Contents

1 Template

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

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())
```

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 Algebra

3.1 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;
}
```

3.2 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;
}
```

3.3 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;
}
```

3.4 Greatest common divisor

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

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

3.4.1 Least common multiple

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

3.4.2 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;
}
```

3.5 Linear Diophantine Equations

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

3.5.1 Any solution

```cpp
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;
}
```

3.6 Integer Factorization

3.6.1 Pollard's Rho

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

```cpp
/**
 *  @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;
}
```

4 Graphs

4.1 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);
}
```

4.2 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();
    }
}
```

4.2.1 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);
```

```

    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);
}
```

4.3 Flood Fill

$O(n+m)$

```
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;
}
```

;

4.4 Topological Sort (Directed Acyclic Graph)

4.4.1 DFS Variation

$O(n+m)$

```
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
}
```

4.4.2 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);
    }
}
```

4.5 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;
}
```

4.6 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;
}
```

4.7 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});
            }
        }
    }
}
```

5 Math Formulas

5.1 Sum of an arithmetic progression

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

5.2 Permutation with repeated elements

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

5.3 Check if is geometric progression

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

5.4 Bitwise equations

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

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

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

5.5 Cube of Binomial

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

5.5.1 Sum of Cubes

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

5.5.2 Difference of Cubes

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

5.6 Binomial expansion

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

6 Facts

6.1 XOR

6.1.1 Self-inverse property

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

6.1.2 Identity element

$a \oplus 0 = a$

6.1.3 Commutative

$a \oplus b = b \oplus a$

6.1.4 Associative

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