

GEMPRO Notebook

GEMPRO

Conteúdo

1	Contest	3
1.1	template.cpp	3
1.2	.bashrc	3
1.3	.vimrc	3
2	Matemática	3
2.1	Algoritmo Euclideano	3
2.2	Crivo	3
2.3	CRT	4
2.4	Teste de primalidade	4
3	Estruturas de Dados	4
3.1	Fenwick Tree	4
3.2	Segtree	5
3.3	Max Add Segtree	5
3.4	Lazy Segtree	6
3.5	Sparse Table	7
3.6	Disjoin Set Union	7
3.7	Zeta e Mobius	7
3.8	Árvore de Li Chao	8
3.9	Árvore de Li Chao++	8
3.10	Árvore Cartesiana	9
3.11	Convolução	9
3.11.1	FFT	10
3.11.2	NTT	10
4	Grafos	11
4.1	Caminho Euleriano	11
4.2	Max Flow	11
4.2.1	Dinic	11
4.3	Min Cost Flow	12
5	Árvores	12
5.1	Base	12
5.2	Movimento em árvores	13
5.3	HLD	13
5.4	LCA	14
6	Strings	14
6.1	Suffix Array	14
6.2	Maior prefixo comum (LCP)	15
6.3	KMP	15
7	Geometria	15
7.1	Base	15
7.2	Interseção de retas	15
7.3	Interseção de segmentos	16
7.4	Interseção de círculo e reta	16
7.5	Interseção de círculos	16
7.6	Tangentes comuns a dois círculos	16
7.7	Área de um polígono	16
7.8	Verificar se um ponto está dentro de um polígono	16
7.9	Soma Minkowski de polígonos convexos	16
7.10	Convex hull	16
7.11	Convex hull online	17
7.12	Menor distância euclideana	17
A	Fórmulas Úteis	17
A.1	Contagem e Probabilidade	17

A.2 Somas	17
B Teoremas e Fatos	18
C Ideias	18
C.1 Gerais	18
C.2 Calculando somas e integrais	18
C.3 Problemas de Otimização	19
D Debugging	19
D.1 Geral	19
D.2 WA - Wrong Answer	19
D.3 RE - Runtime Error	19

Contest

template.cpp

```
#include <bits/stdc++.h>

using namespace std;

using i64 = int64_t;
using i32 = int32_t;
using i128 = __int128;
using VI = vector<i32>;
using VVI = vector<VI>;
using PII = pair<i32, i32>;
using VII = vector<PII>;
using WVII = vector<VII>;

// debug stuff, no need to type at first
void PRINT(auto x) { cerr << '\t' << x << endl; }
void PRINT(const char *name, auto... x) {
    cerr << "DEBUG> " << name << " = [" << endl;
    (PRINT(x), ...);
    cerr << ']' << endl;
}

#define DEBUG(...) PRINT(__VA_ARGS__, __VA_ARGS__)

// #define int i64

i32 main() {
    cin.tie(0) -> sync_with_stdio(0);
    return 0;
}
```

.bashrc

```
c() {
    g++ "$1.cpp" -o "$1" -std=c++20
}
```

```
cs() {
    g++ "$1.cpp" -o "$1" -std=c++20
    ↪ -fsanitize=address,undefined,signed-integer-overflow -ggdb
}
```

.vimrc

```
syntax on
set cin ts=4 sw=4 udf cul nu is

ca Hash w !cpp -dD -P -fpreprocessed \| tr -d '[:space:]' \| md5sum \| cut -c
↪ -6
```

Matemática

Algoritmo Euclideano

Encontra inteiros x , y e g tais que $ax + by = g$ e $g = \gcd(a, b)$. Útil para calcular inversos módulo M . Complexidade: $O(\log(\min(a, b)))$.

8c6176 - math/egcd.cpp - 9 lines

```
i64 egcd(i64 a, i64 b, i64 &x, i64 &y) {
    if (b == 0) {
        x = 1, y = 0;
        return a;
    }
    i64 g = egcd(b, a % b, y, x);
    y -= (a / b) * x;
    return g;
} // 8c6176
```

Crivo

Encontra todos os números primos menores que n . Além disso, também encontra para cada inteiro i entre 0 e n , qual o menor fator primo de i . Com-

plexidade: $O(n)$.

157cee - math/sieve.cpp - 16 lines

```
struct Sieve {
    VI primes, spf;
    Sieve(int n): spf(n) {
        for (int i = 2; i < n; i++) {
            if (!spf[i]) {
                spf[i] = i;
                primes.push_back(i);
            }
            for (int j: primes) {
                if (j * i >= n) break;
                spf[i * j] = j;
                if (j == spf[i]) break;
            }
        }
    }
};
```

CRT

Teste de primalidade

Verificar se um número é primo em $O(\log(N))$.

83fc9d - math/primalty.cpp - 33 lines

```
i64 powm(i64 x, i64 e, i64 mod) {
    i64 r = 1;
    while (e) {
        if (e & 1) r = i128(r) * x % mod;
        e >>= 1;
        x = i128(x) * x % mod;
    }
    return r;
} // 3218d7

bool checkPrime(i64 p, i64 a) {
    i64 k = p - 1, d = 1;
```

```
while (~k & 1) k >>= 1;
for (i64 e = k; e; e >>= 1) {
    if (e & 1) d = i128(d) * a % p;
    a = i128(a) * a % p;
}
if (d == 1 || d == p - 1) return true;
while ((k <= 1) < p - 1) {
    d = i128(d) * d % p;
    if (d == p - 1) return true;
}
return false;
} // 479294

bool isPrime(i64 p) {
    if (p == 1) return false;
    for (int i: {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37}) {
        if (p == i) return true;
        if (!checkPrime(p, i)) return false;
    }
    return true;
} // c77db5
```

Estruturas de Dados

Fenwick Tree

Atua com um array de inteiros a_0, a_1, \dots, a_{n-1} . Suporta dois tipos de operação:

- $add(i, val)$: adiciona o valor val ao índice i ($a_i \rightarrow a_i + val$). Complexidade: $O(\log(n))$.
- $sum(L, R)$: Calcula a soma $a_L + a_{L+1} + \dots + a_{R-1}$. Complexidade: $O(\log(n))$.

1bc144 - dsa/fenwick-tree.cpp - 14 lines

```
struct Fenwick {
    vector<i64> t;
    int n;
```

```

Fenwick(int N) : t(N), n(N) {}
void add(int i, i64 val) {
    for (i++; i <= n; i += i & -i) t[i - 1] += val;
} // 196fa1
i64 sum(int r) {
    i64 s = 0;
    for (; r; r -= r & -r) s += t[r - 1];
    return s;
} // 46bb3f
i64 sum(int l, int r) { return sum(r) - sum(l); }
};

```

Segtree

Atua com uma operação \oplus em elementos de uma monoide S com elemento neutro e , em um array a_1, a_2, \dots, a_{n-1} com $a_i \in S$. Suporta dois tipos de operações:

- $set(i, val)$: modifica o array na posição i ($a_i \rightarrow val$). Complexidade: $O(\log(n))$.
- $sum(L, R)$: calcula $a_l \oplus a_{l+1} \oplus \dots \oplus a_{r-1}$. Complexidade: $O(\log(n))$.

b76e87 - dsa/segtree.cpp - 18 lines

```

template<class S, S (*op)(S, S), S (*e)()>
struct Segtree {
    vector<S> t;
    int n;
    Segtree(int N): t(2 * N, e()), n(N) {}
    void set(int i, S value) {
        t[i += n] = value;
        for (i >>= 1; i; i >>= 1) t[i] = op(t[i << 1], t[i << 1 | 1]);
    } // 630e57
    S query(int l, int r) {
        S al = e(), ar = e();
        for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
            if (l & 1) al = op(al, t[l++]);
            if (r & 1) ar = op(t[--r], ar);
        }
        return op(al, ar);
    }
};

```

```

} // 9ee903
};

```

Max Add Segtree

Segtree que permite soma em range, e calcular o máximo em um range

c008ec - dsa/max-add-segtree.cpp - 41 lines

```

struct MaxAddSegtree {
    int n, h;
    vector<i64> t, d;
    MaxAddSegtree(int n) : n(n), h(32 - __builtin_clz(n)), t(2 * n), d(n) {}
    void apply(int p, i64 v) {
        t[p] += v;
        if (p < n) d[p] += v;
    } // 94936b
    void build(int p) {
        while (p > 1) p >>= 1, t[p] = max(t[p << 1], t[p << 1 | 1]) + d[p];
    } // c20980
    void push(int p) {
        for (int s = h; s > 0; --s) {
            int i = p >> s;
            if (d[i]) {
                apply(i << 1, d[i]);
                apply(i << 1 | 1, d[i]);
                d[i] = 0;
            }
        }
    } // 452305
    void update(int l, int r, i64 v) {
        l += n, r += n;
        int l0 = l, r0 = r;
        for (; l < r; l >>= 1, r >>= 1) {
            if (l & 1) apply(l++, v);
            if (r & 1) apply(--r, v);
        }
        build(l0); build(r0 - 1);
    } // 8a7429
    i64 query(int l, int r) {

```

```

    l += n, r += n;
    push(l); push(r - 1);
    i64 res = -1e18;
    for (; l < r; l >>= 1, r >>= 1) {
        if (l & 1) res = max(res, t[l++]);
        if (r & 1) res = max(res, t[--r]);
    }
    return res;
} // f655b5
};

```

Lazy Segtree

Similar a uma segtree normal, mas suporta modificações em range. Suporta três tipos de operações:

- $set(i, val)$: modifica o array na posição i ($a_i \rightarrow val$). Complexidade: $O(\log(n))$.
- $sum(L, R)$: calcula $a_l \oplus a_{l+1} \oplus \dots \oplus a_{r-1}$. Complexidade: $O(\log(n))$.
- $apply(L, R, U)$: Aplica a função *mapping* no range $[L, R)$ ($a_i \rightarrow mapping(U, a_i)$, para $i = L, L+1, \dots, R-1$). Complexidade: $O(\log(n))$.

4ea409 - dsa/lazy-segtree.cpp - 51 lines

```

template<class S, S (*op)(S, S), S (*e)(), class U, S (*mapping)(U, S), U
    (*compose)(U, U), U (*id)()>
struct LazySegtree {
    vector<S> t;
    vector<U> d;
    int n;
    LazySegtree(int N): t(4 * N, e()), d(2 * N, id()), n(N) {}
    void push(int i) {
        if (i < n) {
            d[i << 1] = compose(d[i], d[i << 1]);
            d[i << 1 | 1] = compose(d[i], d[i << 1 | 1]);
        }
        if (i < 2 * n) {
            t[i << 1] = mapping(d[i], t[i << 1]);
            t[i << 1 | 1] = mapping(d[i], t[i << 1 | 1]);
            d[i] = id();
        }
    }
};

```

```

    }
} // 722677
void set(int j, S val) { set(j, val, 0, n, 1); }
void set(int j, S val, int tl, int tr, int i) {
    push(i);
    if (tl + 1 == tr) {
        t[i] = val;
        return;
    }
    int tm = (tl + tr) / 2;
    if (j < tm) set(j, val, tl, tm, i << 1);
    else set(j, val, tm, tr, i << 1 | 1);
    t[i] = op(t[i << 1], t[i << 1 | 1]);
} // eadaef
void apply(int l, int r, U u) { apply(l, r, u, 0, n, 1); }
void apply(int l, int r, U u, int tl, int tr, int i) {
    push(i);
    if (r <= tl || tr <= l) return;
    if (l <= tl && tr <= r) {
        t[i] = mapping(u, t[i]);
        if (i < 2 * n) d[i] = compose(u, d[i]);
        return;
    }
    int tm = (tl + tr) / 2;
    apply(l, r, u, tl, tm, i << 1);
    apply(l, r, u, tm, tr, i << 1 | 1);
} // 630c93
S sum(int l, int r) { return sum(l, r, 0, n, 1); }
S sum(int l, int r, int tl, int tr, int i) {
    push(i);
    if (r <= tl || tr <= l) return e();
    if (l <= tl && tr <= r) return t[i];
    int tm = (tl + tr) / 2;
    return op(sum(l, r, tl, tm, i << 1), sum(l, r, tm, tr, i << 1 | 1));
} // f3ba16
};

```

Sparse Table

Permite calcular soma de funções idempotentes em um intervalo de um array.

Ex: Calcular o mínimo no intervalo $[L, R]$ Complexidade:

- Construção: $O(n \log(n))$.
- Consulta: $O(1)$.

Outros exemplos de funções incluem *gcd*, *lcm* e *max*.

2f7d9d - dsa/sparse-table.cpp - 18 lines

```
template <class S, S (*op)(S, S)> struct SparseTable {
    vector<vector<S>> t;
    VI lg;
    void build(vector<S> &v) {
        int n = ssize(v);
        lg.assign(n + 1, 0);
        for (int i = 2; i <= n; i++) lg[i] = lg[i >> 1] + 1;
        t.assign(lg[n] + 1, vector<S>(n));
        t[0] = v;
        for (int k = 1; k <= lg[n]; k++)
            for (int i = 0; i + (1 << k) <= n; i++)
                t[k][i] = op(t[k - 1][i], t[k - 1][i + (1 << (k - 1))]);
    } // 007810
    int query(int l, int r) {
        int k = lg[r - l];
        return op(t[k][l], t[k][r - (1 << k)]);
    } // f9bbfc
};
```

Disjoin Set Union

Representa uma lista de conjuntos, e suporta dois tipos de operações:

- *merge*(X, Y): Junta os conjuntos dos números X e Y .
- *root*(X): encontra o elemento representante do conjunto de X . Dois números estão no mesmo conjunto quando eles tem o mesmo representante.

aca3d9 - dsa/dsu.cpp - 17 lines

```
struct DSU {
    VI par, cnt;
    DSU(int n): par(n), cnt(n, 1) {
        iota(par.begin(), par.end(), 0);
    }
    int root(int x) {
        if (par[x] == x) return x;
        return par[x] = root(par[x]);
    } // 809557
    bool merge(int x, int y) {
        x = root(x), y = root(y);
        if (x == y) return false;
        if (cnt[x] < cnt[y]) swap(x, y);
        cnt[x] += cnt[y], par[y] = x;
        return true;
    } // 3bef8c
};
```

Zeta e Mobius

Zeta: dada uma função $f : 2^{[1..n]} \rightarrow S$, encontra uma função $g : 2^{[1..n]} \rightarrow S$ tal que

$$g(A) = \sum_{B \subseteq A} f(B)$$

Mobius: realiza a transformação inversa.

6b8543 - dsa/zeta.cpp - 28 lines

```
VI subsetZeta(VI a) {
    int n = a.size();
    for (int j = 1; j < n; j <= 1)
        for (int i = 1; i < n; i++)
            if (i & j) a[i] += a[i ^ j];
    return a;
} // d4653e
VI subsetMobius(VI a) {
    int n = a.size();
    for (int j = 1; j < n; j <= 1)
```

```

        for (int i = 1; i < n; i++)
            if (i & j) a[i] -= a[i ^ j];
    return a;
} // 00943a
VI supersetZeta(VI a) {
    int n = a.size();
    for (int j = 1; j < n; j <= 1)
        for (int i = 1; i < n; i++)
            if (i & j) a[i ^ j] += a[i];
    return a;
} // 6242d1
VI supersetMobius(VI a) {
    int n = a.size();
    for (int j = 1; j < n; j <= 1)
        for (int i = 1; i < n; i++)
            if (i & j) a[i ^ j] -= a[i];
    return a;
} // f044b4

```

Árvore de Li Chao

Suporta dois tipos de operações:

- *insert(line)*: adiciona a reta $line(y = ax + b)$ ao conjunto de retas. Complexidade: $O(\log(N))$.
- *query(X)*: encontra a reta do conjunto que minimiza o valor $aX + b$. Complexidade: $O(\log(N))$.

Onde N é o tamanho do intervalo em que as retas atuam.

f40448 - dsa/li-chao.cpp - 29 lines

```

struct Line {
    i64 a, b;
    i64 operator()(int x) { return a * x + b; }
};
struct LiChao {
    vector<Line> t;
    int n;
    LiChao(int N): t(4 * N, {0, LLONG_MAX}), n(N) {} // Replace with
    ↪ LLONG_MIN in case of finding max

```

```

void insert(Line line) { insert(line, 0, n, 1); }
void insert(Line line, int tl, int tr, int i) {
    int tm = tl + (tr - tl) / 2;
    if (line(tm) < t[i](tm)) swap(line, t[i]); // Replace with > in case
    ↪ of finding max
    if (tl + 1 == tr) return;
    if (t[i](tl) < line(tl)) insert(line, tm, tr, i << 1 | 1); // Replace
    ↪ with > in case of finding max
    else insert(line, tl, tm, i << 1);
} // f784a4
Line query(int x) {
    Line ans = {0, LLONG_MAX}; // Replace with LLONG_MIN in case of
    ↪ finding max
    query(x, ans, 0, n, 1);
    return ans;
} // 2aca6b
void query(int x, Line &line, int tl, int tr, int i) {
    if (t[i](x) < line(x)) line = t[i]; // Replace with > in case of
    ↪ finding max
    int tm = tl + (tr - tl) / 2;
    if (tl + 1 == tr) return;
    if (x < tm) query(x, line, tl, tm, i << 1);
    else query(x, line, tm, tr, i << 1 | 1);
} // 973d96
};

```

Árvore de Li Chao++

Suporta dois tipos de operações:

- *insert(line, L, R)*: adiciona a reta $line(y = ax + b)$ ao conjunto de retas, atuando no intervalo L, R . Complexidade: $O(\log^2(R - L))$.
- *query(X)*: encontra a reta do conjunto que minimiza o valor $aX + b$. Complexidade: $O(\log(L))$.

7a833b - dsa/li-chao-extended.cpp - 36 lines

```

struct Line {
    i64 a, b;
    i64 operator()(int x) { return a * x + b; }

```



```

};
struct LiChao {
    vector<Line> t;
    int n;
    LiChao(int N): t(4 * N, {0, LLONG_MAX}), n(N) {} // Replace with
    ↪ LLONG_MIN in case of finding max
    void insert(Line line, int L, int R) { insert(line, L, R, 0, n, 1); }
    void insert(Line line, int L, int R, int tl, int tr, int i) {
        int tm = tl + (tr - tl) / 2;
        if (L >= tr || R <= tl) return;
        if (L <= tl && tr <= R) {
            if (line(tm) < t[i](tm)) swap(line, t[i]); // Replace with > in
            ↪ case of finding max
            if (tl + 1 == tr) return;
            if (t[i](tl) < line(tl)) insert(line, L, R, tm, tr, i << 1 | 1);
            ↪ // Replace with > in case of finding max
            else insert(line, L, R, tl, tm, i << 1);
        } else {
            if (tl + 1 == tr) return;
            insert(line, L, R, tm, tr, i << 1 | 1);
            insert(line, L, R, tl, tm, i << 1);
        }
    }
} // 0b7e08
Line query(int x) {
    Line ans = {0, LLONG_MAX}; // Replace with LLONG_MIN in case of
    ↪ finding max
    query(x, ans, 0, n, 1);
    return ans;
} // 2aca6b
void query(int x, Line &line, int tl, int tr, int i) {
    if (t[i](x) < line(x)) line = t[i]; // Replace with > in case of
    ↪ finding max
    int tm = tl + (tr - tl) / 2;
    if (tl + 1 == tr) return;
    if (x < tm) query(x, line, tl, tm, i << 1);
    else query(x, line, tm, tr, i << 1 | 1);
} // 973d96
};

```

Árvore Cartesiana

Constrói uma árvore cartesiana dos inteiros $0, 1, \dots, n-1$ a partir de um critério de comparação. A árvore é retornada no formato de um array *par* de tal forma que *par_i* é o pai do nó *i* na árvore. A raiz da árvore é o nó mínimo de acordo com o critério de comparação. Complexidade: $O(n)$.

bb7763 - dsa/cartesian-tree.cpp - 20 lines

```

VI cartTree(int n, auto le) {
    int last;
    VI st, par(n);
    for (int i = 0; i < n; i++){
        par[i] = i, last = -1;
        while (st.size() && le(par[i], st.back())) {
            if (last != -1) par[last] = st.back();
            last = st.back();
            st.pop_back();
        }
        if (last != -1) par[last] = i;
        st.push_back(i);
    }
    while (st.size() > 1) {
        last = st.back();
        st.pop_back();
        par[last] = st.back();
    }
    return par;
} // bb7763

```

Convolução

Calcula a convolução de dois arrays em $O(n \log(n))$. A convolução de dois arrays *a* e *b* é definida como um array *c* tal que:

$$c_k = \sum_{i+j=k} a_i b_j$$

FFT

bd675d - dsa/fft.cpp - 43 lines

```
using ld = double;
const long double PI = numbers::pi;
using VD = vector<ld>;
using C = complex<ld>;

void fft(vector<C> &a) {
    int n = a.size(), L = 31 - __builtin_clz(n);
    static vector<complex<long double>> R(2, 1);
    static vector<C> rt(2, 1);
    for (static int k = 2; k < n; k <= 1) {
        R.resize(n); rt.resize(n);
        auto x = polar(1.0L, PI / k);
        for (int i = k; i < 2 * k; i++)
            rt[i] = R[i] = i & 1 ? R[i / 2] * x : R[i / 2];
    }
    VI rev(n);
    for (int i = 0; i < n; i++) rev[i] = (rev[i / 2] | (i & 1) << L) / 2;
    for (int i = 0; i < n; i++) if (i < rev[i]) swap(a[i], a[rev[i]]);
    for (int k = 1; k < n; k <= 1)
        for (int i = 0; i < n; i += k << 1)
            for (int j = 0; j < k; j++) {
                auto x = (ld *)&rt[j + k], y = (ld *)&a[i + j + k];
                C z(x[0] * y[0] - x[1] * y[1], x[0] * y[1] + x[1] * y[0]);
                a[i + j + k] = a[i + j] - z;
                a[i + j] += z;
            }
} // ed7a4e

VD conv(VD &a, VD &b) {
    if (a.empty() || b.empty())
        return {};
    VD res(ssize(a) + ssize(b) - 1);
    int L = 32 - __builtin_clz(res.size()), n = 1 << L;
    vector<C> in(n), out(n);
    copy(a.begin(), a.end(), in.begin());
    for (int i = 0; i < b.size(); i++) in[i].imag(b[i]);
    fft(in);
```

```
    for (C &x : in) x *= x;
    for (int i = 0; i < n; i++) out[i] = in[-i & (n - 1)] - conj(in[i]);
    fft(out);
    for (int i = 0; i < res.size(); i++) res[i] = imag(out[i]) / (4 * n);
    return res;
} // fb4703
```

NTT

a359a7 - dsa/ntt.cpp - 53 lines

```
using VL = vector<i64>;

const int MOD = 998244353, root = 62;

i64 powm(i64 x, i64 e) {
    i64 r = 1;
    while (e) {
        if (e & 1) (r *= x) %= MOD;
        e >>= 1;
        (x *= x) %= MOD;
    }
    return r;
} // ae0a09

void ntt(VL &a) {
    int n = a.size(), k = bit_width(unsigned(n)) - 1;
    static array<i64, 1 << 23> r = {1, 1};
    for (static int t = 2, s = 2; t < n; t <= 1, s++) {
        array z = {111, powm(root, MOD >> s)};
        for (int i = t; i < t << 1; i++) r[i] = r[i >> 1] * z[i & 1] % MOD;
    }
    static array<i64, 1 << 23> b, c;
    copy(a.begin(), a.end(), b.begin());
    for (int m = n; m >= 1; m >>= 1) {
        for (int l = 0; l < n; l += m << 1)
            for (int i = 0; i < m; i++)
                c[l + i] = b[l + 2 * i], c[l + m + i] = b[l + 2 * i + 1];
        copy(c.begin(), c.begin() + n, b.begin());
    }
```

```

for (int m = 1; m < n; m <= 1) {
    for (int l = 0; l < n; l += m < 1)
        for (int i = 0; i < m; i++) {
            i64 z = r[m + i] * b[l + m + i] % MOD;
            c[l + i] = b[l + i] + z;
            c[l + m + i] = b[l + i] + (MOD - z);
        }
    for (int i = 0; i < n; i++) b[i] = c[i] < MOD ? c[i] : c[i] - MOD;
}
copy(b.begin(), b.begin() + n, a.begin());
} // 39c196

VL conv(VL &a, VL &b) {
    if (a.empty() || b.empty()) return {};
    int s = ssize(a) + ssize(b) - 1, B = 32 - __builtin_clz(s), n = 1 << B;
    int inv = powm(n, MOD - 2);
    VL L(a), R(b), out(n);
    L.resize(n), R.resize(n);
    ntt(L), ntt(R);
    for (int i = 0; i < n; ++i)
        out[-i & (n - 1)] = (i64)L[i] * R[i] % MOD * inv % MOD;
    ntt(out);
    return {out.begin(), out.begin() + s};
} // c28e2c

```

Grafos

Caminho Euleriano

Encontra um caminho euleriano em um grafo, se existir. Recebe uma lista de adjacência da forma $[x, e]$, onde x é o vizinho e e é o id da aresta. Para grafos não direcionados, basta usar o mesmo id para arestas indo e voltando. Complexidade: $O(V + E)$.

4f6be1 - graph/eulerian-walk.cpp - 14 lines

```

pair<VI, VI> eulerWalk(int n, int m, VVII &adj, int s = 0) {
    VI path, walk, d(n), vis(m), ne(n);
    VII st = {{s, -1}};

```

```

d[s]++; // Remove to force cycles
while (st.size()) {
    auto [x, ex] = st.back();
    int &ei = ne[x], end = adj[x].size(), y, e;
    if (ei == end) { path.push_back(x); walk.push_back(ex);
        ↪ st.pop_back(); continue; }
    tie(y, e) = adj[x][ei++];
    if (!vis[e]) { d[x]--; d[y]++; vis[e] = 1; st.emplace_back(y, e); }
}
for (int x: d) if (x < 0 || walk.size() != m) return {{}, {}};
return {{path.rbegin(), path.rend()}, {walk.rbegin() + 1, walk.rend()}};
} // 4f6be1

```

Max Flow

Dinic

Encontra o fluxo máximo de s a t dado um grafo com arestas e capacidades. Complexidade: $O(N^2M)$.

- $addEdge(from, to, capacity)$: adiciona uma aresta ao grafo de $from$ para to com capacidade $capacity$.
- $flow(s, t)$: encontra o fluxo máximo de s para t .
- $flow(s, t, max)$: o mesmo que a última, mas limita o fluxo a max .

Nota: A complexidade também é limitada pelo valor do fluxo máximo, se o fluxo máximo é F , então a complexidade é menor ou igual a $O(FM)$.

1cb7ae - graph/dinic.cpp - 55 lines

```

struct Dinic {
    struct Edge {
        i64 s, t, f, c, r;
    };
    vector<Edge> e;
    VVI adj;
    VI ne, lvl;
    Dinic(int n): adj(n), ne(n), lvl(n) {}
    void addEdge(int a, int b, i64 c) {
        adj[a].push_back(e.size());

```

```

e.emplace_back(a, b, 0, c, e.size() + 1);
adj[b].push_back(e.size());
e.emplace_back(b, a, 0, 0, e.size() - 2);
} // 5c5805
bool bfs(int s, int t) {
    queue<int> q;
    q.push(s);
    fill(lvl.begin(), lvl.end(), -1);
    lvl[s] = 0;
    while (q.size()) {
        int x = q.front();
        q.pop();
        for (int i: adj[x]) {
            if (e[i].f == e[i].c) continue;
            if (lvl[e[i].t] != -1) continue;
            lvl[e[i].t] = lvl[x] + 1;
            q.push(e[i].t);
        }
    }
    return lvl[t] == -1;
} // 4feb4a
i64 dfs(int x, int t, i64 f) {
    if (f == 0) return 0;
    if (x == t) return f;
    for (int &te = ne[x]; te < adj[x].size(); te++) {
        int i = adj[x][te], y = e[i].t;
        if (lvl[y] != lvl[x] + 1) continue;
        if (i64 df = dfs(y, t, min(f, e[i].c - e[i].f))) {
            e[i].f += df;
            e[i ^ 1].f -= df;
            return df;
        }
    }
    return 0;
} // 9b6f3b
i64 flow(int s, int t, i64 mx = LLONG_MAX) {
    i64 mf = 0;
    while (true) {
        if (bfs(s, t)) break;
        fill(ne.begin(), ne.end(), 0);
        while (i64 f = dfs(s, t, mx - mf)) mf += f;
    }
}

```

```

    }
    return mf;
} // 539629
};

```

Min Cost Flow

Árvores

Base

Inclui todas as funcionalidades básicas de uma árvore para serem usadas no resto dos templates de árvore. Depois de adicionar todas as arestas, use *build(root)* para construir a árvore com raiz em *root*.

00b4f5 - tree/base.cpp - 25 lines

```

struct Tree {
    VVI adj;
    VI cnt, par, dst, tin, tout, nxt;
    int t;
    Tree(int n): adj(n), cnt(n), par(n), dst(n), tin(n), tout(n), nxt(n),
        ↪ t(0) {}
    void addEdge(int a, int b) { adj[a].push_back(b); adj[b].push_back(a); }
    void build(int r = 0) { build(r, -1); par[r] = r; nxt[r] = r; prepare(r);
        ↪ }
    void build(int x, int pre) {
        cnt[x] = 1;
        for (int &y: adj[x]) if (y != pre) {
            par[y] = x, dst[y] = dst[x] + 1;
            build(y, x);
            cnt[x] += cnt[y];
            if (cnt[y] > cnt[adj[x][0]]) swap(adj[x][0], y);
        }
    } // 6e5d12
    void prepare(int x) {
        tin[x] = t++;
        for (int y: adj[x]) if (y != par[x]) {
            nxt[y] = y == adj[x][0] ? nxt[x] : y;
        }
    }
}

```

```

        prepare(y);
    }
    tout[x] = t;
} // 1b3e55
};

```

Movimento em árvores

Inclui funcionalidades de movimento em árvores:

- $lca(x, y)$ encontra o menor ancestral comum entre x e y . Complexidade: $O(\log(n))$.
- $up(x, d)$ encontra o d -ésimo ancestral de x . Complexidade: $O(\log(n))$.
- $dist(x, y)$ encontra a distância entre os vértices x e y (em número de arestas no caminho). Complexidade: $O(\log(n))$.
- $move(x, y, d)$ encontra o d -ésimo vértice no menor caminho de x a y . Complexidade: $O(\log(n))$.

53a574 - tree/tree-move.cpp - 32 lines

```

struct TreeMove {
    int n;
    VI w, tm;
    Tree &t;
    TreeMove(Tree &t): n(t.adj.size()), w(n), tm(n), t(t) {
        for (int i = 0; i < n; i++) {
            w[t.tin[i]] = i;
            tm[i] = t.tin[i];
        }
    }
    int lca(int x, int y) {
        while (t.nxt[x] != t.nxt[y]) {
            if (t.dst[t.nxt[x]] > t.dst[t.nxt[y]]) x = t.par[t.nxt[x]];
            else y = t.par[t.nxt[y]];
        }
        return w[min(tm[x], tm[y])];
    } // 74d125
    int up(int x, int d) {
        while (t.dst[x] - t.dst[t.nxt[x]] < d) {
            d -= t.dst[x] - t.dst[t.nxt[x]] + 1;

```

```

        x = t.par[t.nxt[x]];
    }
    return w[tm[x] - d];
} // 3fb308
int move(int x, int y, int d) {
    int l = lca(x, y), ld = t.dst[l], dx = t.dst[x] - ld, dy = t.dst[y] -
        ld;
    return d > dx ? up(y, dx + dy - d) : up(x, d);
} // ea0e64
int dist(int x, int y) {
    return t.dst[x] + t.dst[y] - 2 * t.dst[lca(x, y)];
} // 0a8d1a
};

```

HLD

Obtém a decomposição do caminho de x a y em partes do array de dfs. É garantido que vão existir no máximo $O(\log(n))$ partes. O argumento *ord* serve para indicar que é preciso diferenciar caminhos subindo e descendo (útil quando a ordem dos vértices no caminho importa).

ac6fba - tree/hld-path.cpp - 19 lines

```

VII hldPath(Tree &t, int x, int y, bool ord = false) {
    VII p, rp;
    while (t.nxt[x] != t.nxt[y]) {
        if (t.dst[t.nxt[x]] < t.dst[t.nxt[y]]) {
            rp.emplace_back(t.tin[t.nxt[y]], t.tin[y]);
            y = t.par[t.nxt[y]];
        } else {
            if (ord) p.emplace_back(t.tin[x], t.tin[t.nxt[x]]);
            else p.emplace_back(t.tin[t.nxt[x]], t.tin[x]);
            x = t.par[t.nxt[x]];
        }
    }
    if (t.dst[x] <= t.dst[y] || ord)
        p.emplace_back(t.tin[x], t.tin[y]);
    else p.emplace_back(t.tin[y], t.tin[x]);
    reverse(rp.begin(), rp.end());

```

```

    p.insert(p.begin(), rp.begin(), rp.end());
    return p;
} // ac6fba

```

LCA

Permite calcular o LCA de dois vértices em $O(1)$.

2539ab - tree/lca.cpp - 23 lines

```

using RMQ = SparseTable<int, [](int a, int b) { return min(a, b); }>;

struct LCA {
    int t = 0;
    VI tm, path, ret;
    RMQ rmq;
    LCA(VVI &adj, int r = 0) : tm(adj.size()) {
        dfs(adj, r, -1);
        rmq.build(ret);
    }
    void dfs(VVI &adj, int x, int pre) {
        tm[x] = t++;
        for (int y: adj[x]) if (y != pre) {
            path.push_back(x), ret.push_back(tm[x]);
            dfs(adj, y, x);
        }
    }
} // c5a5ad

int lca(int x, int y) {
    if (x == y) return x;
    tie(x, y) = minmax(tm[x], tm[y]);
    return path[rmq.query(x, y)];
} // e8105a
};

```

Strings

Suffix Array

Dada uma string S , ordena todos os sufixos em ordem lexicográfica. Retorna um array SA tal que

SA_i = índice do começo do i -ésimo sufixo em ordem lexicográfica

Complexidade: $O(n \log(n))$, onde n é o tamanho da string.

950818 - strings/suffix-array.cpp - 28 lines

```

VI sufArr(string &s) { // Could also be vector<T> &s
    int n = s.size();
    VI c(n), d(n), e(n), sb(n), sa(n), cnt(n + 1);
    iota(sa.begin(), sa.end(), 0);
    sort(sa.begin(), sa.end(), [&](int i, int j) { return s[i] < s[j]; });
    c[sa[0]] = 1;
    for (int i = 1; i < n; i++)
        c[sa[i]] = c[sa[i - 1]] + (s[sa[i]] != s[sa[i - 1]]);
    for (int k = 1; c[sa[n - 1]] != n; k <= 1) {
        for (int i = 0; i < n; i++) d[i] = i + k < n ? c[i + k] : 0;
        auto srt = [&](auto &C) {
            fill(cnt.begin(), cnt.end(), 0);
            for (int i = 0; i < n; i++) cnt[C[i] + 1]++;
            for (int i = 1; i <= n; i++) cnt[i] += cnt[i - 1];
            for (int i: sa) sb[cnt[C[i]]++] = i;
            swap(sa, sb);
        };
        srt(d); srt(c);
        int lc = 0, ld = 0, le = 0;
        for (int i: sa) {
            if (c[i] != lc || d[i] != ld) e[i] = le + 1;
            else e[i] = le;
            lc = c[i], ld = d[i], le = e[i];
        }
        swap(c, e);
    }
    return sa;
}

```

```
} // 950818
```

Maior prefixo comum (LCP)

Dada uma string e o suffix array dela, calcula para cada duas strings consecutivas no suffix array, qual o maior prefixo comum delas. Mais formalmente, calcula um array *LCP* tal que

$$LCP_i = \text{maior prefixo comum dos sufixos } SA_i \text{ e } SA_{i+1}$$

Complexidade: $O(n)$. Onde n é o tamanho da string.

f8cd22 - strings/lcp.cpp - 16 lines

```
VI lcpArr(string &s, VI &sa) {
    int n = s.size(), k = 0;
    VI r(n), lcp(n);
    for (int i = 0; i < n; i++) r[sa[i]] = i;
    for (int i = 0; i < n; i++) {
        if (r[i] == n - 1) {
            k = 0;
            continue;
        }
        int j = sa[r[i] + 1];
        while (i + k < n && j + k < n && s[i + k] == s[j + k]) k++;
        lcp[r[i]] = k;
        if (k) k--;
    }
    return lcp;
} // f8cd22
```

KMP

Dada uma string S , calcula um array *pre* de forma que

$$pre_i = \text{maior valor de } k < i \text{ tal que } S_0 S_1 \cdots S_{k-1} = S_{i-k+1} S_{i-k+2} \cdots S_i$$

Complexidade: $O(n)$.

568171 - strings/kmp.cpp - 10 lines

```
VI kmp(string s) {
    int n = s.size(), len = 0;
    VI pre(n);
    for (int i = 1; i < n; i++) {
        while (len > 0 && s[i] != s[len]) len = pre[len - 1];
        if (s[i] == s[len]) len++;
        pre[i] = len;
    }
    return pre;
} // 568171
```

Geometria

Base

Base para usar em outros templates de geometria.

440b05 - geometry/prelude.cpp - 9 lines

```
using D = i64;
using point = complex<D>;

D dot(point a, point b) {
    return real(conj(a) * b);
} // c6b21e
D cross(point a, point b) {
    return imag(conj(a) * b);
} // a8ce1d
```

Interseção de retas

Encontra o ponto de interseção de duas retas.

cc6127 - geometry/line-intersection.cpp - 5 lines

```
// D = double;
```

```
point lineIntersect(point a1, point d1, point a2, point d2) {
    return a1 + cross(a2 - a1, d2) / cross(d1, d2) * d1;
} // cc6127
```

Interseção de segmentos

Verifica se dois segmentos de reta se intersectam.

3b079c - geometry/segment-intersection.cpp - 10 lines

```
#define CK(a, b, t) max(t(a[0]), t(a[1])) < min(t(b[0]), t(b[1]))
#define CKK(a, b, t) (s(a[0], a[1], b[t]) == 0 && dot(b[t] - a[0], b[t] -
↪ a[1]) <= 0)

bool segInterChk(point a, point b, point c, point d) {
    array p = {a, b}, q = {c, d};
    auto s = [](point p, point q, point r) { return cross(q - p, r - p); };
    if (CK(p, q, real) || CK(p, q, imag) || CK(q, p, real) || CK(q, p, imag))
        ↪ return false;
    if (CKK(p, q, 0) || CKK(p, q, 1) || CKK(q, p, 0) || CKK(q, p, 1)) return
        ↪ true;
    return (s(a, b, c) > 0) != (s(a, b, d) > 0) && (s(c, d, a) > 0) != (s(c,
        ↪ d, b) > 0);
} // c22bf9
```

Interseção de círculo e reta

Interseção de círculos

Tangentes comuns a dois círculos

Área de um polígono

Calcula a área de um polígono dada a lista de vértices. A área é positiva se os vértices estão no sentido anti-horário, e negativa caso contrário. Complexidade: $O(n)$.

aa075b - geometry/polygon-area.cpp - 6 lines

```
D polyArea(vector<point> &pt) {
    int n = pt.size();
    D s = 0;
    for (int i = 2; i < n; i++) s += cross(pt[i] - pt[0], pt[i] - pt[0]);
    return s;
} // aa075b
```

Verificar se um ponto está dentro de um polígono

Verifica se um ponto está dentro de um polígono convexo. Use *border* = false para não incluir pontos na borda do polígono. Complexidade: $O(\log(n))$.

b44cc8 - geometry/point-in-polygon.cpp - 10 lines

```
bool pointInPoly(vector<point> &pt, point q, bool border = true) {
    int n = pt.size(), t = border, l = 1, r = n - 1;
    auto v = [&](int i) { return cross(q - pt[0], pt[i] - pt[0]); };
    if (v(l) >= t || v(r) <= -t) return 0;
    while (r - l > 1) {
        int m = (l + r) / 2;
        (v(m) > 0 ? r : l) = m;
    }
    return cross(pt[r] - pt[l], pt[l] - q) < t;
} // b44cc8
```

Soma Minkowski de polígonos convexos

Convex hull

Encontra o convex hull de um conjunto de pontos. O resultado é uma lista de índices dos pontos do convex hull no sentido anti-horário. Complexidade: $O(n \log(n))$.

4737ad - geometry/convex-hull.cpp - 24 lines

```
vector<int> convHull(vector<point> &pt) {
    int n = pt.size(), m;
    VI h, ord(n);
```



```

auto add = [&]() {
    VI st;
    for (int i: ord) {
        while ((m = st.size()) > 1) {
            point a = pt[st[m - 1]], b = pt[st[m - 2]], c = pt[i];
            if (cross(b - a, c - a) < 0) break; // > for clockwise, <= to
            ↪ include non-vertices
            st.pop_back();
        }
        st.push_back(i);
    }
    st.pop_back();
    h.insert(h.end(), st.begin(), st.end());
};
iota(ord.begin(), ord.end(), 0);
auto top = [] (auto a) { return make_pair(real(a), imag(a)); };
sort(ord.begin(), ord.end(), [&](int i, int j) { return top(pt[i]) >
    ↪ top(pt[j]); });
add();
reverse(ord.begin(), ord.end());
add();
return h;
} // 4737ad

```

Convex hull online

Menor distância euclideana

Corte de Polígono

Fórmulas Úteis

Contagem e Probabilidade

Número de soluções da equação $x_1 + x_2 + \dots + x_k = n$ onde $x_i \geq 0$:

$$\binom{n+k-1}{k-1}$$

Número de soluções da equação $x_1 + x_2 + \dots + x_k = n$ onde $x_i \geq 1$:

$$\binom{n-1}{k-1}$$

Valor esperado do número de tentativas até sucesso com probabilidade p de sucesso:

$$\frac{1}{p}$$

Números de Stirling do primeiro tipo:

$c(n, k)$ = Número de permutações de $\{1, 2, \dots, n\}$ com k ciclos

$$s(n, k) = (-1)^{n-k} c(n, k)$$

$$x(x-1)(x-2)\dots(x-n+1) = s(n, 1)x + s(n, 2)x^2 + \dots + s(n, n)x^n$$

$$c(n+1, k) = n \times c(n, k) + c(n, k-1), k > 0$$

$$s(n+1, k) = -n \times s(n, k) + s(n, k-1), k > 0$$

Somas

$$\sum_{k=0}^n \binom{n}{k} = 2^n$$

$$\sum_{k=1}^n k \binom{n}{k} = n 2^{n-1}$$

$$c(n, n-k) = \sum_{0 \leq i_1 < i_2 < \dots < i_k < n} i_1 i_2 \dots i_k$$

$$\sum_{k=0}^n c(n, k) x^k = n! \binom{n+x-1}{x-1}, x > 0$$

$$\sum_{k=m}^n c(n, k) \binom{k}{m} = c(n+1, m+1)$$

$$\sum_{k=m}^n c(n+1, k+1) \binom{k}{m} (-1)^{m-k} = c(n, m)$$

$$\sum_{k=m}^n c(k, m) \frac{n!}{k!} = c(n+1, m+1)$$

$$\sum_{k=0}^n (m+k) c(m+k, k) = c(n+m+1, n)$$

$$\sum_{k=1}^n x^k = 1 + x + \dots + x^{n-1} = \frac{x^n - 1}{x - 1}$$

$$\sum_{k=1}^n kx^{k-1} = 1 + 2x + 3x^2 + \dots + nx^{n-1} = \frac{nx^{n+1} - ((n+1)x^n) + 1}{(x-1)^2}, x \neq 1$$

$$\sum_{k=1}^n k = \frac{n(n+1)}{2}$$

$$\sum_{k=1}^n k^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{k=1}^n k^3 = \frac{n^2(n+1)^2}{4}$$

Teoremas e Fatos

Teorema de Erdos-Szekeres: Toda sequência de $(r-1)(s-1)+1$ inteiros distintos possui uma subsequência crescente de tamanho r ou uma subsequência decrescente de tamanho s .

O valor esperado é linear:

$$\mathbb{E}(aX + bY) = a\mathbb{E}(X) + b\mathbb{E}(Y)$$

Teorema EGZ: Dados $2n-1$ inteiros $a_1, a_2, \dots, a_{2n-1}$, existem n índices i_1, i_2, \dots, i_n tais que

$$a_{i_1} + a_{i_2} + \dots + a_{i_n} = 0 \pmod{n}$$

Teorema de Lucas: Se $m = \overline{m_k m_{k-1} \dots m_1}$ e $n = \overline{n_k n_{k-1} \dots n_1}$ na base p , então

$$\binom{m}{n} = \binom{m_k}{n_k} \binom{m_{k-1}}{n_{k-1}} \dots \binom{m_1}{n_1} \pmod{p}$$

Teorema de Kummer: For a prime p ,

$$v_p\left(\binom{a+b}{a}\right) = \text{number of carries of } a+b \text{ in base } p$$

Ideias

Gerais

- Em problemas envolvendo somas em intervalos de arrays, pode ser mais fácil trabalhar com o array de somas de prefixos.
- Muitas DPs de bitmask podem ser simplificadas para usar apenas alguma propriedade da bitmask. (Número de bits, etc)
- Se o problema descreve um processo que parece complicado, talvez seja mais fácil ver o processo ao contrário.

Calculando somas e integrais

- Verificar se existe uma fórmula fechada
- Inverter a ordem da soma (em somatórios duplos)
- Transformar em somatório duplo e inverter a ordem
- Verificar se a soma pode ser calculada de um jeito eficiente (sem ser uma fórmula fechada)
- No caso de calcular várias somas, verificar se existe uma relação entre elas (Calcular uma pode ajudar a calcular outra)

Problemas de Otimização

- Encontrar um upper bound e provar que ele sempre é alcançável.
- Encontrar uma construção boa, e provar que não é possível fazer melhor.
- Escrever um brute-force e testar com casos pequenos para encontrar algum padrão.

Debugging

Geral

- Trocou nomes de variáveis? (m por n , j por i , ...)
- Esqueceu algum caso especial? ($n = 1$, $m = 0$, ...)
- Esqueceu de trocar entre 0-based e 1-based?

WA - Wrong Answer

- Esqueceu o `define int i64`?
- Esqueceu algum caso base?
- verificar se o valor inicial da DP está correto.

RE - Runtime Error

- Verificar asserts.
- Verificar limites de vectors.
- Verificar se `maxn` é o suficiente.