Contents

1 Setting

1.1 vimrc

2 Math

3 Data Structure

3.1 Idx Tree

```
#include <stdio.h>
#include <algorithm>
#include <vector>
using namespace std;
//Max index Tree
typedef struct IT{
    int n;
    vector<int> mx;
    IT(const vector<int>& a){
        n=a.size();
        mx.resize(n*4);
        init(a,0,n-1,1);
    }

    int init(const vector<int>& a,int l,int r,int n){
        if(l==r)mx[n]=a[l];
        int m=(l+r)/2;
        return mx[n]=max(init(a,l,m,n*2),init(a,m+1,r,n*2+1));
    }
    int quary(int l,int r,int n,int nl,int nr){
        if(r<nl||nr<l)return 0;
        if(l<=nl&&nr<r)return mx[n];
        int m=(l+r)/2;
        return max(quary(l,r,n*2,nl,m),quary(l,r,n*2+1,m+1,nr));
    }

    int quary(int l,int r){
        return quary(l,r,1,0,n-1);
    }

    int update(int i,int x,int n,int nl,int nr){
        if(i<nl&&nr<i)return mx[n];
        if(nl==nr)return mx[n]=x;
        int m = (l+r)/2;
        return mx[n]=max(update(i,x,n*2,nl,m), update(i,x,n*2+1,m+1,nr));
    }

    int update(int i, int x){
        return update(i,x,1,0,n-1);
    }
}IT;
```

4 Graph

4.1 Ford Fulkerson

```
#include <stdio.h>
#include <vector>
#include <algorithm>
using namespace std;

typedef struct edge{
    int to;
    int cost;
    int from;
}edge;

vector<edge> v[11111];
int used[11111];
int flow(int s,int d,int mn){
    if(s==d)return mn;
```

```
    used[s]=1;
    for(int i=0;i<v[s].size();i++){
        edge e=v[s][i];
        if(used[e.to]==0&&e.cost>0){
            int x=flow(e.to,d,min(mn,e.cost));
            if(x>0){
                v[s][i].cost-=x;
                v[e.to][e.from].cost+=x;
                return x;
            }
        }
    }
    return 0;
}

// small n
vecetor<int> v[5010];
int flow[5010][5010];
int ford(int s,int d,int mn){
    if(s==d)return mn;
    used[s]=1;
    for(int i=0;i<v[s].size();i++){
        int e=v[s][i];
        if(used[e]==0&&flow[s][e]>0){
            int x=flow(e,d,min(mn,flow[s][e]));
            if(x>0){
                v[s][e]-=x;
                v[e][s]+=x;
                return x;
            }
        }
    }
    return 0;
}
```

4.2 Edmond Karp

```
const int INF = 987654321;
int n;
int cap[n][n], flow[n][n];

int edmond(int s, int t){
    memset(flow, 0, sizeof(flow));
    int totalflow = 0;
    while(true){
        vector<int> parent(n,-1);
        queue<int> q;
        parent[s]=s;
        q.push(s);
        while(!q.empty()){
            int here = q.front();q.pop();
            for(int there = 0;there < n;there++){
                if(cap[here][here] - flow[here][there] > 0 && parent[there] ==
                   -1){
                    q.push(there);
                    parent[there] = here;
                }
            }
        }
        if(parent[t] == -1)break;
        int amount = INF;
        for(int p=t;p!=s;p=parent[p]){
            amount = min(cap[parent[p]][p] - flow[parent[p]][p], amount);
        }
        for(int p=t;p!=s;p=parent[p]){
            flow[parent[p]][p]+=amount;
            flow[p][parent[p]]-=amount;
        }
        totalflow+=amount;
    }
    return totalflow;
}
```

4.3 Dijkstra

```
#include <stdio.h>
#include <algorithm>
#include <vector>
#include <queue>
using namespace std;
typedef pair<int,int> pii;
const int INF = 2147483647;

int n,m,a,b,c;
int d[1111];
vector<pii> g[1111];

void dijk(vector<pii>* g,int *d,int n,int k){ // start from k
    for(int i=0;i<n;i++){
        d[i]=INF;
    }
    d[k]=0;
    priority_queue<pii,vector<pii>,greater<pii> > pq;
    pq.push(pii(0,k));
    while(!pq.empty()){
        pii p=pq.top();pq.pop();
        int dis=p.first, v=p.second;
        if(d[v]<dis)continue;
        for(int i=0;i<g[v].size();i++){
            pii q=g[v][i];
            int to = q.first, cost = q.second;
            if(d[to] > cost+dis){
                pq.push(pii(d[to] = cost+dis,to));
            }
        }
    }
}
```

4.4 Min-cost Maximum Flow

```
#include <functional>
#include <queue>
#include <limits>
#include <vector>
#include <algorithm>

using namespace std;
// from KCM1700/algorithms

// precondition: there is no negative cycle.
// usage:
//  MinCostFlow mcf(n);
//  for(each edges) mcf.addEdge(from, to, cost, capacity);
//  mcf.solve(source, sink); // min cost max flow
//  mcf.solve(source, sink, 0); // min cost flow
//  mcf.solve(source, sink, goal_flow); // min cost flow with total_flow >=
  goal_flow if possible
struct MinCostFlow
{
    typedef int cap_t;
    typedef int cost_t;

    bool iszerocap(cap_t cap) { return cap == 0; }

    struct edge {
        int target;
        cost_t cost;
        cap_t residual_capacity;
        cap_t orig_capacity;
        size_t revid;
    };

    int n;
    vector<vector<edge>> graph;
    vector<cost_t> pi;
    bool needNormalize, ranbefore;
    int lastStart;

    MinCostFlow(int n) : graph(n), n(n), pi(n, 0), needNormalize(false),
      ranbefore(false) {}
    void addEdge(int s, int e, cost_t cost, cap_t cap)
    {
        if (s == e) return;
        edge forward={e, cost, cap, cap, graph[e].size()};
        edge backward={s, -cost, 0, 0, graph[s].size()};
        if (cost < 0 || ranbefore) needNormalize = true;
        graph[s].emplace_back(forward);
        graph[e].emplace_back(backward);
    }
    bool normalize(int s) {
        auto infinite_cost = numeric_limits<cost_t>::max();
        vector<cost_t> dist(n, infinite_cost);
        dist[s] = 0;
        queue<int> q;
        vector<int> v(n), relax_count(n);
        v[s] = 1; q.push(s);
        while(!q.empty()) {
            int cur = q.front();
            v[cur] = 0; q.pop();
            if (++relax_count[cur] >= n) return false;
            for (const auto &e : graph[cur]) {
                if (iszerocap(e.residual_capacity)) continue;
                auto next = e.target;
                auto ncost = dist[cur] + e.cost;
                if (dist[next] > ncost) {
                    dist[next] = ncost;
                    if (v[next]) continue;
                    v[next] = 1; q.push(next);
                }
            }
        }
        for (int i = 0; i < n; i++) pi[i] = dist[i];
        return true;
    }

    pair<cost_t, cap_t> AugmentShortest(int s, int e, cap_t flow_limit) {
        auto infinite_cost = numeric_limits<cost_t>::max();
        auto infinite_flow = numeric_limits<cap_t>::max();
        typedef pair<cost_t, int> pq_t;
        priority_queue<pq_t, vector<pq_t>, greater<pq_t>> pq;
        vector<pair<cost_t, cap_t>> dist(n, make_pair(infinite_cost, 0));
        vector<int> from(n, -1), v(n);

        if (needNormalize || (ranbefore && lastStart != s))
            normalize(s);
        ranbefore = true;
        lastStart = s;

        dist[s] = pair<cost_t, cap_t>(0, infinite_flow);
        pq.emplace(dist[s].first, s);
        while(!pq.empty()) {
            auto cur = pq.top().second; pq.pop();
            if (v[cur]) continue;
            v[cur] = 1;
            if (cur == e) continue;
            for (const auto &e : graph[cur]) {
                auto next = e.target;
                if (v[next]) continue;
                if (iszerocap(e.residual_capacity)) continue;
                auto ncost = dist[cur].first + e.cost - pi[next] + pi[cur];
                auto nflow = min(dist[cur].second, e.residual_capacity);
                if (dist[next].first <= ncost) continue;
                dist[next] = make_pair(ncost, nflow);
                from[next] = e.revid;
                pq.emplace(dist[next].first, next);
            }
        }
        /** augment the shortest path **/
        auto p = e;
        auto pathcost = dist[p].first + pi[p] - pi[s];
```

```
        auto flow = dist[p].second;
        if (iszerocap(flow)|| (flow_limit <= 0 && pathcost >= 0)) return pair<
          cost_t, cap_t>(0, 0);
        if (flow_limit > 0) flow = min(flow, flow_limit);
        /* update potential */
        for (int i = 0; i < n; i++) {
            if (iszerocap(dist[i].second)) continue;
            pi[i] += dist[i].first;
        }
        while (from[p] != -1) {
            auto nedge = from[p];
            auto np = graph[p][nedge].target;
            auto fedge = graph[p][nedge].revid;
            graph[p][nedge].residual_capacity += flow;
            graph[np][fedge].residual_capacity -= flow;
            p = np;
        }
        return make_pair(pathcost * flow, flow);
    }

    pair<cost_t,cap_t> solve(int s, int e, cap_t flow_minimum = numeric_limits
      <cap_t>::max()) {
        cost_t total_cost = 0;
        cap_t total_flow = 0;
        for(;;) {
            auto res = AugmentShortest(s, e, flow_minimum - total_flow);
            if (res.second <= 0) break;
            total_cost += res.first;
            total_flow += res.second;
        }
        return make_pair(total_cost, total_flow);
    }
};
```

5 Geometry

5.1 Operations

```
#include <stdio.h>

typedef struct point{
    int x,y;
}point;

int ccw(const point &a, const point &b, const point &c){
    return a.x*b.y+b.x*c.y+c.x*a.y - a.y*b.x-b.y*c.x-c.y*a.x;
}
```

6 String

6.1 KMP

```
#include <stdio.h>
#include <string.h>

void kmp(char *t, char *p, int *r, int *ff){
    int l = strlen(t)
    for(int i=0,j=0;i<l;i++){
        if(t[i]!=p[j]){
            if(j==0)r[i]=0;
            else{
                i--;
                j=ff[j-1];
            }
        }
        else r[i]=++j;
    }
}

int main(){
    int n;
    char a[1000], b[1000]; // a: 찾을문자열 , b: 대상문자열.
    int ff[1000], d[1000]; // ff: 실패함수배열 , d: 결과배열 .입력
    //
    kmp(a+1,a,ff+1,ff); // 실패함수생성
    kmp(b,a,d,ff);

}
```

7 Miscellaneous

7.1 Magic Numbers

ì ì : 10 007 , 10 009 , 10 111 , 31 567 , 70 001 , 1 000 003 , 1 000 033 , 4 000 037 , 1 000 000 007 , 1 000 000 009