Contents

1 Flow

1.1 Dinic

```
struct Dinic {
  int level[maxn], n, s, t;
  struct Edge {
    int to, rev, cap;
    Edge() {}
    Edge(int a, int b, int c): to(a), cap(b), rev(c) {}
  };
  vector<Edge> G[maxn];
  bool bfs() {
    memset(level, -1, sizeof(level));
    level[s] = 0;
    queue<int> que; que.push(s);
    while (que.size()) {
      int tmp = que.front(); que.pop();
      for (auto e : G[tmp]) {
        if (e.cap > 0 && level[e.to] == -1) {
          level[e.to] = level[tmp] + 1;
          que.push(e.to);
        }
      }
    }
    return level[t] != -1;
  }
  int flow(int now, int low) {
    if (now == t) return low;
    int ret = 0;
    for (auto &e : G[now]) {
      if (e.cap > 0 && level[e.to] == level[now] + 1) {
        int tmp = flow(e.to, min(e.cap, low - ret));
        e.cap -= tmp; G[e.to][e.rev].cap += tmp;
        ret += tmp;
      }
    }
    if (ret == 0) level[now] = -1;
    return ret;
  }
  Dinic(int _n, int _s, int _t): n(_n), s(_s), t(_t) {
    fill(G, G + maxn, vector<Edge>());
  }
  void add_edge(int a, int b, int c) {
    G[a].push_back(Edge(b, c, G[b].size()));
    G[b].push_back(Edge(a, 0, G[a].size() - 1));
  }
  int maxflow() {
    int ret = 0;
    while (bfs()) ret += flow(s, inf);
    return ret;
  }
};
```

1.2 MinCostMaxFlow

```
struct MincostMaxflow {
  struct Edge {
    int to, rev, cap, w;
    Edge() {}
    Edge(int a, int b, int c, int d): to(a), cap(b), w(
    c), rev(d) {}
  };
  int n, s, t, p[maxn], id[maxn];
  int d[maxn];
  bool inque[maxn];
  vector<Edge> G[maxn];
  pair<int, int> spfa() {
    memset(p, -1, sizeof(-1));
    fill(d, d + maxn, inf);
    memset(id, -1, sizeof(id));
    d[s] = 0; p[s] = s;
    queue<int> que; que.push(s); inque[s] = true;
    while (que.size()) {
      int tmp = que.front(); que.pop();
      inque[tmp] = false;
      int i = 0;
      for (auto e : G[tmp]) {
```

```
        if (e.cap > 0 && d[e.to] > d[tmp] + e.w) {
          d[e.to] = d[tmp] + e.w;
          p[e.to] = tmp;
          id[e.to] = i;
          if (!inque[e.to]) que.push(e.to), inque[e.to]
     = true;
        }
        ++i;
      }
    }
    if (d[t] == inf) return make_pair(-1, -1);
    int a = inf;
    for (int i = t; i != s; i = p[i]) {
      a = min(a, G[p[i]][id[i]].cap);
    }
    for (int i = t; i != s; i = p[i]) {
      Edge &e = G[p[i]][id[i]];
      e.cap -= a; G[e.to][e.rev].cap += a;
    }
    return make_pair(a, d[t]);
  }
  MincostMaxflow(int _n, int _s, int _t): n(_n), s(_s),
     t(_t) {
    fill(G, G + maxn, vector<Edge>());
  }
  void add_edge(int a, int b, int cap, int w) {
    G[a].push_back(Edge(b, cap, w, (int)G[b].size()));
    G[b].push_back(Edge(a, 0, -w, (int)G[a].size() - 1)
    );
  }
  pair<int, int> maxflow() {
    int mxf = 0, mnc = 0;
    while (true) {
      pair<int, int> res = spfa();
      if (res.first == -1) break;
      mxf += res.first; mnc += res.first * res.second;
    }
    return make_pair(mxf, mnc);
  }
};
```

1.3 Maximum weighted Bipartite matching

```
struct Hungarian {
  int w[maxn][maxn], lx[maxn], ly[maxn];
  int match[maxn], n;
  bool s[maxn], t[maxn];
  bool dfs(int now) {
    s[now] = true;
    for (int i = 0; i < n; ++i) {
      if (lx[now] + ly[i] == w[now][i] && !t[i]) {
        t[i] = true;
        if (match[i] == -1 || dfs(match[i])) {
          match[i] = now;
          return true;
        }
      }
    }
    return false;
  }
  void relabel() {
    int a = inf;
    for (int i = 0; i < n; ++i) if (s[i]) {
      for (int j = 0; j < n; ++j) if (!t[j]) {
        a = min(a, lx[i] + ly[j] - w[i][j]);
      }
    }
    for (int i = 0; i < n; ++i) {
      if (s[i]) lx[i] -= a;
      if (t[i]) ly[i] += a;
    }
  }
  Hungarian(int n): n(n) {
    memset(w, 0, sizeof(w));
    memset(lx, 0, sizeof(lx));
    memset(ly, 0, sizeof(ly));
    memset(match, -1, sizeof(match));
  }
  void add_edge(int a, int b, int c) {
    w[a][b] = c;
  }
  int solve() {
    for (int i = 0; i < n; ++i) for (int j = 0; j < n;
    ++j) lx[i] = max(lx[i], w[i][j]);
    for (int i = 0; i < n; ++i) {
      while (true) {
        memset(s, false, sizeof(s)); memset(t, false,
    sizeof(t));
        if (dfs(i)) break;
        else relabel();
      }
    }
    int ans = 0;
    for (int i = 0; i < n; ++i) ans += w[match[i]][i];
    return ans;
  }
};
```

2 Math

2.1 FFT

```
const double pi = acos(-1);
const complex<double> I(0, 1);
complex<double> omega[maxn + 1];

void prefft() {
  for (int i = 0; i <= maxn; ++i) omega[i] = exp(i * 2
     * pi / maxn * I);
}
void fft(vector<complex<double>>& a, int n, bool inv=
    false) {
  int basic = maxn / n;
  int theta = basic;
  for (int m = n; m >= 2; m >>= 1) {
    int h = m >> 1;
    for (int i = 0; i < h; ++i) {
      complex<double> w = omega[inv ? maxn - (i * theta
     % maxn) : i * theta % maxn];
      for (int j = i; j < n; j += m) {
        int k = j + h;
        complex<double> x = a[j] - a[k];
        a[j] += a[k];
        a[k] = w * x;
      }
    }
    theta = (theta * 2) % maxn;
  }
  int i = 0;
  for (int j = 1; j < n - 1; ++j) {
    for (int k = n >> 1; k > (i ^= k); k >>= 1);
    if (j < i) swap(a[i], a[j]);
  }
  if (inv) for (int i = 0; i < n; ++i) a[i] /= (double)
    n;
}
void invfft(vector<complex<double>>& a, int n) {
  fft(a, n, true);
}
```

2.2 Miller-Rabin

```
// n < 4759123141    chk = [2, 7, 61]
// n < 1122004669633 chk = [2, 13, 23, 1662803]
// n < 2^64      chk = [2, 325, 9375, 28178, 450775,
    9780504, 1795265022]

long long fpow(long long a, long long n, long long mod)
     {
  long long ret = 1LL;
  for (; n; n >>= 1) {
    if (n & 1) ret = (__int128)ret * (__int128)a % mod;
    a = (__int128)a * (__int128)a % mod;
  }
  return ret;
}
```

```
bool check(long long a, long long u, long long n, int t
    ) {
  a = fpow(a, u, n);
  if (a == 0) return true;
  if (a == 1 || a == n - 1) return true;
  for (int i = 0; i < t; ++i) {
    a = (__int128)a * (__int128)a % n;
    if (a == 1) return false;
    if (a == n - 1) return true;
  }
  return false;
}
bool is_prime(long long n) {
  if (n < 2) return false;
  if (n % 2 == 0) return n == 2;
  long long u = n - 1; int t = 0;
  for (; u & 1; u >>= 1, ++t);
  for (long long i : chk) {
    if (!check(i, u, n, t)) return false;
  }
  return true;
}
```

2.3 Extend GCD

```
template <typename T> tuple<T, T, T> extgcd(T a, T b) {
  if (!b) return make_tuple(a, 1, 0);
  T d, x, y;
  tie(d, x, y) = extgcd(b, a % b);
  return make_tuple(d, y, x - (a / b) * y);
}
```

2.4 Matrix

```
template <typename T> class Matrix {
  public:
    int n, m, mod;
    vector<vector<T>> mat;
    Matrix(int n, int m, int mod=0, bool I=false): n(n)
    , m(m), mod(mod) {
      mat.resize(n);
      for (int i = 0; i < n; ++i) mat[i].resize(m);
      if (!I) return;
      for (int i = 0; i < n; ++i) mat[i][i] = 1;
    }
    Matrix operator+(const Matrix& rhs) const {
      Matrix ret(n, m, mod);
      for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
          ret.mat[i][j] = mat[i][j] + rhs.mat[i][j];
          if (mod) ret.mat[i][j] %= mod;
        }
      }
      return ret;
    }
    Matrix operator-(const Matrix& rhs) const {
      Matrix ret(n, m, mod);
      for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
          ret.mat[i][j] = mat[i][j] - rhs.mat[i][j];
          if (mod) {
            ret.mat[i][j] %= mod;
            ret.mat[i][j] += mod;
            ret.mat[i][j] %= mod;
          }
        }
      }
      return ret;
    }
    Matrix operator*(const Matrix& rhs) const {
      Matrix ret(n, rhs.m, mod);
      for (int i = 0; i < n; ++i) {
        for (int j = 0; j < rhs.m; ++j) {
          for (int k = 0; k < m; ++k) {
            if (mod) ret.mat[i][j] = (ret.mat[i][j] +
    mat[i][k] * rhs.mat[k][j] % mod) % mod;
            else ret.mat[i][j] += mat[i][k] * rhs.mat[k
    ][j];
          }
        }
      }
      return ret;
    }
};
```

3 Graph

3.1 Strongly connected components

```
struct SCC {
  vector<int> G[maxn], R[maxn], topo;
  int n, nscc, scc[maxn], sz[maxn];
  bool v[maxn];
  void dfs(int now) {
    v[now] = true;
    scc[now] = nscc;
    ++sz[nscc];
    for (int u : G[now]) if (!v[u]) {
      dfs(u);
    }
  }
  void rdfs(int now) {
    v[now] = true;
    for (int u : R[now]) if (!v[u]) {
      rdfs(u);
    }
    topo.push_back(now);
  }
  SCC(int n): n(n) {}
  void add_edge(int a, int b) {
    G[a].push_back(b);
    R[b].push_back(a);
  }
  void solve() {
    memset(v, false, sizeof(v));
    for (int i = 0; i < n; ++i) if (!v[i]) rdfs(i);
    reverse(topo.begin(), topo.end());
    memset(v, false, sizeof(v));
    for (int i : topo) if (!v[i]) {
      ++nscc;
      dfs(i);
    }
  }
};
```

3.2 Heavy-Light Decomposition

```
struct HeavyLightDecomp {
  vector<int> G[maxn];
  int tin[maxn], top[maxn], dep[maxn], maxson[maxn], sz
    [maxn], p[maxn], n, clk;
  void dfs(int now, int fa, int d) {
    dep[now] = d;
    maxson[now] = -1;
    sz[now] = 1;
    p[now] = fa;
    for (int u : G[now]) if (u != fa) {
      dfs(u, now, d + 1);
      sz[now] += sz[u];
      if (maxson[now] == -1 || sz[u] > sz[maxson[now]])
     maxson[now] = u;
    }
  }
  void link(int now, int t) {
    top[now] = t;
    tin[now] = ++clk;
    if (maxson[now] == -1) return;
    link(maxson[now], t);
    for (int u : G[now]) if (u != p[now]) {
      if (u == maxson[now]) continue;
      link(u, u);
    }
  }
  HeavyLightDecomp(int n): n(n) {
```

```
    clk = 0;
    memset(tin, 0, sizeof(tin)); memset(top, 0, sizeof(
    top)); memset(dep, 0, sizeof(dep));
    memset(maxson, 0, sizeof(maxson)); memset(sz, 0,
    sizeof(sz)); memset(p, 0, sizeof(p));
  }
  void add_edge(int a, int b) {
    G[a].push_back(b);
    G[b].push_back(a);
  }
  void solve() {
    dfs(0, -1, 0);
    link(0, 0);
  }
  int lca(int a, int b) {
    int ta = top[a], tb = top[b];
    while (ta != tb) {
      if (dep[ta] < dep[tb]) {
        swap(ta, tb); swap(a, b);
      }
      a = p[ta]; ta = top[a];
    }
    if (a == b) return a;
    return dep[a] < dep[b] ? a : b;
  }
  vector<pair<int, int>> get_path(int a, int b) {
    int ta = top[a], tb = top[b];
    vector<pair<int, int>> ret;
    while (ta != tb) {
      if (dep[ta] < dep[tb]) {
        swap(ta, tb); swap(a, b);
      }
      ret.push_back(make_pair(tin[ta], tin[a]));
      a = p[ta]; ta = top[a];
    }
    ret.push_back(make_pair(min(tin[a], tin[b]), max(
    tin[a], tin[b])));
    return ret;
  }
};
```

3.3 2-Satisfiability

```
class TwoSat {
  private:
    vector<int> G[maxn << 1];
    bool v[maxn << 1];
    int s[maxn << 1], c;
    bool dfs(int now) {
      if (v[now ^ 1]) return false;
      if (v[now]) return true;
      v[now] = true;
      s[c++] = now;
      for (int u : G[now]) if (!dfs(u)) return false;
      return true;
    }
  public:
    void add_edge(int a, int b) {
      G[a].push_back(b);
    }
    bool solve() {
      for (int i = 0; i < maxn << 1; i += 2) {
        if (!v[i] && !v[i + 1]) {
          c = 0;
          if (!dfs(i)) {
            while (c) v[s[--c]] = false;
            if (!dfs(i + 1)) return false;
          }
        }
      }
      return true;
    }
};
```

4 Data Structures

4.1 Treap

```
struct Treap {
#define size(t) (t ? t->sz : 0)
  struct Node {
    int val;
    int pri, sz;
    Node *lc, *rc;
    Node(T v): pri(rand()), val(v) {
      lc = rc = nullptr;
      sz = 1;
    }
    void pull() {
      sz = size(lc) + size(rc) + 1;
    }
  } *root;
  Node *merge(Node *a, Node *b) {
    if (!a || !b) return a ? a : b;
    if (a->pri > b->pri) {
      a->rc = merge(a->rc, b);
      a->pull();
      return a;
    } else {
      b->lc = merge(a, b->lc);
      b->pull();
      return b;
    }
  }
  void split(Node *t, int k, Node *&a, Node *&b) {
    if (!t) { a = b = nullptr; return; }
    if (t->val <= k) {
      a = t;
      split(t->rc, k, a->rc, b);
      a->pull();
    } else {
      b = t;
      split(t->lc, k, a, b->lc);
      b->pull();
    }
  }
  int kth(Node *t, int k) {
    if (size(t->lc) + 1 == k) return t->val;
    if (size(t->lc) + 1 > k) return kth(t->lc, k);
    return kth(t->rc, k - size(t->lc) - 1);
  }
  void clear(Node *t) {
    if (!t) return;
    if (t->lc) clear(t->lc);
    if (t->rc) clear(t->rc);
    delete t;
  }
  Treap(unsigned seed=time(nullptr)) {
    srand(seed);
    root = nullptr;
  }
  ~Treap() {
    clear(root);
    root = nullptr;
  }
  void insert(int val) {
    Node *a, *b;
    split(root, val - 1, a, b);
    root = merge(merge(a, new Node(val)), b);
  }
  void erase(int val) {
    Node *a, *b, *c, *d;
    split(root, val - 1, a, b);
    split(b, val, c, d);
    c = merge(c->lc, c->rc);
    root = merge(a, merge(c, d));
  }
  int find(int k) {
    return kth(root, k);
  }
#undef size
};
```

4.2 Leftlist Tree

```
template <typename T> class LeftlistTree {
  private:
#define rank(t) (t ? t->s : 0)
    struct Node {
      T val;
      int s;
      Node *lc, *rc;
      Node(T v): val(v) {
        lc = rc = nullptr;
        s = 1;
      }
    } *root;
    Node *merge(Node *a, Node *b) {
      if (!a || !b) return a ? a : b;
      if (a->val < b->val) swap(a, b);
      a->rc = merge(a->rc, b);
      if (rank(a->lc) < rank(a->rc)) swap(a->lc, a->rc)
    ;
      a->s = rank(a->rc) + 1;
      return a;
    }
    void clear(Node *t) {
      if (!t) return;
      if (t->lc) clear(t->lc);
      if (t->rc) clear(t->rc);
      delete t;
    }
  public:
    LeftlistTree() {
      root = nullptr;
    }
    void push(T val) {
      root = merge(root, new Node(val));
    }
    void pop() {
      T ret = root->val;
      Node *tmp = root;
      root = merge(root->lc, root->rc);
      delete tmp;
    }
    T top() {
      return root->val;
    }
    void merge(LeftlistTree t) {
      root = merge(root, t->root);
    }
};
```

5 Geometry

5.1 Points

```
struct pt {
  double x, y;
  pt(): x(0.0), y(0.0) {}
  pt(double x, double y): x(x), y(y) {}
  pt operator+(const pt& a) const { return pt(x + a.x,
    y + a.y); }
  pt operator-(const pt& a) const { return pt(x - a.x,
    y - a.y); }
  double operator*(const pt& a) const { return x * a.x
    + y * a.y; }
  double operator^(const pt& a) const { return x * a.y
    - y * a.x; }
  bool operator<(const pt& a) const { return x == a.x ?
     y < a.y : x < a.x; }
};
```

5.2 Convex Hull

```
double cross(const pt& o, const pt& a, const pt& b) {
  return (a - o) ^ (b - o);
}
int rsd;

vector<pt> convex_hull(vector<pt> p) {
  sort(p.begin(), p.end());
  int m = 0;
  vector<pt> ret(2 * p.size());
  for (int i = 0; i < p.size(); ++i) {
    while (m >= 2 && cross(ret[m - 2], ret[m - 1], p[i
    ]) < 0) --m;
    ret[m++] = p[i];
  }
  rsd = m - 1;
  for (int i = p.size() - 2, t = m + 1; i >= 0; --i) {
    while (m >= t && cross(ret[m - 2], ret[m - 1], p[i
    ]) < 0) --m;
    ret[m++] = p[i];
  }
  ret.resize(m - 1);
  return ret;
}
```

5.3 Rotating Caliper

```
void rotating_caliper(vector<pt> p) {
  vector<pt> ch = convex_hull(p);
  int tbz = ch.size();
  int lpr = 0, rpr = rsd;
  // ch[lpr], ch[rpr]
  while (lpr < rsd || rpr < tbz - 1) {
    if (lpr < rsd && rpr < tbz - 1) {
      pt rvt = ch[rpr + 1] - ch[rpr];
      pt lvt = ch[lpr + 1] - ch[lpr];
      if ((lvt ^ rvt) < 0) ++lpr;
      else ++rpr;
    }
    else if (lpr == rsd) ++rpr;
    else ++lpr;
    // ch[lpr], ch[rpr]
  }
}
```

6 String

6.1 KMP

```
int f[maxn];

int kmp(const string& a, const string& b) {
  f[0] = -1; f[1] = 0;
  for (int i = 1, j = 0; i < b.size() - 1; f[++i] = ++j
    ) {
    if (b[i] == b[j]) f[i] = f[j];
    while (j != -1 && b[i] != b[j]) j = f[j];
  }
  for (int i = 0, j = 0; i - j + b.size() <= a.size();
    ++i, ++j) {
    while (j != -1 && a[i] != b[j]) j = f[j];
    if (j == b.size() - 1) return i - j;
  }
  return -1;
}
```

6.2 Suffix Array

```
struct SuffixArray {
  int sa[maxn], tmp[2][maxn], c[maxn], _lcp[maxn], r[
    maxn], n;
  string s;
  SparseTable st;
  void suffixarray() {
     int* rank = tmp[0];
    int* nRank = tmp[1];
    int A = 128;
    for (int i = 0; i < A; ++i) c[i] = 0;
```

```
    for (int i = 0; i < s.length(); ++i) c[rank[i] = s[
    i]]++;
    for (int i = 1; i < A; ++i) c[i] += c[i - 1];
    for (int i = s.length() - 1; i >= 0; --i) sa[--c[s[
    i]]] = i;
    for (int n = 1; n < s.length(); n *= 2) {
      for (int i = 0; i < A; ++i) c[i] = 0;
      for (int i = 0; i < s.length(); ++i) c[rank[i
    ]]++;
      for (int i = 1; i < A; ++i) c[i] += c[i - 1];
      int* sa2 = nRank;
      int r = 0;
      for (int i = s.length() - n; i < s.length(); ++i)
     sa2[r++] = i;
      for (int i = 0; i < s.length(); ++i) if (sa[i] >=
     n) sa2[r++] = sa[i] - n;
      for (int i = s.length() - 1; i >= 0; --i) sa[--c[
    rank[sa2[i]]]] = sa2[i];
      nRank[sa[0]] = r = 0;
      for (int i = 1; i < s.length(); ++i) {
        if (!(rank[sa[i - 1]] == rank[sa[i]] && sa[i -
    1] + n < s.length() && rank[sa[i - 1] + n] == rank[
    sa[i] + n])) r++;
        nRank[sa[i]] = r;
      }
      swap(rank, nRank);
      if (r == s.length() - 1) break;
      A = r + 1;
    }
  }
  void solve() {
    suffixarray();
    for (int i = 0; i < n; ++i) r[sa[i]] = i;
    int ind = 0; _lcp[0] = 0;
    for (int i = 0; i < n; ++i) {
      if (!r[i]) { ind = 0; continue; }
      while (i + ind < n && s[i + ind] == s[sa[r[i] -
    1] + ind]) ++ind;
      _lcp[r[i]] = ind ? ind-- : 0;
    }
    st = SparseTable(n, _lcp);
  }
  int lcp(int L, int R) {
    if (L == R) return n - L - 1;
    L = r[L]; R = r[R];
    if (L > R) swap(L, R);
    ++L;
    return st.query(L, R);
  }
  SuffixArray(string s): s(s), n(s.length()) {}
  SuffixArray() {}
};
```