Contents

1 Basic

1.1 vimrc

1.2 int128

```
__int128 parse(string &s) {
    int128 ret = 0;
  for (int i = 0 ; i < (int)s.size() ; i++)</pre>
    if ('0' <= s[i] && s[i] <= '9')</pre>
      ret = 10 * ret + s[i] - '0';
  return ret;
#define O ostream
0& operator << (0 &out, __int128_t v) {</pre>
  0::sentry s(out);
  if (s) {
      _uint128_t uv = v < 0 ? -v : v;
    char buf[128], *d = end(buf);
      *(--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++)</pre>
       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); //
    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 \rightarrow 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;</pre>
    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

```
//<<<<<<PERSISTENT
#define PTR Sptr<Node>
//========
#define PTR Node*
//>>>>>>ORIGIN
#define PNN pair<PTR, PTR>
struct Treap {
 struct Node {
   PTR 1; PTR r;
   int sz; char c;
   Node (char c = 0) : c(c), l(NULL), r(NULL) {
     sz = 1;
//<<<<<<PRESISTENT
 PTR ver[MAXVER];
 int verCnt;
 Treap() { verCnt = 0; }
 inline Sptr<Node> copy(Sptr<Node> &u){
   return _new(*u);
 }
//========
 PTR rt;
 Treap() { rt = NULL; }
 ~Treap() {
   clear(rt)
 void clear(Node *u) {
   if (!u) return ;
   clear(u->1);
   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\rightarrow sz = 1 + size(push(u\rightarrow l)) + size(push(u\rightarrow r));
    // pull function
    return u;
  inline PTR& push(PTR &u) {
    if (!u) return u;
    // push function
    return u;
  PNN split(PTR &T, int x) {
    if (!T) return {(PTR)NULL, (PTR)NULL};
//<<<<<<PRESISTENT
    Sptr<Node> res = copy(T);
    if (size(T->1) < x){
      PNN tmp = split(T \rightarrow r, x - 1 - size(<math>T \rightarrow l));
      res->r = tmp.F;
      return {pull(res), tmp.S};
    } else {
      PNN tmp = split(T \rightarrow 1, x);
      res->l = tmp.S;
      return {tmp.F, pull(res)};
    }
//========
    if (size(push(T)->1) < x) {
      PNN tmp = split(T \rightarrow r, x - size(<math>T \rightarrow l) - 1);
      T \rightarrow r = tmp.F;
      return {pull(T), tmp.S};
    } else {
      PNN tmp = split(T->1, x);
      T \rightarrow 1 = tmp.S;
      return {tmp.F, pull(T)};
    }
//>>>>>ORIGIN
  PTR merge(PTR &T1, PTR &T2) {
    if (!T1 || !T2) return T1 ? T1 : T2;
//<<<<<<PRESISTENT
    Sptr<Node> res;
    if (rand() % (size(T1) + size(T2)) < size(T1)){</pre>
      res = copy(T1);
      res->r = merge(T1->r, T2);
    } else {
      res = copy(T2);
      res->l = merge(T1, T2->l);
    return pull(res);
    if (rand() % (size(T1) + size(T2)) < size(T1)) {</pre>
      T1->r = merge(push(T1)->r, T2);
       return pull(T1);
    } else {
      T2 \rightarrow l = merge(T1, push(T2) \rightarrow l);
      return pull(T2);
//>>>>>>ORIGIN
  }
};
```