

Contents

1 Basic

1.1 vimrc

```
se nu ai hls et ru ic is sc cul
se re=1 ts=4 sts=4 sw=4 ls=2 mouse=a
syntax on
hi cursorline cterm=none ctermbg=89
set bg=dark
inoremap {<CR> {<CR>}<Esc>ko<tab>
inoremap ( ()<Esc>i
inoremap [ []<Esc>i
inoremap > <c-r>=ClosePair('')<CR>
inoremap ] <c-r>=ClosePair(']')<CR>
function! ClosePair(char)
    if getline('.')[col('.') - 1] == a:char
        return "\<Right>"
    else
        return a:char
    endif
endfunction
```

1.2 shell script.cpp

```
g++ -O2 -std=c++17 -Wall -Wextra -Wshadow -o
$1 $1.cpp
```

1.3 init

```
#include <bits/stdc++.h>
using namespace std;
#pragma GCC optimize("Ofast")
#define int long long
#define ld long double
#define pb push_back
#define X first
#define Y second
typedef pair<int, int> pii;
signed main(){
    ios_base::sync_with_stdio(0);
    cin.tie(0);
}
```

2 Graph

2.1 kosaraju

```
// 0-base
vector<int> graph[MAXN];
vector<int> rev_graph[MAXN];
vector<int> path;
int visit[MAXN];
int group[MAXN];
int gindex = 0;
void dfs1(int root) {
    if (visit[root]) return;
    visit[root] = 1;
    for (auto it : graph[root])
        dfs1(it);
    path.push_back(root);
}
void dfs2(int root, int gid) {
    if (visit[root]) return;
    visit[root] = 1;
```

```
group[root] = gid;
for (auto it : rev_graph[root])
    dfs2(it, gid);
}
void kosaraju (int node_cnt) {
    for (int i = 0; i < node_cnt; ++i)
        for (auto child : graph[i])
            rev_graph[child].push_back(i);

    for (int i = 0; i < node_cnt; ++i)
        if (!visit[i]) dfs1(i);

    for (int i = 0; i < node_cnt; ++i)
        visit[i] = 0;

    for (int i = path.size() - 1; i >= 0; i --)
        if (!visit[path[i]])
            dfs2(path[i], gindex++);
}
```

2.2 MCMF

```
//use this to find max flow slower
struct MCMF { // 0-base
    struct edge {
        ll from, to, cap, flow, cost, rev;
    } * past[MAXN];
    vector<edge> G[MAXN];
    bitset<MAXN> inq;
    // mx for the max flow
    ll dis[MAXN], up[MAXN], s, t, mx, n;
    bool BellmanFord(ll &flow, ll &cost) {
        fill(dis, dis + n, INF);
        queue<ll> q;
        q.push(s), inq.reset(), inq[s] = 1;
        up[s] = mx - flow, past[s] = 0, dis[s] = 0;
        while (!q.empty()) {
            ll u = q.front();
            q.pop(), inq[u] = 0;
            if (!up[u]) continue;
            for (auto &e : G[u])
                if (e.flow != e.cap && dis[e.to] >
                    dis[u] + e.cost) {
                    dis[e.to] = dis[u] + e.cost, past[
                        e.to] = &e;
                    up[e.to] = min(up[u], e.cap - e.
                        flow);
                    if (!inq[e.to]) inq[e.to] = 1, q.
                        push(e.to);
                }
        }
        if (dis[t] == INF) return 0;
        flow += up[t], cost += up[t] * dis[t];
        for (ll i = t; past[i]; i = past[i]->
            from) {
            auto &e = *past[i];
            e.flow += up[t], G[e.to][e.rev].flow
                -= up[t];
        }
        return 1;
    }
    ll MinCostMaxFlow(ll _s, ll _t, ll &cost)
    {
```

```

    s = _s, t = _t, cost = 0;
    ll flow = 0;
    while (BellmanFord(flow, cost));
    return flow;
}
void init(ll _n, ll _mx) {
    n = _n, mx = _mx;
    for (int i = 0; i < n; ++i) G[i].clear();
}
void add_edge(ll a, ll b, ll cap, ll cost)
{
    G[a].pb(edge{a, b, cap, 0, cost, G[b].
        size()});
    G[b].pb(edge{b, a, 0, 0, -cost, G[a].
        size() - 1});
}
};

```

2.3 dijkstra

```

vector<pii> graph[MAXN];
int dist[MAXN] = {};
void dijkstra(int s) { // 0-base
    int is_visited[MAXN] = {};
    priority_queue<pii, vector<pii>, greater
        <pii>> pq;
    dist[s] = 0;
    pq.push({0, s});
    while (!pq.empty()) {
        int curr_node = pq.top().second;
        pq.pop();
        if (is_visited[curr_node])
            continue;
        is_visited[curr_node] = 1;
        for (auto edge : graph[curr_node]) {
            if (!is_visited[edge.first] and
                dist[edge.first] > dist[
                    curr_node] + edge.second) {
                dist[edge.first] = dist[
                    curr_node] + edge.second;
                pq.push({dist[curr_node] +
                    edge.second, edge.first});
            }
        }
    }
} // graph[from].pb({to, cost})

```

2.4 tarjan

```

vector<pii> graph[MAXN];
int dfn[MAXN], low[MAXN], visited_cnt = 0;
int mapping[MAXN], bcc_cnt = 0;
vector<pii> tree[MAXN];
int tree_mater[MAXN];
bool ans[MAXN];

stack<int> tarjan_stack;
void tarjan(int curr, int parent) {
    tarjan_stack.push(curr);
    dfn[curr] = low[curr] = ++visited_cnt;

    for (pii nxt : graph[curr]) {
        if (!dfn[nxt.first]) {
            tarjan(nxt.first, curr);

```

```

        }
        if (nxt.first != parent) {
            low[curr] = min(low[curr], low[
                nxt.first]);
        }
    }

    if (low[curr] == dfn[curr]) {
        while (tarjan_stack.top() != curr) {
            mapping[tarjan_stack.top()] =
                bcc_cnt;
            tarjan_stack.pop();
        }
        mapping[tarjan_stack.top()] =
            bcc_cnt;
        tarjan_stack.pop();
        bcc_cnt++;
    }
}

```

3 Data Structure

3.1 DSU

```

struct dsu {
    vector<size_t> pa, size;
    dsu(size_t size_) : pa(size_), size(size_,
        1) {
        iota(pa.begin(), pa.end(), 0);
    }
    void unite(size_t x, size_t y) {
        x = find(x), y = find(y);
        if (x == y) return;
        if (size[x] < size[y]) swap(x, y);
        pa[y] = x;
        size[x] += size[y];
    }
    size_t find(size_t x) {
        return pa[x] == x ? x : find(pa[x]);
    }
    void com_unite(size_t x, size_t y) {
        pa[find(x)] = find(y);
    }
};

```

3.2 BIT

```

int bit[MAXN+1]={0};
void modify(int pos,int val){
    while(pos<=n){
        bit[pos]+=val;
        pos+=pos&-pos;
    }
}
int prefix_sum(int pos){
    int ans=0;
    while(pos>=1){
        ans+=bit[pos];
        pos-=pos&-pos;
    }
    return ans;
}

```

3.3 LCT

// warning: it may not correct

```

struct LCT { // use tree size as example
#define lson(x) (tree[x].ch[0])
#define rson(x) (tree[x].ch[1])
#define NORMAL 0
#define fa(x) (tree[x].fa)
#define get(x) (tree[tree[x].fa].ch[1] == x)
#define is_root(x) (tree[tree[x].fa].ch[0]
    != x and tree[tree[x].fa].ch[1] != x)
    struct Node {
        int ch[2] = {};
        int fa = 0, rev = 0;
        int size = 0, v_size = 0;
    } tree[MAXN + 1];
    void reverse(int x) {
        if (x) {
            swap(lson(x), rson(x));
            tree[x].rev ^= 1;
        }
    }
    void push_up(int x) {
        tree[x].size = 1 + tree[lson(x)].size +
            tree[rson(x)].size + tree[x].v_size;
        // and other attribute you want
    }
    void push_down(int x) {
        if (tree[x].rev != NORMAL) {
            reverse(lson(x));
            reverse(rson(x));
            tree[x].rev = NORMAL;
        }
        // and other tag you want
    }
    void update(int x) {
        if (!is_root(x)) update(fa(x));
        push_down(x);
    }
    void rotate(int x) {
        int father = fa(x), gfather = fa(father),
            is_right = get(x);
        if (!is_root(father)) tree[gfather].ch[
            get(father)] = x;
        tree[father].ch[is_right] = tree[x].ch[!
            is_right], tree[tree[x].ch[!is_right]
            ].fa = father;
        tree[x].ch[!is_right] = father, fa(x) =
            gfather, fa(father) = x;
        push_up(father), push_up(x);
    }
    void splay(int x) {
        update(x);
        while (!is_root(x)) {
            int father = fa(x);
            if (!is_root(father))
                rotate(get(father) == get(x) ?
                    father : x);
            rotate(x);
        }
        push_up(x);
    }
    int access(int x) {
        int new_ch = 0;
        while (x) {
            splay(x);
            tree[x].v_size -= tree[new_ch].size -
                tree[rson(x)].size;

```

```

            rson(x) = new_ch, push_up(x);
            new_ch = x, x = fa(x);
        }
        return new_ch;
    }
    void make_root(int x) {
        int new_splay_root = access(x);
        reverse(new_splay_root);
    }
    void link(int x, int p) {
        make_root(x);
        splay(x);
        fa(x) = p;
        tree[p].v_size += tree[x].size;
        push_up(p);
    }
    void split(int x, int y) {
        make_root(x);
        access(y);
        splay(y);
    }
    void cut(int x, int y) {
        split(x, y);
        fa(x) = lson(y) = 0;
        push_up(y);
    }
    int find(int x) {
        access(x);
        splay(x);
        push_down(x);
        while (lson(x)) {
            x = lson(x);
            push_down(x);
        }
        return x;
    }
    int query(int x, int y) {
        split(x, y);
        return tree[y].size;
    }
} lctree;

```

4 Math

4.1 simplex

```

namespace simplex {
    using T = long double;
    const int N = 410, M = 30010;
    const T eps = 1e-7;
    int n, m;
    int Left[M], Down[N];
    // Ax <= b, max c^T x
    // result : v, xi = sol[i]. 1 based
    T a[M][N], b[M], c[N], v, sol[N];
    bool eq(T a, T b) { return fabs(a - b) <
        eps; }
    bool ls(T a, T b) { return a < b && !eq(a,
        b); }
    void init(int p, int q) {
        n = p; m = q; v = 0;
        for(int i = 1; i <= m; i++){
            for(int j = 1; j <= n; j++) a[i][j]=0;
        }
        for(int i = 1; i <= m; i++) b[i]=0;
    }
}

```

```

    for(int i = 1; i <= n; i++) c[i]=sol[i]
        ]=0;
}
void pivot(int x,int y) {
    swap(Left[x], Down[y]);
    T k = a[x][y]; a[x][y] = 1;
    vector<int> nz;
    for(int i = 1; i <= n; i++){
        a[x][i] /= k;
        if(!eq(a[x][i], 0)) nz.push_back(i);
    }
    b[x] /= k;
    for(int i = 1; i <= m; i++){
        if(i == x || eq(a[i][y], 0)) continue;
        k = a[i][y]; a[i][y] = 0;
        b[i] -= k*b[x];
        for(int j : nz) a[i][j] -= k*a[x][j];
    }
    if(eq(c[y], 0)) return;
    k = c[y]; c[y] = 0;
    v += k*b[x];
    for(int i : nz) c[i] -= k*a[x][i];
}
// 0: found solution, 1: no feasible
// solution, 2: unbounded
int solve() {
    for(int i = 1; i <= n; i++) Down[i] = i;
    for(int i = 1; i <= m; i++) Left[i] = n+
        i;
    while(1) { // Eliminating negative b[i]
        int x = 0, y = 0;
        for(int i = 1; i <= m; i++) if (ls(b[i]
            ], 0) && (x == 0 || b[i] < b[x])) x = i;
        if(x == 0) break;
        for(int i = 1; i <= n; i++) if (ls(a[x]
            ][i], 0) && (y == 0 || a[x][i] < a[
            x][y])) y = i;
        if(y == 0) return 1;
        pivot(x, y);
    }
    while(1) {
        int x = 0, y = 0;
        for(int i = 1; i <= n; i++)
            if (ls(0, c[i]) && (!y || c[i] > c[y]
                )) y = i;
        if(y == 0) break;
        for(int i = 1; i <= m; i++)
            if (ls(0, a[i][y]) && (!x || b[i]/a[
                i][y] < b[x]/a[x][y])) x = i;
        if(x == 0) return 2;
        pivot(x, y);
    }
    for(int i = 1; i <= m; i++) if(Left[i]
        <= n) sol[Left[i]] = b[i];
    return 0;
}
}

```

4.2 fraction

```

struct Frac{
    ll fz;
    ll fm;
    Frac(){}

```

```

    Frac(ll n):fz(n),fm(1){}
};
ll GCD(ll a,ll b){
    if(b==0){
        return a;
    }
    return GCD(b,a%b);
}
Frac Simplify(Frac a){
    Frac ans;
    ll gcd=GCD(abs(a.fz),a.fm);
    ans.fz=a.fz/gcd;
    ans.fm=a.fm/gcd;
    return ans;
}
Frac operator+(Frac a,Frac b){
    Frac ans;
    ans.fz=a.fz*b.fm+b.fz*a.fm;
    ans.fm=a.fm*b.fm;
    return Simplify(ans);
}
Frac operator-(Frac a,Frac b){
    Frac ans;
    ans.fz=a.fz*b.fm-b.fz*a.fm;
    ans.fm=a.fm*b.fm;
    return Simplify(ans);
}
Frac operator*(Frac a,Frac b){
    Frac ans;
    ans.fz=a.fz*b.fz;
    ans.fm=a.fm*b.fm;
    return Simplify(ans);
}
Frac operator/(Frac a,Frac b){
    Frac ans;
    ans.fz=a.fz*b.fm;
    ans.fm=a.fm*b.fz;
    return Simplify(ans);
}
bool operator<(Frac a,Frac b){
    return a.fz*b.fm<b.fz*a.fm;
}
bool operator==(Frac a,Frac b){
    return a.fz==b.fz&& a.fm==b.fm;
}

```

4.3 mobius sequence and linear sieve

```

vector<int> primes;
bool not_prime[MAXN + 1];
char mobius[MAXN + 1];
void gen_factorize (int n) {
    for (int i = 2; i <= n; ++i) {
        if (!not_prime[i]) {
            primes.push_back(i);
            mobius[i] = -1;
        }
        for (int prime : primes) {
            if (i * prime > n)
                break;
            not_prime[i * prime] = 1;
            if (i % prime == 0) {
                mobius[i * prime] = 0;
                break;
            }
        }
    }
}

```

```

        mobius[i * prime] = mobius[i] *
            -1;
    }
}

```

4.4 miller rabin

```

// n < 4,759,123,141 3 : 2, 7, 61
// n < 1,122,004,669,633 4 : 2, 13, 23,
    1662803
// n < 3,474,749,660,383 6 : pirmses <= 13
// n < 2^64 7 :
// 2, 325, 9375, 28178, 450775, 9780504,
    1795265022
bool Miller_Rabin(ll a, ll n) {
    if ((a = a % n) == 0) return 1;
    if (n % 2 == 0) return n == 2;
    ll tmp = (n - 1) / ((n - 1) & (1 - n));
    ll t = __lg(((n - 1) & (1 - n))), x = 1;
    for (; tmp; tmp >>= 1, a = mul(a, a, n))
        if (tmp & 1) x = mul(x, a, n);
    if (x == 1 || x == n - 1) return 1;
    while (--t)
        if ((x = mul(x, x, n)) == n - 1) return
            1;
    return 0;
}

```

4.5 pollard rho

```

map<ll, int> cnt;
void PollardRho(ll n) {
    if (n == 1) return;
    if (prime(n)) return ++cnt[n], void();
    if (n % 2 == 0) return PollardRho(n / 2),
        ++cnt[2], void();
    ll x = 2, y = 2, d = 1, p = 1;
    #define f(x, n, p) ((mul(x, x, n) + p) % n)
    while (true) {
        if (d != n && d != 1) {
            PollardRho(n / d);
            PollardRho(d);
            return;
        }
        if (d == n) ++p;
        x = f(x, n, p), y = f(f(y, n, p), n, p);
        d = gcd(abs(x - y), n);
    }
}

```

4.6 mod inverse

```

int modInverse(int num, int p) {
    int p0 = p;
    int y = 0, x = 1;
    if (p == 1) return 0;
    while (num > 1) {
        int q = num / p;
        int t = p;
        p = num % p, num = t;
        t = y, y = x - q * y;
        x = t;
    }
    if (x < 0) x += p0;
}

```

```

return x;
}

```

5 String

5.1 lexicographically smallest rotation

```

string s;
int n=s.size();
vector<int> kmp(n*2,-1);
int ans=0;
for(int i=1;i<2*n;i++){
    int x=kmp[i-ans-1];
    while(x!=-1 && s[i%n]!=s[(ans+x+1)%n]){
        if(s[i%n]<s[(ans+x+1)%n]) ans=i-x-1;
        x=kmp[x];
    }
    if(x==-1 && s[i%n]!=s[ans%n]){
        if(s[i%n]<s[ans%n]) ans=i;
        kmp[i-ans]=-1;
    }
    else kmp[i-ans]=x+1;
}

```

6 Other

6.1 fast clear array

```

struct ArrayNode{
    int val;
    int gen=-1;
};
struct Array{
    ArrayNode _[MAXN];
    int gen=0;
    void zero(){
        gen++;
    }
    int get(int pos){
        if(_[pos].gen==gen){
            return _[pos].val;
        }else{
            return 0;
        }
    }
    void set(int pos,int val){
        _[pos].gen=gen;
        _[pos].val=val;
    }
};

```

6.2 priority queue

```

priority_queue<type, vector<type>, function<
    bool(type, type)>> pq(cmp);
// change type and cmp as you want

```

6.3 fast multiplication

```

ll mul (ull x, ull y, ll p) {
    return (x * y - (ull)((ld)x / p * y) * p
        + p) % p;
}

```