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

```
#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) = uv % 10 + '0'; 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 = 0;
  for (int i = 0 ; i < (int)s.size() ; i++)
    if ('0' <= s[i] && s[i] <= '9')
      v = 10 * v + s[i] - '0';
  return in;
}
```

2 Flow

2.1 Dinic

```
struct Graph { int n; struct Edge;
  struct Node : vector<Edge*> { int d;
  }_memN[MAXN], *node[MAXN], *s, *t;
  struct Edge { Node *v; Edge *r; LL c; Edge() {}
    Edge(Node *v, Edge *r, LL c) : v(v), r(r), c(c) {}
  }_memE[MAXM], *ptrE;
  inline void addEdge(int u, int v, LL c) {
    Edge *pos = ptrE;
    node[u]->emplace_back(new (ptrE++) Edge(node[v],
        pos + 1, c));
    node[v]->emplace_back(new (ptrE++) Edge(node[u],
        pos, c));
  }
  Graph (int n) : n(n) { ptrE = _memE;
    for (int i = 0 ; i < n ; i++)
      node[i] = _memN + i;
  }
  inline LL maxFlow(int _s, int _t) {
    s = node[_s]; t = node[_t]; LL flow = 0;
    while (bfs()) flow += dfs(s, INF);
    return flow;
  }
  inline bool bfs() {
    for (int i = 0 ; i < n ; i++)
      node[i]->d = -1;
    queue<Node*> q; q.push(s); s->d = 0;
    while (q.size()) {
      auto u = q.front(); q.pop();
      for (auto &e : *u) {
        if (!e->c || ~e->v->d) continue;
```

```
        e->v->d = u->d + 1; q.push(e->v);
      }
    }
    return ~t->d;
  }
  LL dfs(Node *u, LL a) {
    if (u == t || !a) return a; LL flow = 0;
    for (auto &e : *u) {
      if (u->d + 1 != e->v->d) continue;
      LL f = dfs(e->v, min(e->c, a));
      e->c -= f; e->r->c += f;
      flow += f; a -= f;
    }
    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;
int main() { heap_t a, b;
  a.clear(); a.push(1); a.push(3);
  b.clear(); 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;
  }
  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 = _new(*T);
    if (size(T->l) < x){
      PNN tmp = split(T->r, x - 1 - size(T->l));
      res->r = tmp.F; return {pull(res), tmp.S};
    } else {
      PNN tmp = split(T->l, x);
      res->l = tmp.S; return {tmp.F, pull(res)};
    }
//====================
    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)};
    }
//>>>>>>>>>>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))
      res = _new(*T1), res->r = merge(T1->r, T2);
    else
      res = _new(*T2), res->l = merge(T1, T2->l);
    return pull(res);
//====================
    if (rand() % (size(T1) + size(T2)) < size(T1))
      return T1->r = merge(push(T1)->r, T2), pull(T1);
    else
      return T2->l = merge(T1, push(T2)->l), pull(T2);
//>>>>>>>>>>ORIGIN
  }
};
```

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
  }
};
```

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

4.2 CutBridge

```
struct Graph { int V, stamp;
  struct Node : vector<Node*> {
    int low, dfn; bool is_cut; Node *pa;
    Node() { low = dfn = -1;
      is_cut = false; pa = NULL;
    }
  }_memN[MAXN], *node[MAXN];
  Graph (int V) : V(V) { stamp = 0;
    for (int i = 0 ; i < V ; i++)
      node[i] = _memN + i;
  }
  void addEdge(int u, int v) {
    node[u]->push_back(node[v]);
    node[v]->push_back(node[u]);
  }
  void Tarjan(Node *u, Node *pa) {
    u->pa = pa; u->dfn = u->low = ++stamp;
    for (auto &v : *u) if (!~v->dfn)
      Tarjan(v, u), u->low = min(u->low, v->low);
    else if (pa != v)
      u->low = min(u->low, v->dfn);
  }
  void CutBridge() { int rt_son = 0;
    Tarjan(node[0], NULL);
    for (int i = 1 ; i < V ; i++) {
      Node *pa = node[i]->pa;
      if (pa == node[0]) rt_son++;
      else if (node[i]->low >= pa->dfn)
        pa->is_cut = true;
    }
    if (rt_son > 1) node[0]->is_cut = true;
    for (int i = 0 ; i < V ; i++)
      if (node[i]->is_cut)
        /* node[i] is a cut */;
    for (int i = 0 ; i < V ; i++) {
      Node *pa = node[i]->pa;
      if (pa && node[i]->low > pa->dfn)
        /* pa and node[i] is a bridge */;
    }
  }
};
```

4.3 Dijkstra

```
typedef struct Edge { int v; LL w;
  bool operator > (const Edge &b) const {
    return w > b.w;
  }
} S;
vector<LL> Dijkstra(vector<vector<Edge> > &G, int s) {
  priority_queue<S, vector<S>, greater<S> > pq;
  vector<LL> d(G.size(), INF);
  d[s] = 0; pq.push({s, d[s]});
  while (pq.size()) {
    auto p = pq.top(); pq.pop();
    if (d[p.v] < p.w) continue;
    for (auto &e : G[p.v]) {
      if (d[e.v] > d[p.v] + e.w) {
        d[e.v] = d[p.v] + e.w;
        pq.push({e.v, d[e.v]});
      }
    }
  }
  return d;
}
```

4.4 Blossom

```
struct Graph { struct Edge; int V;
  struct Node : vector<Edge*> {
    Node *p, *s, *m; int S, v;
    Node() : S(-1), v(-1) { p = s = m = NULL; }
  }_memN[MAXN], *node[MAXN];
  struct Edge { Node *v;
    Edge(Node *v = NULL) : v(v) {}
  }_memE[MAXM], *ptrE;
  Graph (int V) : V(V) { ptrE = _memE;
    for (int i = 0 ; i < V ; i++) node[i] = _memN + i;
  }
  void addEdge(int u, int v) {
    node[u]->push_back(new (ptrE++) Edge(node[v]));
    node[v]->push_back(new (ptrE++) Edge(node[u]));
  }
  inline int maxMatch() { int ans = 0;
    for (int i = 0 ; i < V ; i++)
      if (!node[i]->m && bfs(node[i])) ans++;
    return ans;
  }
  inline bool bfs(Node *u) {
    for (int i = 0 ; i < V ; i++)
      node[i]->s = node[i], node[i]->S = -1;
    queue<Node*> q; q.push(u), u->S = 0;
    while (q.size()) { u = q.front(); q.pop();
      for (auto &e : *u) { Node *v = e->v;
        if (!~v->S) { v->p = u; v->S = 1;
          if (!v->m) return augment(u, v);
          q.push(v->m); v->m->S = 0;
        } else if (!v->S && v->s != u->s) {
          Node *l = LCA(v->s, u->s);
          flower(v, u, l, q); flower(u, v, l, q);
        }
      }
    }
    return false;
  }
  inline bool augment(Node *u, Node *v) {
    for (Node *l; u; v = l, u = v ? v->p : NULL)
      l = u->m, u->m = v, v->m = u;
    return true;
  }
  inline Node* LCA(Node *u, Node *v) {
    static int t = 0;
    for (++t ; ; swap(u, v)) {
      if (!u) continue; if (u->v == t) return u;
      u->v = t; u = u->m; if (!u) continue;
      u = u->p; if (!u) continue; u = u->s;
    }
  }
  inline void flower(Node *u, Node *v, Node *l, queue<
      Node*> &q) {
    while (u->s != l) { u->p = v; v = u->m;
      if (v->S == 1) q.push(v), v->S = 0;
      u->s = v->s = l; u = v->p;
    }
  }
};
```

5 Geometry

5.1 Point

```
using T = double;
const T EPS = 1e-8;
#define O operator
#define CP (const P &p) const
typedef struct P { T x, y;
  P(T x = 0, T y = 0) : x(x), y(y) {}
  P O+CP { return P(x + p.x, y + p.y); }
  P O-CP { return P(x - p.x, y - p.y); }
  T O*CP { return x * p.x + y * p.y; }
  T O%CP { return x * p.y - y * p.x; }
  P O*(const T c) const { return P(x * c, y * c); }
  P O/(const T c) const { return P(x / c, y / c); }
} V;
T leng2(V a) { return a * a; }
T leng(V a) { return sqrt(leng2(a)); }
T dist2(P a, P b) { return leng2(a - b); }
T dist(P a, P b) { return sqrt(dist2(a, b)); }
```

5.2 Line

```
struct L { P p1, p2; L() {}
    L(P p1, P p2) : p1(p1), p2(p2) {}
};
T dist(P p, L l) {
  V v1 = p - l.p1, v2 = l.p2 - l.p1;
  return abs(v1 % v2) / leng(v2);
}
T dist(L l1, L l2) {
  V v1 = l1.p2 - l1.p1; V v2 = l2.p2 - l2.p1;
  if (v1 % v2 != 0) return 0;
  return dist(l1.p1, l2);
}
int dir(P p, L l) {
  return (l.p2 - l.p1) % (p - l.p1);
}
bool parallel(L l1, L l2) {
  return abs((l1.p1 - l1.p2) % (l2.p1 - l2.p2)) <= EPS
      ;
}
int intersect(L l1, L l2) {
  // -1: infinity, 1: one, 0: none
  return parallel(l1, l2) ?
      (abs(dir(l1.p2, l2)) <= EPS ? -1 : 0) : 1;
}
```

5.3 Segment

```
struct S { P p1, p2; S() {}
  S(P p1, P p2) : p1(p1), p2(p2) {}
};
T dist(P p, S s) {
  V v = s.p2 - s.p1, v1 = p - s.p1, v2 = p - s.p2;
  if (v * v1 < -EPS) return leng(v1);
  if (v * v2 > EPS) return leng(v2);
  return abs(v1 % v2) / leng(v);
}
T btw(P p, S s) { return (s.p1 - p) * (s.p2 - p); }
T dir(P p, S s) { return dir(p, L{s.p1, s.p2}); }
bool intersect(P p, S s) {
```

```
  return abs(dir(p, s)) < EPS && btw(p, s) <= EPS;
}
bool intersection(S s1, S s2) {
  T c1 = dir(s2.p1, s1), c2 = dir(s2.p2, s1);
  T c3 = dir(s1.p1, s2), c4 = dir(s1.p2, s2);
  if (c1 * c2 < -EPS && c3 * c4 < -EPS) return true;
  if (abs(c1) < EPS && intersect(s2.p1, s1)) return
      true;
  if (abs(c2) < EPS && intersect(s2.p2, s1)) return
      true;
  if (abs(c3) < EPS && intersect(s1.p1, s2)) return
      true;
  if (abs(c4) < EPS && intersect(s1.p2, s2)) return
      true;
  return false;
}
T dist(S s1, S s2) {
  if (intersection(s1, s2)) return 0;
  return min({
    dist(s1.p1, s2), dist(s1.p2, s2),
    dist(s2.p1, s1), dist(s2.p2, s1)
  });
}
```

5.4 Convex

```
bool convexCmp(const P& l, const P& r) {
  return (l.x < r.x) || ((l.x == r.x) && (l.y < r.y));
}
T cross(const P &o, const P &a, const P &b) {
  return (a - o) % (b - o);
}
vector<P> convexHull(vector<P> &ps) {
  sort(ps.begin(), ps.end(), convexCmp);
  vector<P> stk, ret;
  for (auto &p : ps) {
    while (stk.size() >= 2 && cross(stk[stk.size() -
        2], stk.back(), p) <= 0)
      stk.pop_back();
    stk.push_back(p);
  }
  stk.pop_back();
  for (auto &p : stk) ret.push_back(p);
  reverse(ps.begin(), ps.end()); stk.clear();
  for (auto &p : ps) {
    while (stk.size() >= 2 && cross(stk[stk.size() -
        2], stk.back(), p) <= 0)
      stk.pop_back();
    stk.push_back(p);
  }
  for (auto &p : stk) ret.push_back(p);
  return ret;
}
```