0 Contents

1 Flow

1.1 Dinic

```
#define INF 0x3f3f3f3f
#define LINF 0x3f3f3f3f3f3f3f3fLL
#include <vector>
#include <queue>
struct Dinic
{
  typedef long long int T;
  struct edge{
    int u, v;
    T c, f;
    edge(int _u, int _v, T _c, T _f): u(_u),v(_v),c(_c)
        ,f(_f){}
  };
  int n, s, t;
  vector<vector<int> > G;
  vector<edge> E;
  vector<int> cur, vis, d;
  Dinic(int _n):n(_n){
    G.resize(n+1);
    vis.resize(n+1); cur.resize(n+1); d.resize(n+1);
    for(int i=0; i<=n; i++)d[i] = INF;
  }
  void pb(int u, int v, T cap){
    G[u].push_back(E.size());
    E.push_back(edge(u, v, cap, 0));
    G[v].push_back(E.size());
    E.push_back(edge(v, u, 0, 0));
  }
  int bfs() {
    queue<int> q;
    for(int i=0; i<=n; i++)vis[i] = 0;
    q.push(s); d[s] = 0;
    while(!q.empty()) {
      int u = q.front(); q.pop();
      vis[u] = 1;
      for(int i=0; i<(int)G[u].size(); i++) {
        edge e = E[G[u][i]];
        if(e.c - e.f > 0 && !vis[e.v]) {
          d[e.v] = d[u] + 1;
          q.push(e.v);
        }
      }
    }
    return vis[t];
  }
  T dfs(int u, T a) {
    if(u == t || !a)return a;
    T totf = 0, f;
    for(int &i=cur[u]; i<(int)G[u].size(); i++) {
      edge &e = E[G[u][i]], &r=E[G[u][i]^1];
      if(d[e.v] != d[u]+1)continue;
      f = dfs(e.v, min(a, e.c - e.f));
      if(f<=0)continue;
      e.f += f; r.f -= f;
      totf += f;
      a -= f; if(!a)break;
    }
    return totf;
  }

  T operator()(int _s, int _t) {
    s = _s, t = _t;
    T maxf = 0;
    while(bfs()) {
      for(int i=0; i<=n; i++)cur[i] = 0;
      maxf += dfs(s, LINF);
    }
    return maxf;
  }
};
```

1.2 Min Cost Flow

```
#include <vector>
#include <queue>
#define ll long long int
```

```
#define LINF 2147483647000000LL
#define INF 2147483647
using namespace std;
struct MCF {
  struct edge
  {
    int u, v, c, f;
    ll co;
    edge(int _u, int _v, int _c, ll _co){ u = _u, v =
        _v, c = _c; co = _co; f = 0; }
  };
  vector<vector<int> > G;
  vector<edge> E;
  vector<ll> d;
  vector<int> inq, arg, p;
  int N, s, t;
  MCF(int _n) {
    N = _n;
    G.resize(_n+1);
    d.resize(_n+1); inq.resize(_n+1);
    arg.resize(_n+1); p.resize(_n+1);
    E.clear();
  }
  void pb(int u, int v, int c, ll co) {
    G[u].push_back(E.size());
    E.push_back(edge(u, v, c, co));
    G[v].push_back(E.size());
    E.push_back(edge(v, u, 0, -co));
  }
  bool BF(int &flow, ll &cost) {
    for(int i=0;i<=N;i++)p[i] = 0, inq[i] = 0, d[i] =
        LINF;
    queue<int> Q;
    Q.push(s);
    d[s]=0; inq[s] = 1; arg[s] = INF;
    while(!Q.empty()) {
      int x=Q.front(); Q.pop(); inq[x] = 0;
      for(int i=0; i<(int)G[x].size(); i++) {
        edge &e=E[G[x][i]];
        if(d[x] + e.co < d[e.v] && e.c > e.f) {
          d[e.v] = d[x]+e.co;
          p[e.v] = G[x][i];
          arg[e.v] = min(arg[x], e.c - e.f);
          if(!inq[e.v])Q.push(e.v), inq[e.v] = 1;
        }
      }
    }
    if(d[t] == LINF)return 0;
    int a = arg[t];
    for(int now = t; now != s; now = E[p[now]].u) {
      E[p[now]].f += a;
      E[p[now]^1].f -= a;
    }
    cost += arg[t] * d[t];
    flow += a;
    return 1;
  }

  pair<int, ll> operator ()(int _s, int _t) {
    s = _s, t = _t;
    int flow=0;
    ll cost=0;
    while(BF(flow, cost)){}
    return pair<int, ll>(flow, cost);
  }
};
```

1.3 Common Modeling Technique

Minimum Path Covering on DAG

1. Path covering without path intersection: For each vertex v, we may construct two vertices v_i and v_o, then for each edge $u \to v$, connect $u_o \to v_i$.
   This forms a bipartite graph. Each selected edge means a "join" of paths. Therefore the cardinality of the minimum path covering on the original graph will be $|V| - m$, where m is the cardinality of the maximum bipartite matching.
2. Covering that allows intersection: Perform Floyd-Warshall to obtain trasitive closure first, then make edge for each pair that are connected, the problem subsequently reduces to the non-intersecting case.

2 Math

2.1 ExtGCD

```
typedef long long int ll;
#define mod 1000000007
void gcd(ll a, ll b, ll &x, ll &y, ll &d) {
  if(!b){ x = 1; y = 0; d = a; return ; }
  gcd(b, a%b, y, x, d); y -= (a/b)*x;
}
ll inv(ll a) {
  ll x, y, d;
  gcd(a, mod, x, y, d);
  return d==1 ? (x+mod)%mod : 0;
}
```

2.2 FFT

1. When convert back to integer, use `LL` can be safer.
2. `eps` are `0.5` generally, but sometime need adjustments.
3. the array `A` and `B` will be changed after DFT, and the result `AB` has been devided by `_n`.

```
#include <stdlib.h>
#include <math.h>
#include <complex>
#include <string.h>
#define MAXN 1048576
#define eps 0.5
#define PI
     3.14159265358979323846264338327950288419716939937510
#define max(a,b) (((a) > (b)) ? (a) : (b))

typedef std::complex<double> comp;

struct FFT{
  int _n;
  comp ww[MAXN], rw[MAXN];
  void init(int n, int m){ // n terms in polynomial
    _n=1; while(_n<n+m)_n<<=1;
    ww[0] = rw[0] = comp(1.0, 0.0);
    for(int k=1; k<_n; k++){
      ww[k]=comp(cos(2*k*PI/_n), sin(2*k*PI/_n));
      rw[_n-k]=ww[k];
    }
  }
  int rev(int n,int x){int res=0;while(n){res<<=1;res|=
      x&1;x>>=1;n>>=1;}return res;}
  void dft(int n, comp *res, comp *w){
    for(int i=0; i<n; i++){int j=rev(n>>1,i);if(i<j){
        comp tmp=res[j];res[j]=res[i];res[i]=tmp;}}
    for(int m=1; m<=n; m<<=1){
      if(m==1)continue;
      int mp = m>>1;
      for(int o = 0; o<n; o+=m){
        for(int i=0; i<mp; i++){
          comp tmp = w[i*(n/m)]*res[o+i+mp];
          res[o+i + mp] = res[o+i] - tmp;
          res[o+i] = res[o+i] + tmp;
        }
      }
    }
  }

  void mult(comp *A, comp *B, comp *AB){
    dft(_n, A, ww); dft(_n, B, ww);
    for(int i=0; i<_n; i++)AB[i] = A[i]*B[i];
    dft(_n, AB, rw);
    for(int i=0; i<_n; i++)AB[i]/=_n;
  }
} fft;

comp A[MAXN], B[MAXN];
comp AB[MAXN];
```

3 Graph

3.1 Cut Vertex and BCC

Determining Bridge $low[v] > pre[u] \Rightarrow v$ is a cut vertex and (u, v) is a bridge

```
#include <stack>
#include <queue>
#include <vector>
#define MAXN 1005
using namespace std;
struct edge {
  int u,v;
  edge(int _u,int _v){u=_u;v=_v;}
};
vector<edge> E;
vector<int> G[MAXN];
int N,M;
void pb(int u,int v) {
  G[u].push_back(E.size());
  E.push_back(edge(u,v));
  G[v].push_back(E.size());
  E.push_back(edge(v,u));
}

stack<edge> S;
int pre[MAXN],low[MAXN],bccno[MAXN];
int iscut[MAXN];
int stamp,bcc_cnt;
vector<int> bcc[MAXN];

int dfs(int u,int fa) {
  low[u]=pre[u]=++stamp;
  int ch=0;
  iscut[u]=0;
  for(int i=0;i<(int)G[u].size();i++) {
    edge e=E[G[u][i]];
    int v=e.v;
    if(!pre[v]) {
      ch++;
      S.push(e);
      low[u]=min(low[u],dfs(v,u));
      if(low[v]>=pre[u]) {
        iscut[u]=true;
        bcc_cnt++;
        bcc[bcc_cnt].clear();
        while(1) {
          edge x=S.top();S.pop();
          if(bccno[x.u]!=bcc_cnt)bcc[bcc_cnt].push_back
              (x.u),bccno[x.u]=bcc_cnt;
          if(bccno[x.v]!=bcc_cnt)bcc[bcc_cnt].push_back
              (x.v),bccno[x.v]=bcc_cnt;
          if(x.u==u&&x.v==v)break;
        }
      }
    } else if(pre[v]<pre[u]&&v!=fa) {
      S.push(e);
      low[u]=min(low[u],pre[v]);
    }
  }
  if(fa<0&&ch==1)iscut[u]=false;
  return low[u];
}
```

4 String

4.1 Aho-Corasick Automata

```
#include <map>
#include <queue>
#define MAXN 1000005
template<typename T>
struct AutoAC{
  struct Node {
    int v;
    map<T, Node*> ch;
    typename map<T, Node*>::iterator find(T k){ return
        ch.find(k); }
    typename map<T, Node*>::iterator begin(){ return ch
        .begin(); }
    typename map<T, Node*>::iterator end(){ return ch.
        end(); }
    Node *at(T k){ return ch.at(k); }
    Node *& operator [](T k){ return this->ch[k]; }
    void insert(T k, Node* v){ ch.insert(pair<T, Node
        *>(k, v)); }

    Node *fail;
  } nodes[MAXN];
  int n;
  Node *root;
  Node *newNode(){ nodes[n].v=0; nodes[n].fail=nullptr
      ; nodes[n].ch.clear(); return nodes+(n++); }
  AutoAC() { n=0; root=newNode(); root->v=0; root->fail
      =nullptr; }
  void init() { n=0; root=newNode(); root->v=0; root->
      fail=nullptr; }

  void insert( const T *s , int k ) {
    Node *now = root;
    for(int i=0; s[i]; i++){
      typename map<T, Node*>::iterator it = now->find(s
          [i]);
      if(it == now->end()){
        now->insert(s[i], newNode());
      }
      now = now->at(s[i]);
    }
    now->v = k;
  }
  void buildFail() {
    queue<Node*> q;
    q.push(root);
    while(!q.empty()) {
      Node *x = q.front(); q.pop();
      for(typename map<T, Node*>::iterator it = x->
          begin(); it!=x->end(); it++){
        T next = it->first;
        Node *cur = x->fail;
        while(cur&&cur->find(next) == cur->end())cur =
            cur->fail;
        it->second->fail = cur ? cur->at(next) : root;
        q.push(it->second);
      }
    }
  }
  int search( const T *s ) {
    int res=0;
    Node *cur = root;
    for(int i=0; s[i]; i++){
      while(cur && cur->find(s[i]) == cur->end())cur =
          cur->fail;
      cur = cur ? cur->at(s[i]) : root;
      if(cur->v)cnt[cur->v]++;
      res = max(cnt[cur->v], res);
    }
    return res;
  }
};
```