

Contents

1 Basic

1.1 vimrc

```
set nu ai si cin ts=4 sw=4 sts=4 mouse=a expandtab
syn on
imap {<CR> {<CR>}<Esc>ko
map <F5> :w<LF>:!g++ -O2 -std=c++11 % && echo "----
    Start-----" && ./a.out<LF>
map <F6> :w<LF>:!g++ -O2 -std=c++11 % && echo "----
    Start-----" && time ./a.out < input.in<LF>
map <F9> :t!e %<LF>
```

1.2 int128

```
__int128 parse(string &s) {
    __int128 ret = 0;
    for (int i = 0; i < (int)s.size(); i++)
        if ('0' <= s[i] && s[i] <= '9')
            ret = 10 * ret + s[i] - '0';
    return ret;
}
#define O ostream
O& operator << (O &out, __int128_t v) {
    O::sentry s(out);
    if (s) {
        __uint128_t uv = v < 0 ? -v : v;
        char buf[128], *d = end(buf);
        do {
            *(&d) = "0123456789"[uv % 10];
            uv /= 10;
        } while (uv != 0);
        if (uv < 0)
            *(&d) = '-';
        int len = end(buf) - d;
        if (out.rdbuf()->sputn(d, len) != len)
            out.setstate(ios_base::badbit);
    }
    return out;
}
#define I istream
I& operator >> (I &in, __int128_t &v) {
    string s; in >> s;
    v = parse(s);
    return in;
}
```

2 Flow

2.1 Dinic

```
struct Graph{
    struct Node; struct Edge;
    int V;
    struct Node : vector<Edge*>{
        iterator cur; int d;
        Node(){ clear(); }
    }_memN[MAXN], *node[MAXN];
    struct Edge{
        Node *u, *v;
        Edge *rev;
        LL c, f;
        Edge(){ }
        Edge(Node *u, Node *v, LL c, Edge *rev) : u(u), v(v),
            c(c), f(0), rev(rev){ }
    }_memE[MAXM], *ptrE;
```

```
Graph(int _V) : V(_V) {
    for (int i = 0; i < V; i++)
        node[i] = _memN + i;
    ptrE = _memE;
}
void addEdge(int _u, int _v, LL _c){
    *ptrE = Edge(node[_u], node[_v], _c, ptrE + 1);
    node[_u]->push_back(ptrE++);
    *ptrE = Edge(node[_v], node[_u], _c, ptrE - 1); //
        direction
    node[_v]->push_back(ptrE++);
}
Node *s, *t;
LL maxFlow(int _s, int _t){
    s = node[_s], t = node[_t];
    LL flow = 0;
    while (bfs()) {
        for (int i = 0; i < V; i++)
            node[i]->cur = node[i]->begin();
        flow += dfs(s, INF);
    }
    return flow;
}
bool bfs(){
    for (int i = 0; i < V; i++) node[i]->d = -1;
    queue<Node*> q; q.push(s); s->d = 0;
    while (q.size()) {
        Node *u = q.front(); q.pop();
        for (auto e : *u) {
            Node *v = e->v;
            if (!v->d && e->c > e->f)
                q.push(v), v->d = u->d + 1;
        }
    }
    return ~t->d;
}
LL dfs(Node *u, LL a){
    if (u == t || !a) return a;
    LL flow = 0, f;
    for (; u->cur != u->end(); u->cur++) {
        auto &e = *u->cur; Node *v = e->v;
        if (u->d + 1 == v->d && (f = dfs(v, min(a, e->c -
            e->f))) > 0) {
            e->f += f; e->rev->f -= f;
            flow += f; a -= f;
            if (!a) break;
        }
    }
    return flow;
}
};
```

2.2 MCMF

```
struct Graph {
    struct Node; struct Edge; int V;
    struct Node : vector<Edge*> {
        bool inq; Edge *pa; LL a, d;
        Node() { clear(); }
    }_memN[MAXN], *node[MAXN];
    struct Edge{
        Node *u, *v; Edge *rev;
        LL c, f, _c; Edge() {}
        Edge(Node *u, Node *v, LL c, LL _c, Edge *rev)
            : u(u), v(v), c(c), f(0), _c(_c), rev(rev) {}
    }_memE[MAXM], *ptrE;
    Graph(int _V) : V(_V) {
        for (int i = 0; i < V; i++)
            node[i] = _memN + i;
        ptrE = _memE;
    }
    void addEdge(int u, int v, LL c, LL _c) {
        *ptrE = Edge(node[u], node[v], c, _c, ptrE + 1);
        node[u]->push_back(ptrE++);
        *ptrE = Edge(node[v], node[u], 0, -_c, ptrE - 1);
```

```

        node[v]->push_back(ptrE++);
    }
    Node *s, *t;
    bool SPFA() {
        for (int i = 0 ; i < V ; i++) node[i]->d = INF,
            node[i]->inq = false;
        queue<Node*> q; q.push(s); s->inq = true;
        s->d = 0, s->pa = NULL, s->a = INF;
        while (q.size()) {
            Node *u = q.front(); q.pop(); u->inq = false;
            for (auto &e : *u) {
                Node *v = e->v;
                if (e->c > e->f && v->d > u->d + e->c) {
                    v->d = u->d + e->c;
                    v->pa = e; v->a = min(u->a, e->c - e->f);
                    if (!v->inq) q.push(v), v->inq = true;
                }
            }
        }
        return t->d != INF;
    }
}

pLL maxFlowMinCost(int _s, int _t) {
    s = node[_s], t = node[_t];
    pLL res = {0, 0};
    while (SPFA()) {
        res.F += t->a;
        res.S += t->d * t->a;
        for (Node *u = t ; u != s ; u = u->pa->u) {
            u->pa->f += t->a;
            u->pa->rev->f -= t->a;
        }
    }
    return res;
}
};

```

3 DataStructure

3.1 unorderedMap

```

struct Key {
    int F, S;
    Key() {}
    Key(int _x, int _y) : F(_x), S(_y) {}
    bool operator == (const Key &b) const {
        return tie(F, S) == tie(b.F, b.S);
    }
};

struct KeyHasher {
    size_t operator() (const Key &b) const {
        return k.F + k.S * 100000;
    }
};

typedef unordered_map<Key, int, KeyHasher> map_t;

```

3.2 pbdsTree

```
#include <bits/extc++.h>
using namespace __gnu_pbds;
using namespace std;
typedef tree<int, null_type, less<int>, rb_tree_tag,
    tree_order_statistics_node_update> set_t;
typedef cc_hash_table<int, int> umap_t;
int main() {
    set_t s; s.insert(12); s.insert(505);
    assert(*s.find_by_order(0) == 12);
    assert(s.find_by_order(2) == end(s));
    assert(s.order_of_key(12) == 0);
    assert(s.order_of_key(505) == 1);
    s.erase(12);
    assert(*s.find_by_order(0) == 505);
    assert(s.order_of_key(505) == 0);
}
```

3.3 pbdsHeap

```
#include <bits/extc++.h>
typedef __gnu_pbds::priority_queue<int> heap_t;
heap_t a, b;
int main() {
    a.clear(); b.clear();
    a.push(1); a.push(3);
    b.push(2); b.push(4);
    assert(a.top() == 3);
    assert(b.top() == 4);
    a.join(b);
    assert(a.top() == 4);
    assert(b.empty());
}
```

3.4 Sptr

```
template <typename T> struct Sptr{
    pair<T, int> *p;
    T *operator->(){return &p->F;}
    T &operator*(){return p->F;}
    operator pair<T, int>*(){}return p;}
    Sptr &operator = (const Sptr& t){
        if (p && !--p->S) delete p;
        (p = t.p) && ++p->S;
        return *this;
    }
    Sptr(pair<T, int> *t = 0) : p(t){ p && ++p->S;}
    Sptr(const Sptr &t) : p(t.p) { p && ++p->S; }
    ~Sptr(){ if (p && !--p->S) delete p; }
};
```

3.5 Treap

```
//<<<<<<<<PERSISTENT
#define PTR Sptr<Node>
//=====
#define PTR Node*
//>>>>>>>>ORIGIN
#define PNN pair<PTR, PTR>
struct Treap {
    struct Node {
        PTR l; PTR r;
        int sz; char c;
        Node (char c = 0) : c(c), l(NULL), r(NULL) {
            sz = 1;
        }
    };
    vector<PTR> rt;
    Treap() { rt.resize(rt.size() + 1, NULL); }
//<<<<<<<<PRESISTENT
//=====
~Treap() { clear(rt.back()) }
void clear(PTR u) {
    if (u) clear(u->l), clear(u->r), delete u;
}
//>>>>>>>>ORIGIN
inline PTR _new(const Node &u) {
//<<<<<<<<PERSISTENT
return PTR(new _ptrCntr<Node>(u));
//=====
return new Node(u.v);
//>>>>>>>>ORIGIN
}
inline int size(PTR &u) {
return u ? u->sz : 0;
}
inline PTR& pull(PTR &u) {
u->sz = 1 + size(push(u->l)) + size(push(u->r));
// pull function
return u;
}
```

[illegible]

3.6 SegmentTree

```
//<<<<<<<<PRESISTENT
#define PTR Sptr<Node>
//=====
#define PTR Node*
//>>>>>>>>ORIGIN
struct SegmentTree {
    struct Node {
        int L, R, v; PTR l; PTR r;
        Node (int L = 0, int R = 0) : v(0),
            l(NULL), r(NULL), L(L), R(R) {}
    };
//<<<<<<<<PRESISTENT
//=====
PTR buf; PTR ptr;
~SegmentTree(){ clear(rt.back()); delete []buf; }
void clear(Node *u){
    if (u) clear(u->l), clear(u->r), delete u;
}
//>>>>>>>>ORIGIN
vector<PTR> rt;
```

```
SegmentTree(int n) {
    rt.resize(rt.size() + 1, NULL);
    rt.back() = build(0, n);
//<<<<<<<<<PRESISTENT
//=====================================================
    buf = new Node[__lg(n) * 4 + 5];
//>>>>>>>>ORIGIN
}
inline PTR _new(const Node &u) {
//<<<<<<<<<PERSISTENT
    return PTR(new _ptrCntr <Node>(u));
//=====================================================
    return new Node(u.L, u.R);
//>>>>>>>>ORIGIN
}
PTR build(int L, int R) {
    PTR u = _new(Node(L, R));
    if (u->R - u->L == 1)
        return u;
    int M = (R + L) >> 1;
    u->l = build(L, M);
    u->r = build(M, R);
    return pull(u);
}
PTR pull(PTR u, PTR l, PTR r) {
    if (!l || !r) return l ? l : r;
    push(l); push(r);
    // pull function
    return u;
}
PTR pull(PTR u) { return pull(u, u->l, u->r); }
void push(PTR u) {
    if (!u) return ;
    // push function
}
PTR query(int qL, int qR, PTR u = NULL) {
//<<<<<<<<<PRESISTENT
    if (!u) u = rt.back();
//=====================================================
    if (!u) u = rt.back(), ptr = buf;
//>>>>>>>>ORIGIN
    if (u->R <= qL || qR <= u->L) return NULL;
    if (qL <= u->L && u->R <= qR) return u;
    push(u);
//<<<<<<<<<PRESISTENT
    PTR ret = _new(Node(qL, qR));
    return pull(ret, query(qL, qR, u->l), query(qL, qR,
        u->r));
//=====================================================
    return pull(ptr++, query(qL, qR, u->l), query(qL,
        qR, u->r));
//>>>>>>>>ORIGIN
}
PTR modify(int mL, int mR, int v, PTR u = NULL) {
    if (!u) u = rt.back();
    if (u->R <= mL || mR <= u->L) return u;
//<<<<<<<<<PRESISTENT
    PTR ret = _new(*u);
    if (mL <= u->L && u->R <= mR) {
        // tag;
        return ret;
    }
    push(u);
    ret->l = modify(mL, mR, v, u->l);
    ret->r = modify(mL, mR, v, u->r);
    return pull(ret);
//=====
    if (mL <= u->L && u->R <= mR) {
        // modify function
        return u;
    }
    push(u);
    modify(mL, mR, v, u->l);
    modify(mL, mR, v, u->r);
    return pull(u);
//>>>>>>>>ORIGIN
}
```

};

```

        minW = min(minW, maxW);
    }
    return minW;
}

```

3.7 SparseTable

```

struct SparseTable{
    vector<vector<int> > data;
    int (*op)(int a, int b);
    SparseTable(vector<int> &arr, int (*_op)(int a, int b
        )) {
        op = _op;
        int n = (int)arr.size(), lgN = __lg(n) + 1;
        data.resize(lgN);
        for (int i = 0 ; i < n ; i++)
            data[0].push_back(arr[i]);
        for (int h = 1 ; h < lgN ; h++){
            int len = 1 << (h - 1), i = 0;
            for (; i + len < n ; i++)
                data[h].push_back(op(data[h-1][i], data[h-1][i+
                    len]));
            if (!i) break;
            for (; i < n ; i++)
                data[h].push_back(data[h-1][i]);
        }
    }
    int query(int l, int r){
        int h = __lg(r - l);
        int len = 1 << h;
        return op(data[h][l], data[h][r-len]);
    }
};

```

3.8 BIT

```

struct BIT {
    vector<int> data; int n;
    BIT(int n) : n(n) {
        data.clear(); data.resize(n + 1, 0);
    }
    int lowbit(int x) { return x & -x; }
    int query(int x) { x++;
        int ret = 0;
        while (x > 0) ret += data[x], x -= lowbit(x);
        return ret;
    }
    void modify(int x, int d) { x++;
        while (x <= n) data[x] += d, x += lowbit(x);
    }
};

```

4 Graph

4.1 MMC

```

double MMC(vector<vector<Edge> > &G) {
    int n = G.size(); G.resize(n + 1);
    for (int i = 0 ; i < n ; i++)
        G[n].push_back({i, 0});
    n++;
    vector<vector<LL> > d(n, vector<LL>(n + 1, INF));
    d[n - 1][0] = 0;
    for (int k = 1 ; k <= n ; k++)
        for (int i = 0 ; i < n ; i++)
            for (auto &e : G[i])
                d[e.v][k] = min(d[e.v][k], d[i][k - 1] + e.w);
    double minW = INF;
    for (int i = 0 ; i < n ; i++) {
        double maxW = -INF;
        for (int k = 0 ; k < n ; k++)
            maxW = max(maxW, (d[i][n] - d[i][k]) / double(n -
                k));
    }
}

```