i;
22
23     for (int i = 2; i <= n; i++) {
24       if (phi[i] == i) {
25         for (int j = i; j <= n; j += i)
26           phi[j] -= phi[j] / i;
27       }
28     }
29 }
```

9.5. Potenciacion Binaria

```

1 using ll = long long;
2 const int MAXN = 1e6; // límite superior de n
3 const ll MOD = 1e9 + 7; // primo grande
4
5 // Potenciacion binaria modular a^b mod p
6 ll binpow(ll a, ll b, ll m = MOD) {
7   a %= m;
8   ll res = 1;
9   while (b > 0) {
10     if (b & 1)
11       res = res * a % m;
12     a = a * a % m;
13     b >>= 1;
14   }
15   return res;
16 }
```

9.6. Sieve

```

1 // Criba de Eratostenes: Hasta N = 10^6
2 void sieve(vector<bool>& is_prime) {
3   int N = (int) is_prime.size();
4   if (!is_prime[0]) is_prime.assign(N+1, true);
5   is_prime[0] = is_prime[1] = false;
6   for (int p = 2; p * p <= N; p++) {
7     if (is_prime[p]) {
8       for (int i = p * p; i <= N; i += p) {
9         is_prime[i] = false;
10      }
11    }
12  }
13 }
```

9.7. Sieve Bitset

```

1 // Hasta N = 10^8 aprox en 1s
2 const int MAX_V = 1e7 + 5;
3 bitset<MAX_V> composite;
4 void sieve() {
5   composite[0] = composite[1] = true;
6   for (int i = 2; i * i < MAX_V; i++) {
7     if (composite[i]) continue;
```

```

8     for (int j = i * i; j < MAX_V; j += i) {
9         composite[j] = true;
10    }
11 }
12 }
13
14 int main() {
15     sieve();
16     for (int i = 2; i < 100; i++) {
17         cout << i << " is_prime:" << !composite[i] << '\n';
18     }
19 }
```

9.8. Sum Of Divisors

```

1 /* Sum of divs
2 long long SumOfDivisors(long long num) {
3     long long total = 1;
4
5     for (int i = 2; (long long)i * i <= num; i++) {
6         if (num % i == 0) {
7             int e = 0;
8             do {
9                 e++;
10                num /= i;
11            } while (num % i == 0);
12
13            long long sum = 0, pow = 1;
14            do {
15                sum += pow;
16                pow *= i;
17            } while (e-- > 0);
18            total *= sum;
19        }
20    }
21    if (num > 1) {
22        total *= (1 + num);
23    }
24    return total;
25 }
```

10. Numerical Methods

10.1. Calculating The Determinant Of A Matrix By Gauss

```

1 const double EPS = 1E-9;
2 int n;
3 vector < vector<double> > a (n, vector<double> (n));
4
5 double det = 1;
6 for (int i=0; i<n; ++i) {
7     int k = i;
8     for (int j=i+1; j<n; ++j)
9         if (abs (a[j][i]) > abs (a[k][i]))
10            k = j;
11     if (abs (a[k][i]) < EPS) {
12         det = 0;
13         break;
14     }
```

```

15     swap (a[i], a[k]);
16     if (i != k)
17         det = -det;
18     det *= a[i][i];
19     for (int j=i+1; j<n; ++j)
20         a[i][j] /= a[i][i];
21     for (int j=0; j<n; ++j)
22         if (j != i && abs (a[j][i]) > EPS)
23             for (int k=i+1; k<n; ++k)
24                 a[j][k] -= a[i][k] * a[j][i];
25 }
26
27 cout << det;

```

10.2. Gauss Method For Solving System Of Linear Equations

```

1 const double EPS = 1e-9;
2 const int INF = 2; // it doesn't actually have to be infinity or a big number
3
4 int gauss (vector < vector<double> > a, vector<double> & ans) {
5     int n = (int) a.size();
6     int m = (int) a[0].size() - 1;
7
8     vector<int> where (m, -1);
9     for (int col=0, row=0; col<m && row<n; ++col) {
10         int sel = row;
11         for (int i=row; i<n; ++i)
12             if (abs (a[i][col]) > abs (a[sel][col]))
13                 sel = i;
14         if (abs (a[sel][col]) < EPS)
15             continue;
16         for (int i=col; i<=m; ++i)
17             swap (a[sel][i], a[row][i]);
18         where[col] = row;
19
20         for (int i=0; i<n; ++i)
21             if (i != row) {
22                 double c = a[i][col] / a[row][col];
23                 for (int j=col; j<=m; ++j)
24                     a[i][j] -= a[row][j] * c;
25             }
26         ++row;
27     }
28
29     ans.assign (m, 0);
30     for (int i=0; i<m; ++i)
31         if (where[i] != -1)
32             ans[i] = a[where[i]][m] / a[where[i]][i];
33     for (int i=0; i<n; ++i) {
34         double sum = 0;
35         for (int j=0; j<m; ++j)
36             sum += ans[j] * a[i][j];
37         if (abs (sum - a[i][m]) > EPS)
38             return 0;
39     }
40
41     for (int i=0; i<m; ++i)
42         if (where[i] == -1)
43             return INF;

```

```

44     return 1;
45 }
```

10.3. Gauss Solving Modular Slae

```

1 int gauss (vector < bitset<N> > a, int n, int m, bitset<N> & ans) {
2     vector<int> where (m, -1);
3     for (int col=0, row=0; col<m && row<n; ++col) {
4         for (int i=row; i<n; ++i)
5             if (a[i][col]) {
6                 swap (a[i], a[row]);
7                 break;
8             }
9         if (! a[row][col])
10            continue;
11         where[col] = row;
12
13         for (int i=0; i<n; ++i)
14             if (i != row && a[i][col])
15                 a[i] ^= a[row];
16             ++row;
17     }
18     // The rest of implementation is the same as above
19 }
```

10.4. Integration By Simpson'S Formula

```

1 const int N = 1000 * 1000; // number of steps (already multiplied by 2)
2
3 double simpson_integration(double a, double b){
4     double h = (b - a) / N;
5     double s = f(a) + f(b); // a = x_0 and b = x_2n
6     for (int i = 1; i <= N - 1; ++i) { // Refer to final Simpson's formula
7         double x = a + h * i;
8         s += f(x) * ((i & 1) ? 4 : 2);
9     }
10    s *= h / 3;
11    return s;
12 }
```

11. String

11.1. Kmp

```

1 string s; cin >> s;
2 int n = s.size();
3 vector<int> pi(n);
4 for (int i = 1; i < n; i++) {
5     int j = pi[i - 1];
6     while (j > 0 && s[i] != s[j]) {
7         j = pi[j - 1];
8     }
9     if (s[i] == s[j]) j++;
10    pi[i] = j;
11 }
12
13 string pattern, text;
14 cin >> pattern >> text;
15 string combined = pattern + "#" + text;
16 vector<int> pi_combined(combined.size());
```

```
17 for (int i = 1; i < combined.size(); i++) {
18     int j = pi_combined[i - 1];
19     while (j > 0 && combined[i] != combined[j]) {
20         j = pi_combined[j - 1];
21     }
22     if (combined[i] == combined[j]) j++;
23     pi_combined[i] = j;
24 }
25 vector<int> occurrences;
26 for (int i = pattern.size() + 1; i < combined.size(); i++) {
27     if (pi_combined[i] == pattern.size()) {
28         occurrences.push_back(i - 2 * pattern.size());
29     }
30 }
31 // pi[i] = longitud del prefijo mas largo que es sufijo en s[0..i]
32 // occurrences contiene las posiciones donde pattern aparece en text
```

11.2. Trie

```
1 struct Trie {
2     struct Node {
3         vector<int> next;
4         bool is_end;
5         int count;
6         Node() : next(26, -1), is_end(false), count(0) {}
7     };
8     vector<Node> nodes;
9     Trie() { nodes.emplace_back(); }
10
11    void insert(string& s) {
12        int cur = 0;
13        for (char c : s) {
14            int idx = c - 'a';
15            if (nodes[cur].next[idx] == -1) {
16                nodes[cur].next[idx] = nodes.size();
17                nodes.emplace_back();
18            }
19            cur = nodes[cur].next[idx];
20            nodes[cur].count++;
21        }
22        nodes[cur].is_end = true;
23    }
24
25    bool search(string& s) {
26        int cur = 0;
27        for (char c : s) {
28            int idx = c - 'a';
29            if (nodes[cur].next[idx] == -1) return false;
30            cur = nodes[cur].next[idx];
31        }
32        return nodes[cur].is_end;
33    }
34
35    int count_prefix(string& s) {
36        int cur = 0;
37        for (char c : s) {
38            int idx = c - 'a';
39            if (nodes[cur].next[idx] == -1) return 0;
40            cur = nodes[cur].next[idx];
41        }
42        return nodes[cur].count;
43    }
44 }
```

```
41     }
42     return nodes[cur].count;
43 }
44 };
45
46 // uso:
47 // Trie trie;
48 // int n; cin >> n;
49 // for (int i = 0; i < n; i++) {
50 //   string s; cin >> s;
51 //   trie.insert(s);
52 }
53 // string query; cin >> query;
54 // bool exists = trie.search(query); // true si query existe en el trie
55 // int count = trie.count_prefix(query); // cantidad de strings que tienen query como
56 // prefijo
57
57 Trie trie;
58 trie.insert("hello");
59 trie.insert("hell");
60 bool found = trie.search("hello"); // true
61 int count = trie.count_prefix("hel"); // 2 (hello y hell)
```

11.3. Z Algorithm

```
1 string s; cin >> s;
2 int n = s.size();
3 vector<int> z(n);
4 int l = 0, r = 0;
5 for (int i = 1; i < n; i++) {
6   if (i <= r) {
7     z[i] = min(r - i + 1, z[i - 1]);
8   }
9   while (i + z[i] < n && s[z[i]] == s[i + z[i]]) {
10     z[i]++;
11   }
12   if (i + z[i] - 1 > r) {
13     l = i;
14     r = i + z[i] - 1;
15   }
16 }
17
18 string pattern, text;
19 cin >> pattern >> text;
20 string combined = pattern + "#" + text;
21 vector<int> z_combined(combined.size());
22 int l_combined = 0, r_combined = 0;
23 for (int i = 1; i < combined.size(); i++) {
24   if (i <= r_combined) {
25     z_combined[i] = min(r_combined - i + 1, z_combined[i - l_combined]);
26   }
27   while (i + z_combined[i] < combined.size() && combined[z_combined[i]] == combined[i +
28     z_combined[i]]) {
29     z_combined[i]++;
30   }
31   if (i + z_combined[i] - 1 > r_combined) {
32     l_combined = i;
33     r_combined = i + z_combined[i] - 1;
34   }
35 }
```

```
34 }
35 vector<int> occurrences;
36 for (int i = pattern.size() + 1; i < combined.size(); i++) {
37     if (z_combined[i] == pattern.size()) {
38         occurrences.push_back(i - pattern.size() - 1);
39     }
40 }
41 // z[i] = longitud del substring mas largo que empieza en i y es prefijo de s
42 // occurrences contiene las posiciones donde pattern aparece en text
```