Team Contest Reference
getRandomNumber(){return 4;}

Universität zu Lübeck

22. November 2012

Inhaltsverzeichnis

Für Primzahlen gilt immer (aber nicht nur für Primzahlen)

$$a^p \equiv a \mod p \quad \text{bzw.} \quad a^{p-1} \equiv 1 \mod p.$$

```
static boolean[] sieve(int until) {
  boolean[] a = new boolean[until + 1];
  Arrays.fill(a, true);
  for (int i = 2; i < Math.sqrt(a.length); i++) {
    if (a[i]) {
      for (int j = i * i; j < a.length; j += i) a[j] = false;
    }
  }
  return a; // a[i] == true, iff. i is prime. a[0] is ignored
}
```

```
static boolean isPrim(int p) {
  if (p < 2 || p > 2 && p % 2 == 0) return false;
  for (int i = 3; i <= Math.sqrt(p); i += 2)
    if (p % i == 0) return false;
  return true;
}
```

```
static int[][] mem = new int[MAX_N][(MAX_N + 1) / 2];
static int binoCo(int n, int k) {
  if (k < 0 || k > n) return 0;
  if (2 * k > n) binoCo(n, n - k);
  if (mem[n][k] > 0) return mem[n][k];
  int ret = 1;
  for (int i = 1; i <= k; i++) {
    ret *= n - k + i;
    ret /= i;
    mem[n][i] = ret;
  }
  return ret;
}
```

Bedeutung der größten gemeinsamen Teiler:

$$d = \text{ggT}(a, b) = as + bt$$

Verwendung zu Berechnung des inversen Elements b zu a bezüglich einer Restklassengruppe n (a und n müssen teilerfremd sein):

$$ab \equiv 1 \mod n \quad \Leftrightarrow \quad s \equiv b \mod n \quad \text{für } 1 = \text{ggT}(a, n)$$

1.3.1 Erweiterter Euklidischer Algorithmus

```
static int[] eea(int a, int b) {
  int[] dst = new int[3];
  if (b == 0) {
    dst[0] = a;
    dst[1] = 1;
    return dst; // a, 1, 0
  }
  dst = eea(b, a % b);
  int tmp = dst[2];
  dst[2] = dst[1] - ((a / b) * dst[2]);
  dst[1] = tmp;
  return dst;
}
```

2 Datenstukturen

2.1 Fenwick Tree (Binary Indexed Tree)

```
class FenwickTree {
  private int[] values;
  private int n;
  public FenwickTree(int n) {
    this.n = n;
    values = new int[n];
  }
  public int get(int i) { //get value of i
    int x = values[0];
    while (i > 0) {
      x += values[i];
      i -= i & -i; }
    return x;
  }
  public void add(int i, int x) { // add x to interval [i,n]
    if (i == 0) values[0] += x;
    else {
      while (i < n) {
        values[i] += x;
        i += i & -i; }
    }
  }
}
```

3 Graphenalgorithmen

3.1 Topologische Sortierung

```
static List<Integer> topoSort(Map<Integer, List<Integer>> edges,
    Map<Integer, List<Integer>> revedges) {
  Queue<Integer> q = new LinkedList<Integer>();
  List<Integer> ret = new LinkedList<Integer>();
  Map<Integer, Integer> indeg = new HashMap<Integer, Integer>();
  for (int v : revedges.keySet()) {
    indeg.put(v, revedges.get(v).size());
    if (revedges.get(v).size() == 0)
      q.add(v);
  }
  while (!q.isEmpty()) {
    int tmp = q.poll();
    ret.add(tmp);
    for (int dest : edges.get(tmp)) {
      indeg.put(dest, indeg.get(dest) - 1);
      if (indeg.get(dest) == 0)
        q.add(dest);
    }
  }
  return ret;
}
```

3.2 Prim (Minimum Spanning Tree)

```
#define WHITE 0
#define BLACK 1
#define INF INT_MAX

int baum( int **matrix, int N){
  int i, sum = 0;

  int color[N];
  int dist[N];

    // markiere alle Knoten ausser 0 als unbesucht
  color[0] = BLACK;
  for( i=1; i<N; i++){
    color[i] = WHITE;
    dist[i] = INF;
  }

    // berechne den Rand
  for( i=1; i<N; i++){
        if( dist[i] > matrix[i][nextIndex]){
            dist[i] = matrix[i][nextIndex];
        }
    }

  while( 1){
    int nextDist = INF, nextIndex = -1;

    /* Den naechsten Knoten waehlen */
    for(i=0; i<N; i++){
      if( color[i] != WHITE) continue;

      if( dist[i] < nextDist){
        nextDist = dist[i];
        nextIndex = i;
      }
    }

    /* Abbruchbedingung*/
    if( nextIndex == -1) break;

    /* Knoten in MST aufnehmen */
    color[nextIndex] = RED;
    sum += nextDist;

    /* naechste kuerzeste Distanzen berechnen */
    for( i=0; i<N; i++){
            if( i == nextIndex || color[i] == BLACK ) continue;

            if( dist[i] > matrix[i][nextIndex]){
                dist[i] = matrix[i][nextIndex];
            }
    }
  }

  return sum;
}
```

3.3 Maximaler Fluss (Ford-Fulkerson)

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

#define n_MAX 36
#define m_MAX 30
#define SIZE (m+6+2)
#define SIZE_MAX 38
#define QUELLE (m+6)
#define SENKE (m+7)
#define NONE -1
#define INF INT_MAX/2

int n, m;
int capacity[SIZE_MAX][SIZE_MAX];
int flow[SIZE_MAX][SIZE_MAX];
int queue[SIZE_MAX], *head, *tail;
int state[SIZE_MAX];
```

```
18 int pred[SIZE_MAX];
19
20 enum { XS, S, M, L, XL, XXL };
21 enum { UNVISITED, WAITING, PROCESSED };
22
23 int strToOffset( char *str);
24 int maxFlow( int quelle, int senke);
25
26 int main(){
27
28     int numOfProps;
29     scanf("%d\n", &numOfProps);
30
31     while( numOfProps--){
32         scanf("%d␣%d\n", &n, &m);
33
34         int i, j;
35
36         /* Matrix initialisieren */
37         for( i=0; i< SIZE; i++){
38             for( j=0; j< SIZE; j++){
39                 capacity[i][j] = flow[i][j] = 0;
40
41                 if( i == QUELLE && j < m){
42                     capacity[i][j] = 1;
43                     continue;
44                 }
45
46                 if( j == SENKE && i >= m && i < QUELLE){
47                     capacity[i][j] = n/6;
48                     continue;
49                 }
50             }
51         }
52
53         char str[4];
54
55         /* Matrix einlesen */
56         for( i=0; i< m; i++){
57             scanf("%s", str);
58             capacity[i][m+strToOffset(str)] = 1;
59             scanf("%s", str);
60             capacity[i][m+strToOffset(str)] = 1;
61         }
62
63
64         int foo = maxFlow( QUELLE, SENKE);
65         printf("%s\n", foo >= m ? "YES" : "NO");
66
67     }
68
69     return 0;
70 }
71
72 int strToOffset( char *str){
73     /*snip*/
74 }
75
76 void enqueue( int x){
77     *tail++ = x;
78     state[x] = WAITING;
79 }
80
81 int dequeue(){
82     int x = *head++;
83     state[x] = PROCESSED;
84     return x;
85 }
86
87 int bfs( int start, int target){
88     int u, v;
89     for( u=0; u< SIZE; u++){
90         state[u] = UNVISITED;
91     }
92     head = tail = queue;
93     pred[start] = NONE;
```

```
94
95      enqueue(start);
96
97      while( head < tail){
98          u = dequeue();
99
100         for( v= 0; v< SIZE; v++){
101             if( state[v] == UNVISITED &&
102                 capacity[u][v] - flow[u][v] > 0){
103
104                 enqueue(v);
105                 pred[v] = u;
106             }
107         }
108     }
109
110     return state[target] == PROCESSED;
111 }
112
113 int maxFlow( int quelle, int senke){
114     int max_flow = 0;
115
116     int u;
117
118     while( bfs( quelle, senke)){
119         int increment = INF, temp;
120
121         for( u= senke; pred[u] != NONE; u = pred[u]){
122             temp = capacity[pred[u]][u] - flow[pred[u]][u];
123             if( temp < increment){
124                 increment = temp;
125             }
126         }
127
128         for( u= senke; pred[u] != NONE; u = pred[u]){
129             flow[pred[u]][u] += increment;
130             flow[u][pred[u]] -= increment;
131         }
132
133         max_flow += increment;
134     }
135
136     return max_flow;
137 }
```

4 Geometrische Algorithmen

4.1 Graham Scan (Convex Hull)

```
1  static List<P> graham(List<P> l) {
2    if (l.size() < 3)
3      return l;
4    P temp = l.get(0);
5    for (P p : l)
6      if (temp.y > p.y || temp.y == p.y && temp.x > p.x)
7        temp = p;
8    final P start = temp; // min y (then leftmost)
9
10   Collections.sort(l, new Comparator<P>() {
11     public int compare(P o1, P o2) {
12       if (new Double(Math.atan2(o1.y - start.y, o1.x - start.x)) // same angle
13           .compareTo(Math.atan2(o2.y - start.y, o2.x - start.x)) == 0)
14         return new Double(Math.sqrt((o1.x - start.x)
15             * (o1.x - start.x) + (o1.y - start.y)
16             * (o1.y - start.y))).compareTo((o2.x - start.x)
17             * (o2.x - start.x) + (o2.y - start.y)
18             * (o2.y - start.y)); // use distance
19       return new Double(Math.atan2(o1.y - start.y, o1.x - start.x))
20           .compareTo(Math.atan2(o2.y - start.y, o2.x - start.x));
21     }
22   });
23   Stack<P> s = new Stack<P>();
24   s.add(start);
25   s.add(l.get(1));
26   for (int i = 2; i < l.size(); i++) {
```

```
    while (s.size() >= 2
        && ccw(s.get(s.size() - 2), s.get(s.size() - 1), l.get(i)) <= 0)
      s.pop();
    s.push(l.get(i));
  }
  return s;
}

// turn is counter-clockwise if > 0; collinear if = 0; clockwise else
static double ccw(P p1, P p2, P p3) {
  return (p2.x - p1.x) * (p3.y - p1.y) - (p2.y - p1.y) * (p3.x - p1.x);
}

public static class P {
  double x, y;

  P(double x, double y) {
    this.x = x;
    this.y = y;
  }
  // polar coordinates (not used)
  // double r() { return Math.sqrt(x * x + y * y); }
  // double d() { return Math.atan2(y, x); }
}
```

4.2 Punkt in Polygon

```
/**
 * -1: A liegt links von BC (ausser unterer Endpunkt)
 * 0: A auf BC
 * +1: sonst
 */
public static int KreuzProdTest(double ax, double ay, double bx, double by,
    double cx, double cy) {
  if (ay == by && by == cy) {
    if ((bx <= ax && ax <= cx) || (cx <= ax && ax <= bx)) return 0;
    else return +1;
  }
  if (by > cy) {
    double tmpx = bx, tmpy = by;
    bx = cx;
    by = cy;
    cx = tmpx;
    cy = tmpy;
  }
  if (ay == by && ax == bx) return 0;
  if (ay <= by || ay > cy) return +1;
  double delta = (bx - ax) * (cy - ay) - (by - ay) * (cx - ax);
  if (delta > 0) return -1;
  else if (delta < 0) return +1;
  else return 0;
}

/**
 * Input: P[i] (x[i],y[i]); P[0]:=P[n]
 * -1: Q ausserhalb Polygon
 * 0: Q auf Polygon
 * +1: Q innerhalb des Polygons
 */
public static int PunktInPoly(double[] x, double[] y, double qx, double qy) {
  int t = -1;
  for (int i = 0; i < x.length - 1; i++)
    t = t * KreuzProdTest(qx, qy, x[i], y[i], x[i + 1], y[i + 1]);
  return t;
}
```

5 Verschiedenes

5.1 Potenzmenge

```
static <T> Iterator<List<T>> powerSet(final List<T> l) {
  return new Iterator<List<T>>() {
    int i; // careful: i becomes 2^l.size()
    public boolean hasNext() {
      return i < (1 << l.size());
```

```
    }
    public List<T> next() {
      Vector<T> temp = new Vector<T>();
      for (int j = 0; j < l.size(); j++)
        if (((i >>> j) & 1) == 1)
          temp.add(l.get(j));
      i++;
      return temp;
    }
    public void remove() {}
    };
  }
```

5.2 Longest Common Subsequence

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


int lcs( char *a, char *b){
    int len = strlen( a);
    int lenb =strlen(b);

    int *zeile = malloc( (len+1) * sizeof(int)), *temp,
        *neue = malloc( (len+1) * sizeof(int)), i, j;

    for(i=0; i<len+1; i++){
        zeile[i] = neue[i] = 0;
    }

    for(j=0; j<lenb; j++){
        for(i=0; i<len; i++){
            if( a[i] == b[j]){
                neue[i+1] = zeile[i] + 1;
            } else {
                neue[i+1] = neue[i] > zeile[i+1] ? neue[i] : zeile[i+1];
            }
        }
        temp = zeile;
        zeile = neue;
        neue = temp;
    }

    int res = zeile[len];
    free( zeile);
    free( neue);
    return res;
}
```

5.3 Longest Increasing Subsequence

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

int lis( int *list, int n){
    int *sorted = malloc( n*sizeof(int)), sorted_n;
    int i, *lower, *upper, *mid, *pos;

    if( n == 0) return 0;

    sorted[0] = list[0];
    sorted_n = 1;

    for( i=1; i<n; i++){
        /* binaere Suche */
        lower = list;
        upper = list + sorted_n;
        mid = list + sorted_n / 2;


        while( lower < upper-1){
            if( list[i] < *mid){
                upper = mid;
            } else {
                lower = mid;
```

```
25              }
26
27              mid = lower + (upper-lower) / 2;
28          }
29
30    if( mid == list + sorted_n -1 && *mid < list[i]){
31              *mid = list[i];
32              sorted_n++;
33          }
34
35          if( list[i] < *mid){
36              *mid = list[i];
37          }
38      }
39
40      free( sorted);
41
42      return sorted_n;
43  }
```