Code Template for ACM-ICPC

P_Not_Equal_NP

November 14, 2018

Contents

1 AhoCorasick 1
1.1 AhoCorasick 1

2 FFT 1
2.1 FFT 1

3 Geometry 2
3.1 Geometry 2

4 HungarianAlgorithm 3
4.1 HungarianAlgorithm 3

5 JavaFastIO 4
5.1 JavaFastIO 4

6 SuffixArray 4
6.1 SuffixArray 4

7 SuffixAutomaton 5
7.1 SuffixAutomaton 5

8 Treap 5
8.1 Treap 5

9 ZFunction 6
9.1 ZFunction 6

1 AhoCorasick

1.1 AhoCorasick

```
// Aho-Corasick
struct AhoCorasick
{
    struct Node
    {
        int cnt;
        vector<int> id;
        Node *nextNode, *nextPatternNode,
            *child[ALPHABET_SIZE];

        Node()
        {
            cnt = 0;
            id = vector<int>();
            nextNode = nextPatternNode = NULL;
            FOR(i, 0, ALPHABET_SIZE - 1)
                child[i] = NULL;
        }
    } root;

    void insertString(const string &s, int id)
    {
        Node *p = &root;
        FOR(i, 0, int(s.size()) - 1)
        {
            int z = encode(s[i]);
            if (p->child[z] == NULL)
                p->child[z] = new Node();
            p = p->child[z];
        }
        p->id.pb(id);
    }

    queue<Node*> q;

    void calculateNode()
    {
        q.push(&root);
        while(!q.empty())
        {
            Node *p = q.front();
            q.pop();
            FOR(i, 0, ALPHABET_SIZE - 1)
            if (p->child[i] != NULL)
            {
                Node *c = p->child[i];
                Node *f = p->nextNode;
                while(true)
                {
                    if (f == NULL)
                    {
                        c->nextNode = &root;
                        break;
                    }
                    if (f->child[i] != NULL)
                    {
                        c->nextNode = f->child[i];
                        break;
                    }
                    f = f->nextNode;
                }
                if (c->nextNode->id.empty())
                    c->nextPatternNode =
                        c->nextNode->nextPatternNode;
                else
                    c->nextPatternNode = c->nextNode;
                q.push(p->child[i]);
            }
        }
    }

    void query(const string &s)
    {
        Node *p = &root;
        FOR(i, 0, int(s.size()) - 1)
        {
            int z = encode(s[i]);
            while(p != NULL && p->child[z] == NULL)
                p = p->nextNode;
            if (p == NULL)
                p = &root;
            else
            {
                p = p->child[z];
                p->cnt ++;
            }
        }
    }

    stack<Node*> st;

    void pushAnswer(int *ans)
    {
        q.push(&root);
        while(!q.empty())
        {
            Node *p = q.front();
            q.pop();
            st.push(p);
            FOR(i, 0, ALPHABET_SIZE - 1)
            if (p->child[i] != NULL)
                q.push(p->child[i]);
        }
        while(!st.empty())
        {
            Node *p = st.top();
            st.pop();
            FOR(i, 0, int(p->id.size()) - 1)
                ans[p->id[i]] += p->cnt;
            if (p->nextNode != NULL)
                p->nextNode->cnt += p->cnt;
        }
    }
};
```

2 FFT

2.1 FFT

```
// FFT
const int NBIT = 18;
const int DEGREE = 1 << NBIT;
const double PI = acos(-1);
typedef complex<double> cplx;
cplx W[DEGREE];

int reverseBit(int mask)
```

```
{
    for(int i = 0, j = NBIT - 1; i < j; i ++, j --)
    if (((mask >> i) & 1) != ((mask >> j) & 1))
    {
        mask ^= 1 << i;
        mask ^= 1 << j;
    }
    return mask;
}

void fft(vector<cplx>& v, bool invert = false)
{
    v.resize(DEGREE);
    FOR(i, 0, DEGREE - 1)
    {
        int j = reverseBit(i);
        if (i < j)
            swap(v[i], v[j]);
    }
    vector<cplx> newV = vector<cplx>(DEGREE);
    for(int step = 1; step < DEGREE; step <<= 1)
    {
        double angle = PI / step;
        if (invert)
            angle = -angle;
        W[0] = cplx(1);
        cplx wn = cplx(cos(angle), sin(angle));
        FOR(i, 1, step - 1)
            W[i] = W[i - 1] * wn;

        int startEven = 0;
        int startOdd = step;
        while(startEven < DEGREE)
        {
            FOR(i, 0, step - 1)
            {
                newV[startEven + i] = v[startEven + i] +
                    W[i] * v[startOdd + i];
                newV[startOdd + i] = v[startEven + i] -
                    W[i] * v[startOdd + i];
            }
            startEven += (step << 1);
            startOdd = startEven + step;
        }

        FOR(i, 0, DEGREE - 1)
            v[i] = newV[i];
    }
    if (invert)
        FOR(i, 0, DEGREE - 1)
            v[i] /= DEGREE;
}
```

3 Geometry

3.1 Geometry

```
template<typename T> struct Point
{
    T x, y;

    Point<T>(T x = 0, T y = 0): x(x), y(y) {}

    bool operator == (const Point<T> &c)
    {
        return x == c.x && y == c.y;
    }

    bool operator < (const Point<T> &c)
    {
        return x < c.x || (x == c.x && y < c.y);
    }

    friend ostream& operator << (ostream &os, Point<T>
        point)
    {
        os << point.x << ' ' << point.y;
        return os;
    }

    double distance(const Point<T> &c)
    {
        return sqrt((x - c.x) * (x - c.x) + (y - c.y) *
            (y - c.y));
    }
};

template<typename T> struct Vector2D
{
    T x, y;

    Vector2D<T>(T x = 0, T y = 0): x(x), y(y) {}

    Vector2D<T>(Point<T> from, Point<T> to)
    {
        x = to.x - from.x;
        y = to.y - from.y;
    }

    bool operator == (const Vector2D<T> &c) {return x
        == c.x && y == c.y;}
    bool operator < (const Vector2D<T> &c) {return x <
        c.x || (x == c.x && y < c.y);}

    Vector2D<T> operator += (const Vector2D<T> &a){x +=
        a.x, y += a.y; return (*this);}
    Vector2D<T> operator -= (const Vector2D<T> &a) {x
        -= a.x, y -= a.y; return (*this);}
    Vector2D<T> operator *= (T a) {x *= a, y *= a;
        return (*this);}
    Vector2D<T> operator /= (T a) {x /= a, y /= a;
        return (*this);}

    Vector2D<T> operator + (const Vector2D<T>
        &a){return Vector2D<T>(*this) += a;}
    Vector2D<T> operator - (const Vector2D<T>
        &a){return Vector2D<T>(*this) -= a;}
    Vector2D<T> operator * (T a){return
        Vector2D<T>(*this) *= a;}
    Vector2D<T> operator / (T a){return
        Vector2D<T>(*this) /= a;}

    friend ostream& operator << (const ostream &os,
        Vector2D<T> vect)
    {
        os << vect.x << ' ' << vect.y;
        return os;
    }

    T dot(const Vector2D<T> &c) {return x * c.x + y *
        c.y;}
    T cross(const Vector2D<T> &c) {return x * c.y - y *
        c.x;}

    double length() {return sqrt(x * x + y * y);}
    double angle(const Vector2D<T> &a) {return cross(a)
        / (length() * a.length());}
};

template<typename T> bool cw(Point<T> a, Point<T> b,
     Point<T> c)
{
    return (Vector2D<T>(a, b).cross(Vector2D<T>(b, c)))
        < 0;
}

template<typename T> bool ccw(Point<T> a, Point<T> b,
     Point<T> c)
{
    return (Vector2D<T>(a, b).cross(Vector2D<T>(b, c)))
        > 0;
}

template<typename T> struct Polygon
{
    vector< Point<T> > P;

    Polygon<T>() {};
    Polygon<T>(const vector< Point<T> > &P): P(P) {};

    int vertexCount() {return P.size();}

    T area()
    {
        T result = 0;
        for(int i = 0; i < vertexCount() - 1; i ++)
        {
            T x1 = P[i].x, y1 = P[i].y;
            T x2 = P[i + 1].x, y2 = P[i + 1].y;
            result += x1 * y2 - x2 * y1;
        }
        result += P[P.size() - 1].x * P[0].y - P[0].x *
             P[P.size() - 1].y;
        return abs(result) / 2;
    }

    void makeConvexHull()
    {
        if (vertexCount() == 1)
            return;
        sort(P.begin(), P.end());
        vector< Point<T> > result;
        result.push_back(P[0]);
        FOR(i, 1, vertexCount() - 1)
        {
            while(result.size() >= 2 &&
                !cw(result[result.size() - 2],
                result[result.size() - 1], P[i]))
               result.pop_back();
            result.push_back(P[i]);
        }
        int lock = result.size();
        FORD(i, vertexCount() - 2, 0)
        {
            while(result.size() > lock &&
                !cw(result[result.size() - 2],
                result[result.size() - 1], P[i]))
               result.pop_back();
            result.push_back(P[i]);
        }
        result.pop_back();
        P.clear();
        P = result;
    }
};
```

4 HungarianAlgorithm

4.1 HungarianAlgorithm

```
// Hungarian Algorithm
int n, c[mn][mn], fx[mn], fy[mn];
int matchX[mn], matchY[mn], Queue[mn];
int reachX[mn], reachY[mn], inReachY[mn];
int trace[mn], numX, numY, co = 0, ans = 0;

void setup()
{
    cin >> n;
    FOR(x, 1, n)
    FOR(y, 1, n)
        c[x][y] = maxC;
    int u, v;
    while(cin >> u)
    {
        cin >> v;
        cin >> c[u][v];
    }
}

int findArgumentPath(int s)
{
    co ++;
    numX = numY = 0;
    int l = 1, r = 1;
    Queue[1] = s;
    while(l <= r)
    {
        int x = Queue[l ++];
        reachX[++ numX] = x;
        FOR(y, 1, n)
        if (inReachY[y] != co && C(x, y) == 0)
        {
            inReachY[y] = co;
            reachY[++ numY] = y;
            trace[y] = x;
            if (!matchY[y])
                return y;
            Queue[++ r] = matchY[y];
        }
    }
    return 0;
}

void changeEdge()
{
    int delta = maxC;
    FOR(i, 1, numX)
    {
        int x = reachX[i];
        FOR(y, 1, n)
        if (inReachY[y] != co)
            delta = min(delta, C(x, y));
    }
    FOR(i, 1, numX)
        fx[reachX[i]] += delta;
    FOR(i, 1, numY)
        fy[reachY[i]] -= delta;
}

void argumenting(int y)
{
    while(inReachY[y] == co)
    {
        int x = trace[y];
        int nex = matchX[x];
        matchX[x] = y;
        matchY[y] = x;
        y = nex;
    }
}

void xuly()
{
    FOR(x, 1, n)
    while(true)
    {
        int y = findArgumentPath(x);
        if (y)
        {
            argumenting(y);
            break;
        }
        changeEdge();
    }
    FOR(x, 1, n)
        ans += c[x][matchX[x]];
    cout << ans << '\n';
    FOR(x, 1, n)
        cout << x << ' ' << matchX[x] << '\n';
}
```

5 JavaFastIO

5.1 JavaFastIO

```
// Fast IO class in Java
static class FastReader
{
    final BufferedReader br;
    StringTokenizer st;

    FastReader()
    {
        br = new BufferedReader(new
                InputStreamReader(System.in));
    }

    String next()
    {
        while (st == null || !st.hasMoreElements())
        {
            try
            {
                st = new StringTokenizer(br.readLine());
            }
            catch (IOException e)
            {
                e.printStackTrace();
            }
        }
        return st.nextToken();
    }

    int nextInt()
    {
        return Integer.parseInt(next());
    }
}

static class FastWriter{
    PrintWriter printWriter;

    FastWriter(){
        printWriter = new PrintWriter(new
            BufferedOutputStream(System.out));
    }

    void print(Object object){
        printWriter.print(object);
    }

    void flush(){
        printWriter.flush();
    }
}
```

6 SuffixArray

6.1 SuffixArray

```
// Suffix Array and LCP Array
void calculateSuffixArray(string &s, int* sa, int*
    group, pair< pair<int, int> , int > * data)
{
    int n = s.size();
    FOR(i, 1, n)
        group[i] = s[i - 1];
    for(int length = 1; length <= n; length <<= 1)
    {
        FOR(i, 1, n)
            data[i] = mp(mp(group[i], (i + length > n?
                -1 : group[i + length])), i);
        sort(data + 1, data + n + 1);
        FOR(i, 1, n)
            group[data[i].S] = group[data[i - 1].S] +
                (data[i].F != data[i - 1].F);
    }
    FOR(i, 1, n)
        sa[i] = data[i].S;
}

void calculateLCPArray(string &s, int* lcp, int* sa,
    int* pos)
{
    int n = s.size();
    FOR(i, 1, n)
        pos[sa[i]] = i;
    int result = 0;
    FOR(i, 1, n)
    {
        if (pos[i] == n)
        {
```

```
        result = 0;
        continue;
    }
    int j = sa[pos[i] + 1];
    while(i + result <= n && j + result <= n && s[i
         + result - 1] == s[j + result - 1])
        result ++;
    lcp[pos[i]] = result;
    if (result)
        result --;
  }
}
```

7 SuffixAutomaton

7.1 SuffixAutomaton

```
// Suffix Automaton
class SuffixAutomaton
{
  private:
    class SAState
    {
      public:
        int length;
        SAState *link, *next[26];

        SAState(int length = 0, SAState *link =
            NULL): length(length), link(link)
        {
            FOR(i, 0, 25)
                next[i] = NULL;
        }
    };

    SAState *root, *last;

  public:
    SuffixAutomaton()
    {
        last = root = new SAState(0, NULL);
    }

    void insert(char c)
    {
        c -= 'a';
        SAState* newState = new SAState(last->length
             + 1);
        while (last != NULL && last->next[c] == NULL)
        {
            last->next[c] = newState;
            last = last->link;
        }
        if (last == NULL)
            newState->link = root;
        else
        {
            SAState* stateC = last->next[c];
            if (stateC->length == last->length + 1)
                newState->link = stateC;
            else
            {
                SAState* cloneState = new
                     SAState(last->length + 1,
                     stateC->link);
                FOR(i, 0, 25)
                    cloneState->next[i] =
                        stateC->next[i];
                while (last != NULL && last->next[c]
                     == stateC)
                {
                    last->next[c] = cloneState;
                    last = last->link;
                }
                newState->link = stateC->link =
                     cloneState;
            }
        }
        last = newState;
    }

    bool checkSubstring(string& s)
    {
        SAState* state = root;
        FOR(i, 0, int(s.size()) - 1)
        {
            if (state->next[s[i] - 'a'] == NULL)
                return false;
            state = state->next[s[i] - 'a'];
        }
        return true;
    }
};
```

8 Treap

8.1 Treap

```
// Implicit Treap
template <typename T> class Treap
{
  private:
    class TreapNode
    {
      public:
        T value;
        int priority, cnt;
        TreapNode *lc, *rc;

        TreapNode() {}

        TreapNode(T value): value(value)
        {
            priority = getRandom(1, maxC);
            cnt = 1;
            lc = rc = NULL;
        }
    };

    int getCount(TreapNode* node)
    {
        return (node? node->cnt : 0);
    }

    void updateCount(TreapNode* node)
    {
        if (node)
            node->cnt = getCount(node->lc) +
                 getCount(node->rc) + 1;
    }
```

```
        TreapNode* merge(TreapNode* l, TreapNode* r)
        {
            if (!l || !r)
                return (l? l : r);
            TreapNode* re = NULL;
            if (l->priority > r->priority)
            {
                l->rc = merge(l->rc, r);
                re = l;
            }
            else
            {
                r->lc = merge(l, r->lc);
                re = r;
            }
            updateCount(re);
            return re;
        }

        void split(TreapNode* node, TreapNode*& l,
             TreapNode*& r, int pos, int add = 0)
        {
            if (!node)
            {
                l = r = NULL;
                return;
            }
            int currentPos = add + getCount(node->lc);
            if (pos <= currentPos)
            {
                split(node->lc, l, node->lc, pos, add);
                r = node;
            }
            else
            {
                split(node->rc, node->rc, r, pos,
                    currentPos + 1);
                l = node;
            }
            updateCount(node);
        }

        TreapNode* get(TreapNode* node, int pos, int
             add = 0)
        {
            if (!node)
                return NULL;
            int currentPos = add + getCount(node->lc);
            if (pos == currentPos)
                return node;
            if (pos < currentPos)
                return get(node->lc, pos, add);
            return get(node->rc, pos, currentPos + 1);
        }

        void erase(TreapNode*& node, int pos, int add =
             0)
        {
            if (!node)
                return;
            int currentPos = add + getCount(node->lc);
            if (pos == currentPos)
            {
                delete node;
                node = merge(node->lc, node->rc);
            }
            else if (pos < currentPos)
                erase(node->lc, pos, add);
            else
                erase(node->rc, pos, currentPos + 1);
            updateCount(node);
        }

        void print(TreapNode* node)
        {
            if (!node)
                return;
            print(node->lc);
            cout << node->value << ' ';
            print(node->rc);
        }

        TreapNode* root;

    public:
        Treap()
        {
            root = NULL;
        }

        int size()
        {
            return getCount(root);
        }

        void insert(T value, int pos)
        {
            TreapNode *l = NULL, *r = NULL;
            split(root, l, r, pos);
            TreapNode* newItem = new TreapNode(value);
            root = merge(merge(l, newItem), r);
        }

        void insert(T value)
        {
            insert(value, size());
        }

        T get(int pos)
        {
            return get(root, pos)->value;
        }

        void erase(int pos)
        {
            erase(root, pos);
        }

        void print()
        {
            print(root);
            cout << '\n';
        }
};
```

9 ZFunction

9.1 ZFunction

```
// Z Function
void calculateZFunction(string &s, int *z)
```

```
{
    int n = s.size(), l = 1, r = 1;
    FOR(i, 2, n)
    {
        int k = i - l + 1;
        if (r < i || (r >= i && z[k] >= r - i + 1))
        {
            l = i, r = max(r, i - 1);
            while(r < n && s[r] == s[r - l + 1])
                r ++;
            z[i] = r - l + 1;
        }
        else
            z[i] = z[k];
    }
}
```