Contents

1 Template **2**

2 Search **3**
2.1 Ternary Search 3

3 Data Structures **3**
3.1 Segment Tree 3

4 Sequences **4**
4.1 Max/Min subsegment 4
4.1.1 Max/Min submatrix 5

5 Algebra **5**
5.1 All divisors 5
5.2 Primality test 5
5.3 Binary exponentiation 6
5.4 Greatest common divisor 6
5.4.1 Least common multiple 6
5.4.2 Extended Euclides Algorithm 7
5.5 Linear Diophantine Equations 7
5.5.1 Any solution 7
5.6 Integer Factorization 7
5.6.1 Pollard's Rho 7
5.7 Fast Fourier Transform 9
5.7.1 Polynomial Multiplication 9

6 Graphs **10**
6.1 DFS 10
6.2 BFS 10
6.2.1 Shortest path on unweighted graph 11
6.3 Flood Fill 12
6.4 Topological Sort (Directed Acyclic Graph) 12
6.4.1 DFS Variation 12
6.4.2 Kahn's Algorithm 13
6.5 Bipartite Graph Check (Undirected Graph) 13
6.6 Cycle Check (Directed Graph) 14
6.7 Dijkstra 15

7 Dynamic Programming **15**
7.1 Coin Change 15
7.1.1 Canonicality check 16
7.2 Knapsack 17
7.3 LIS 18
7.4 Travelling Salesman Problem 19

8 Math Formulas **20**
8.1 Sum of an arithmetic progression . . . 20
8.2 Permutation with repeated elements . . . 20
8.3 Check if is geometric progression . . . 20
8.4 Bitwise equations . . . 20
8.5 Cube of Binomial . . . 20
8.5.1 Sum of Cubes . . . 20
8.5.2 Difference of Cubes . . . 20
8.6 Binomial expansion . . . 21

9 Facts **21**
9.1 XOR . . . 21
9.1.1 Self-inverse property . . . 21
9.1.2 Identity element . . . 21
9.1.3 Commutative . . . 21
9.1.4 Associative . . . 21

1 Template

```
#include <bits/stdc++.h>
using namespace std;

#define dedinhos cin.tie(0)->sync_with_stdio(0)
using ll = long long;
#define vll vector<ll>
#define vvll vector<vll>
#define pll pair<ll, ll>
#define vpll vector<pll>
#define vvpll vector<vpll>
#define endl '\n'
#define all(xs) xs.begin(), xs.end()
#define found(x, xs) (xs.find(x) != xs.end())
#define rep(i, a, b) for(ll i = (a); i < (ll)(b); ++i)
#define per(i, a, b) for(ll i = (a); i >= (ll)(b); --i)
#define eb emplace_back

signed main() {


    return 0;
}
```

2 Search

2.1 Ternary Search

$O(\log n)$

Function f(x) is unimodal on an interval [l, r]. Unimodal means: the function strictly increases first, reaches a maximum, and then strictly decreases OR the function strictly decreases first, reaches a minimum and then strictly decreases

```
double ternary_search(double l, double r) {
    double eps = 1e-9; // error limit
    while(r - l > eps) {
        double m1 = l + (r-l) / 3;
        double m2 = r - (r-l) / 3;

        double f1 = f(m1);
        double f2 = f(m2);

        if(f1 < f2)
            l = m1;
        else
            r = m2;
    }

    return f(l);
}
```

3 Data Structures

3.1 Segment Tree

$O(n \log n)$

Maximum value variation.

```
/**
 * Author: Lucian Bicsi
 * Date: 2017-10-31
 * License: CC0
 * Source: folklore
 * Description: Zero-indexed max-tree. Bounds are inclusive
    to the left and exclusive to the right.
 * Can be changed by modifying T, f and unit.
 * Status: stress-tested
 */

struct Tree {
    typedef int T;
    static constexpr T unit = INT_MIN;
```

```
    T f(const T &a,const T &b) { return max(a, b); } // (any
        associative fn)
    vector<T> s; int n;
    Tree(int n = 0, T def = unit) : s(2*n, def), n(n) {}
    void update(int pos, T val) {
        for (s[pos += n] = val; pos /= 2;)
            s[pos] = f(s[pos * 2], s[pos * 2 + 1]);
    }
    T query(int b, int e) { // query [b, e)
        T ra = unit, rb = unit;
        for (b += n, e += n; b < e; b /= 2, e /= 2) {
            if (b % 2) ra = f(ra, s[b++]);
            if (e % 2) rb = f(s[--e], rb);
        }
        return f(ra, rb);
    }
};
```

4 Sequences

4.1 Max/Min subsegment

$O(n)$

```

ll kadane(const vll &a) {
    ll n = a.size();
    ll ans = a[0], ans_l = 0, ans_r = 0;
    ll sum = 0, minus_pos = -1;

    for (ll r = 0; r < n; ++r) {
        sum += a[r];
        if (sum > ans) {
            ans = sum;
            ans_l = minus_pos + 1;
            ans_r = r;
        }
        if (sum < 0) {
            sum = 0;
            minus_pos = r;
        }
    }

    return ans;
}
```

4.1.1 Max/Min submatrix

$O(nm^2)$

```
ll MSR(ll n, ll m, const vvll &a) {
    ll ans = -LLONG_MAX;

    for(ll i=0; i<m; i++) {
        vll r(n+1, 0);

        for(ll j=i; j<m; j++) {
            for(ll k=0; k<n; k++)
                r[k] += a[k][j];

            ans = max(ans, kadane(n, r));
        }
    }

    return ans;
}
```

5 Algebra

5.1 All divisors

$O(\sqrt{n})$

```
vll divisors(ll n) {
  vll divs;
  for (ll i = 1; 1LL * i * i <= n; i++) {
    if (n % i == 0) {
      divs.push_back(i);
      if (i != n / i) {
        divs.push_back(n / i);
      }
      }
    }

  return divs;
}
```

5.2 Primality test

$O(\sqrt{n})$

```
bool isPrime(ll n)
{
    if(n!=2 && n % 2==0)
        return false;
```

```

    for(ll d=3; d*d <= n; d+=2)
    {
        if(n % d==0)
            return false;
    }

    return n >= 2;
}
```

5.3 Binary exponentiation

$O(\log n)$

```
ll binpow(ll a, ll b) {
    ll res = 1;
    while (b > 0) {
        if (b & 1)
            res = res * a;
        a = a * a;
        b >>= 1;
    }
    return res;
}
```

5.4 Greatest common divisor

$O(\log \min(a, b))$

```
ll gcd (ll a, ll b) {
  while (b) {
      a %= b;
      swap(a, b);
  }
  return a;
}
```

5.4.1 Least common multiple

```
ll lcm(ll a, ll b) {
    return a / gcd(a, b) * b;
  }
```

5.4.2 Extended Euclides Algorithm

```
ll gcd(ll a, ll b, ll& x, ll& y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    ll x1, y1;
    ll d = gcd(b, a % b, x1, y1);
    x = y1;
    y = x1 - y1 * (a / b);
    return d;
}
```

5.5 Linear Diophantine Equations

$O(\log \min(a, b))$

5.5.1 Any solution

```
bool find_any_solution(ll a, ll b, ll c, ll &x0, ll &y0, ll
   &g) {
    g = gcd(abs(a), abs(b), x0, y0);
    if (c % g) {
        return false;
    }

    x0 *= c / g;
    y0 *= c / g;
    if (a < 0) x0 = -x0;
    if (b < 0) y0 = -y0;
    return true;
}
```

5.6 Integer Factorization

5.6.1 Pollard's Rho

$O(\sqrt[4]{n} \log n)$

```
/**
 *  @param a first multiplier
 *  @param b second multiplier
 *  @param mod
 *  @return a * b mod n (without overflow)
 *  @brief Multiplies two numbers >= 10^18
 *  Time Complexity: O(log b)
```

```
*/
ll mult(ll a, ll b, ll mod) {
    ll result = 0;
    while (b) {
        if (b & 1)
            result = (result + a) % mod;
        a = (a + a) % mod;
        b >>= 1;
    }
    return result;
}

/**
 *  @param x first multiplier
 *  @param c second multiplier
 *  @param mod
 *  @return f(x) = x^2 + c mod (mod)
 *  @brief Polynomial function chosen for pollard's rho
 *  Time Complexity: O(1)
*/
ll f(ll x, ll c, ll mod) {
    return (mult(x, x, mod) + c) % mod;
}

/**
 *  @param n number that we want to find a factor p
 *  @param x0 number where we will start
 *  @param c constant in polynomial function
 *  @return fac
 *  @brief Pollard's Rho algorithm (works only for composite
    numbers)
 *  if(g==n) try other starting values
 *  Time Complexity: O(n^(1/4) log n)
*/
ll rho(ll n, ll x0=2, ll c=1) {
    ll x = x0;
    ll y = x0;
    ll g = 1;
    while (g == 1) {
        x = f(x, c, n);
        y = f(y, c, n);
        y = f(y, c, n);
        g = gcd(abs(x - y), n);
    }
    return g;
}
```

5.7 Fast Fourier Transform

$O(n \log n)$

```
using cd = complex<double>;
const double PI = acos(-1);

/**
 *  @param a vector that we want to transform
 *  @param invert inverse fft or not
 *  @brief apply fft or inverse fft to a vector
 *  Time Complexity: O(n log n)
*/
void fft(vector<cd> &a, bool invert) {
    ll n = a.size();
    if (n == 1)
        return;

    vector<cd> a0(n / 2), a1(n / 2);
    for (ll i = 0; 2 * i < n; i++) {
        a0[i] = a[2*i];
        a1[i] = a[2*i+1];
    }
    fft(a0, invert);
    fft(a1, invert);

    double ang = 2 * PI / n * (invert ? -1 : 1);
    cd w(1), wn(cos(ang), sin(ang));
    for (ll i = 0; 2 * i < n; i++) {
        a[i] = a0[i] + w * a1[i];
        a[i + n/2] = a0[i] - w * a1[i];
        if (invert) {
            a[i] /= 2;
            a[i + n/2] /= 2;
        }
        w *= wn;
    }
}
```

5.7.1 Polynomial Multiplication

```
/**
 *  @param a first polynomial coefficients
 *  @param b second polynomial coefficients
 *  @return product of two polynomials
 *  @brief Multiplies two polynomials
 *  Time Complexity: O(n log n)
*/
vll multiply(vll const& a, vll const& b) {
```

```
    vector<cd> fa(a.begin(), a.end()), fb(b.begin(), b.end()
        );
    ll n = 1;
    while (n < a.size() + b.size())
        n <<= 1;
    fa.resize(n);
    fb.resize(n);

    fft(fa, false);
    fft(fb, false);
    for (ll i = 0; i < n; i++)
        fa[i] *= fb[i];
    fft(fa, true);

    vll result(n, 0);
    for (ll i = 0; i < n; i++) {
        result[i] += round(fa[i].real());
        if(result[i] >= 10) {
            result[i+1] += result[i] / 10;
            result[i] %= 10;
        }
    }
    return result;
}
```

6 Graphs

6.1 DFS

$O(n+m)$

```
void dfs(ll at, ll n ,vpll adj[], bool visited[]) {
    if(visited[at])
        return;

    visited[at] = true;

    vpll neighbours = adj[at];
    for(auto nex: neighbours)
        dfs(nex.first, n, adj, visited);
}
```

6.2 BFS

$O(n+m)$

```
void bfs(ll s, ll n, vll adj[]) {
    bool visited[n] = {0};
```

```
    visited[s] = true;

    queue<ll> q;
    q.push(s);
    while (!q.empty())
    {
        vll neighbours = adj[q.front()];
        for(auto nex: neighbours) {
            if(!visited[nex]) {
                visited[nex]=true;
                q.push(nex);
            }
        }
        cout << q.front() << '\n';
        q.pop();
    }
}
```

6.2.1 Shortest path on unweighted graph

$O(n + m)$

```
vll solve(ll s, ll n, vll adj[]) {
    bool visited[n] = {0};
    visited[s] = true;

    queue<ll> q;
    q.push(s);
    vll prev(n, -1);
    while (!q.empty())
    {
        vll neighbours = adj[q.front()];
        for(auto nex: neighbours) {
            if(!visited[nex]) {
                visited[nex]=true;
                q.push(nex);
                prev[nex] = q.front();
            }
        }
        q.pop();
    }

    return prev;
}

vll reconstructPath(ll s, ll e, vll prev) {
    vll path;
    for(ll i=e; i!=-1; i=prev[i])
        path.push_back(i);
```

```cpp

    reverse(path.begin(), path.end());

    if(path[0]==s)
        return path;
    else {
        vll place;
        return place;
    }
}

vll bfs(ll s, ll e, ll n, vll adj[]) {
    vll prev = solve(s, n, adj);

    return reconstructPath(s, e, prev);
}
```

6.3 Flood Fill

$O(n+m)$

```cpp
int dir_y[] = {};
int dir_x[] = {};

int ff(int i, int j, char c1, char c2) {
    if ((i < 0) || (i >= n)) return 0;
    if ((j < 0) || (j >= m)) return 0;
    if (grid[i][j] != c1) return 0;

    int ans = 1;
    grid[i][j] = c2;

    for (int d = 0; d < 8; ++d)
        ans += floodfill(i+dir_y[d], j+dir_x[d], c1, c2);

    return ans;
}
```

;

6.4 Topological Sort (Directed Acyclic Graph)

6.4.1 DFS Variation

$O(n+m)$

```cpp
void dfs(ll at, ll n ,vpll adj[], bool visited[], vll &ts) {
    if(visited[at])
        return;

```

```
    visited[at] = true;

    vpll neighbours = adj[at];
    for(auto nex: neighbours)
        dfs(nex.first, n, adj, visited);
    ts.push_back(at);                          // Only change
}
```

6.4.2 Kahn's Algorithm

```
priority_queue<ll, vll, greater<ll>> pq;
for(ll at=0; at<n; at++)          // Push all sources of
    connected components in graph
    if(in_degree[at] == 0)
        pq.push(at);

while(!pq.empty()) {
    ll at = pq.top(); pq.pop();
    vll neighbors = adj[at];
    for(auto nex: neighbors) {
        in_degree[nex]--;
        if(in_degree[nex]>0) continue;
        pq.push(nex);
    }
}
```

6.5 Bipartite Graph Check (Undirected Graph)

$O(n+m)$

```
bool isBipartite(ll s, ll n, vll adj[]) {
    queue<ll> q;
    q.push(s);
    vll color(n, -1); color[s]=0;
    bool flag = true;
    while (!q.empty())
    {
        vll neighbours = adj[q.front()];
        for(auto nex: neighbours) {
            if(color[nex] == -1) {
                color[nex] = 1-(color[q.front()]);
                q.push(nex);
            }
            else if(color[nex] == color[q.front()]) {
                flag = false;
                break;
            }
        }
```

```
        q.pop();
    }

    return flag;
}
```

6.6 Cycle Check (Directed Graph)

$O(n+m)$

```
enum { UNVISITED = -1, VISITED = -2,  EXPLORED=-3};

void cycleCheck(ll at, ll n ,vll adj[], int visited[], ll
    dfs_parent[]) {
    visited[at] = EXPLORED;

    vll neighbours = adj[at];
    for(auto nex: neighbours) {
        if(visited[nex] == UNVISITED) {
            // Tree edges (part of the DFS spanning tree)
            dfs_parent[nex] = at;
            cycleCheck(nex, n, adj, visited);
        }
        else if(visited[nex] == EXPLORED) {
            if(nex == dfs_parent[at]) {
                // Trivial cycle
                // Do something
            }
            else {
                // Non trivial cycle - Back Edge ((u, v)
                    such that v is the ancestor of node u but
                     is not part of the DFS tree)
                // Do something
            }

        }
        else if(visited[nex] == VISITED) {
            // Forward/Cross edge ((u, v) such that v is a
                descendant but not part of the DFS tree)
            // Do something
        }

    }

    visited[at] = VISITED;
}
```

6.7 Dijkstra

$O(n \log n + m \log n)$

```
void dijkstra(ll s, vll & d, vll & p) {
    d.assign(n, LLONG_MAX);
    p.assign(n, -1);

    d[s] = 0;
    priority_queue<pll, vpll, greater<pll>> q;
    q.push({0, s});
    while (!q.empty()) {
        ll v = q.top().second;
        ll d_v = q.top().first;
        q.pop();
        if (d_v != d[v])
            continue;

        for (auto edge : adj[v]) {
            ll to = edge.first;
            ll len = edge.second;

            if (d[v] + len < d[to]) {
                d[to] = d[v] + len;
                p[to] = v;
                q.push({d[to], to});
            }
        }
    }
}
```

7 Dynamic Programming

7.1 Coin Change

$O(nm)$

```
/**
 * @brief Calculates the minimum number of coins required to
     make a target amount using dynamic programming (
    memoization).
 * @param m The target amount of money to reach.
 * @param cs Coins
 * @return The minimum number of coins needed to sum up to '
    m'
 */
ll coin_change(ll m, const vll &cs)
{
    if (m == 0)
```

```
        return 0;

    if (st[m] != -1)
        return st[m];

    auto res = oo;
    for (auto c : cs)
        if (c <= m)
            res = min(res, coin_change(m - c, cs) + 1);
    return st[m] = res;
}
```

7.1.1 Canonicality check

$O(n^3)$

```
/**
 * @brief Makes change for a given amount using a greedy
    approach.
 * Assumes the coin denominations 'xs' are sorted in
    descending order.
 */
vll greedy(ll x, ll N, const vll &xs)
{
    vll res(N, 0);
    for(ll i=0; i<N; i++)
    {
        auto q = x / xs[i];
        x -= q*xs[i];
        res[i] = q;
    }

    return res;
}

/**
 * @brief Calculates the total monetary value of a given
    combination of coins.
 */
ll value(const vll &M, ll N, const vll &xs)
{
    ll res=0;
    for(ll i=0; i<N; i++)
        res += M[i]*xs[i];
    return res;
}

/**
 * @brief Finds the smallest amount of money for which the
    greedy algorithm fails
```

```
 * to produce an optimal solution (i.e., the minimum number
     of coins).
 * This is based on a known algorithm for testing if a coin
     system is "canonical".
 */
ll min_counterexample(ll N, const vll &xs)
{
    if(N <= 2)
        return -1;

    ll ans=oo;

    for(ll i=N-2; i>=0; --i) {
        auto g = greedy(xs[i]-1, N, xs);

        vll M(N, 0);

        for(ll j=0; j<N; ++j)
        {
            M[j] = g[j] + 1;
            auto w = value(M, N, xs);
            auto G = greedy(w, N, xs);

            auto x = accumulate(M.begin(), M.end(), 0);
            auto y = accumulate(G.begin(), G.end(), 0);

            if(x < y)
                ans = min(ans, w);

            M[j]--;
        }
    }

    return ans == oo ? -1 : ans;
}
```

7.2 Knapsack

$O(nm)$

```
/**
 * @brief Finds the maximum sum possible of the knapsack
 * Can solve subset sum problem (change max to logic OR)
 */
pair<ll, vll> knapsack(ll M, const vpll &cs)
{
    ll N = cs.size() - 1; // Elements start at 1

    for(ll i=0; i<=N; i++)
```

```cpp
        st[i][0] = 0;

    for(ll m=0; m<=M; m++)
        st[0][m] = 0;

    for(ll i=1; i<=N; i++)
    {
        for(ll m = 1; m <= M; m++)
        {
            st[i][m] = st[i-1][m];
            ps[i][m] = 0;
            auto [w, v] = cs[i];

            if(w <= M && st[i-1][m-w] + v > st[i][m])
            {
                st[i][m] = st[i-1][m-w] + v;
                ps[i][m] = 1;
            }
        }
    }

    // Elements recuperation
    ll m = M;
    vll is;

    for(ll i=N; i>=1; --i)
    {
        if(ps[i][m])
        {
            is.push_back(i);
            m -= cs[i].first;
        }
    }

    reverse(is.begin(), is.end());

    return {st[N][M], is};
}
```

7.3 LIS

$O(n \log n)$

```cpp
/**
 * @param xs      Target Vector.
 * @param values  True if want values, indexes otherwise.
 * @return        Longest increasing subsequence as values
    or indexes.
```

```
 *  https://judge.yosupo.jp/problem/
    longest_increasing_subsequence
 *  source: Yogi Nam
 *  Time complexity: O(Nlog(N))
*/
vll lis(const vll& xs, bool values) {
    assert(!xs.empty());
    vll ss, idx, pre(xs.size()), ys;
    for(ll i=0; i<xs.size(); i++) {
        // change to upper_bound if want not decreasing
        ll j = lower_bound(all(ss), xs[i]) - ss.begin();
        if (j == ss.size()) ss.eb(), idx.eb();
        if (j == 0) pre[i] = -1;
        else        pre[i] = idx[j - 1];
        ss[j] = xs[i], idx[j] = i;
    }
    ll i = idx.back();
    while (i != -1)
        ys.eb((values ? xs[i] : i)), i = pre[i];
    reverse(all(ys));
    return ys;
}
```

7.4 Travelling Salesman Problem

$O(N^2 2^N)$

```
/**
 * @brief Returns the min cost hamiltonian cycles
 * @param i Current city
 * @param mask Visited cities
 * Can be modified to return the max cost
 * Can include only a set qnt of cities
 * Can modify the dist graph to a non-complete graph:
 * Set dist[i][j] = INT_MAX
 */

int tsp(int i, int mask) {
    if(mask == (1 << n) - 1)
        return dist[i][0];

    if(st[i][mask] == -1)
        return st[i][mask];

    int res = INT_MAX;
    for(int j=0; j<n; j++) {
        if(mask & (1 << j))
            continue;
```

```
            res = min(res, tsp(j, mask | (1 << j), n) + dist[i][
                j]);
        }

    return (st[i][mask] = res);
}
```

8 Math Formulas

8.1 Sum of an arithmetic progression

$S_n = \frac{n}{2}(a_1 + a_n)$

8.2 Permutation with repeated elements

$P_n = \frac{n!}{n_1!n_2!...n_k!}$

8.3 Check if is geometric progression

$a_i^2 = a_{i-1}a_{i+1}$

8.4 Bitwise equations

$a|b = a \oplus b + a\&b$
$a \oplus (a\&b) = (a|b) \oplus b$
$(a\&b) \oplus (a|b) = a \oplus b$

$a + b = a|b + a\&b$
$a + b = a \oplus b + 2(a\&b)$

$a - b = (a \oplus (a\&b)) - ((a|b) \oplus a)$
$a - b = ((a|b) \oplus b) - ((a|b) \oplus a)$
$a - b = (a \oplus (a\&b)) - (b \oplus (a\&b))$
$a - b = ((a|b) \oplus b) - (b \oplus (a\&b))$

8.5 Cube of Binomial

$(a + b)^3 = a^3 + 3a^2b + 3ab^2 + b^3$
$(a - b)^3 = a^3 - 3a^2b + 3ab^2 - b^3$

8.5.1 Sum of Cubes

$a^3 + b^3 = (a + b)(a^2 - ab + b^2)$

8.5.2 Difference of Cubes

$a^3 - b^3 = (a - b)(a^2 + ab + b^2)$

8.6 Binomial expansion

$\binom{n}{k} = \frac{n!}{k!(n-k)!}$
$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k}$

9 Facts

9.1 XOR

9.1.1 Self-inverse property

To cancel a XOR, you can XOR again the same value because $a \oplus a = 0$, so $(value \oplus a) \oplus a = value$

9.1.2 Identity element

$a \oplus 0 = a$

9.1.3 Commutative

$a \oplus b = b \oplus a$

9.1.4 Associative

$(a \oplus b) \oplus c = a \oplus (b \oplus c)$