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> :tabe %.in<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 Treap

```
#define PNN pair<Node*, Node*>
struct Treap {
    struct Node {
        Node *l, *r;
        int sz, v;
        // data
        int minV;
        // tag
        int add;
        bool rev;
        Node (int v = 0) : v(v) {
            l = r = NULL;
            sz = 1;
            add = 0; rev = false;
        }
    }*rt, _mem[MAXN], *ptr;
    Treap() { rt = NULL; ptr = _mem; }
    inline int size(Node *u) {
        return u ? u->sz : 0;
    }
    inline Node*& pull(Node *&u) {
        u->sz = 1 + size(push(u->l)) + size(push(u->r))
            ;
        // pull function
        return u;
    }
    inline Node*& push(Node *&u) {
        if (!u) return u;
        // push function
        return u;
    }
    PNN split(Node *T, int x) {
        if (!T) return {(Node*)NULL, (Node*)NULL};
        if (size(push(T)->l) < x) {
            PNN tmp = split(T->r, x - size(T->l) - 1);
            T->r = tmp.F;
            return {pull(T), tmp.S};
        } else {
            PNN tmp = split(T->l, x);
            T->l = tmp.S;
            return {tmp.F, pull(T)};
        }
    }
    Node* merge(Node *T1, Node *T2) {
        if (!T1 || !T2) return T1 ? T1 : T2;
        if (rand() % (size(T1) + size(T2)) < size(T1))
            {
            T1->r = merge(push(T1)->r, T2);
            return pull(T1);
        } else {
            T2->l = merge(T1, push(T2)->l);
            return pull(T2);
        }
    }
};
```