Team Contest Reference

Geen Syntax ,
Universiteit Leiden

28 juni 2013

Inhoudsopgave

1 Setup **2**
1.1 Bestand “name.cpp” 2
1.2 Bestand “Makefile” 2
1.3 Bestand “setup.sh” 2

2 STL algoritmen **3**
2.1 Binary Search 3

3 Binary Search **3**

4 Fenwick tree **4**

5 Grafen **4**
5.1 Depth First Search 4
5.2 Topologisch sorteren 4
5.3 2-SAT 4
5.4 Code 2-SAT, SCC en DFS 5
5.5 Biconnected components 5
5.6 Kortste pad 6
5.6.1 Dijkstra’s algoritme . . . 6
5.6.2 Bellman-Ford 7
5.6.3 Floyd-Warshall 8
5.6.4 Afwijkende applicaties . . 8
5.7 Flow netwerken 8
5.7.1 Residual capacity graaf . 8
5.7.2 Ford-Fulkerson 9
5.8 Min-cost flow 10
5.9 Min-cut 10
5.10 Minimal spanning tree 10
5.10.1 Kruskals algoritme 10

6 Computational Geometry **11**
6.1 Projecties 11
6.2 Rotaties 11
6.3 Hoekformules 12
6.4 Uit-product 12
6.4.1 Hoek tussen vectoren . . . 12
6.4.2 Bocht naar links of rechts? 12
6.4.3 Snijden twee lijnstukken? 13
6.5 Oppervlakte van een veelhoek . . 13
6.6 Zwaartepunt van een veelhoek . . 13
6.7 Convex hull 14
6.7.1 Graham scan 14

7 Strings **15**
7.1 String Matching 15
7.2 Suffix Array 16
7.3 Longest common subsequence . . 17
7.4 Longest increasing subsequence . 17
7.5 Levenšteinafstand 18

8 Getaltheorie **18**
8.1 Uitgebreide Euclidische algoritme 18
8.2 Priemtest 18
8.3 Partitiefunctie 19

9 Lineaire stelsels oplossen **19**
9.1 Determinant berekenen 20

10 Tips **20**
10.1 Mogelijke algoritmes, inspiratie . 20
10.2 Bugs 21

Deze TCR is geschreven door Raymond van Bommel <mailto:raymondvanbommel@gmail.com>, Josse van Dobben de Bruyn <mailto:josse.vandobbendebruyn@gmail.com> en Erik Massop <mailto:e.massop@hccnet.nl>. Wij vinden het goed als anderen deze TCR gebruiken, *mits* eventuele verbeteringen met ons gedeeld zijn. Ook dient tekst van gelijke strekking als deze alinea in elke versie aanwezig zijn.

1 Setup

1.1 Bestand “name.cpp”

```cpp
/* Opgave: NAME */
// 7+8+7=22 includes
#include <cstdlib>
#include <cstdio>
#include <cmath>
#include <cstring>
#include <cctype>
#include <climits>
#include <cassert>

#include <vector>
#include <deque>
#include <queue>
#include <stack>
#include <list>
#include <set>
#include <map>
#include <string>

#include <iostream>
#include <sstream>
#include <utility>
#include <functional>
#include <limits>
#include <numeric>
#include <algorithm>

using namespace std;

void doit () {

}

int main () {
    int t;
    cin >> t; //scanf ("%d ", &t);
    for (int i = 0; i < t; i++) {
        doit ();
    }
    return 0;
}
/* Opgave: NAME */
```

1.2 Bestand “Makefile”

```
CXXFLAGS=-Wall -Wextra -O2 -ggdb
test: name
    ./name < name.in > mine
    diff -sy mine name.out
    touch test
name: name.cpp
```

1.3 Bestand “setup.sh”

Maak eerst een **backup** van reeds bestaande bestanden! Better be safe than sorry.

```
#!/bin/bash
for d in {A..J}; do
  mkdir -p $d;
  sed -r "s|name|$d|g" < Makefile > "$d/Makefile"
  sed -r "s|NAME|$d|g" < name.cpp > "$d/$d.cpp"
done
```

2 STL algoritmen

2.1 Binary Search

Zoek binair in [`begin`, `end`), waarbij die reeks oplopend gesorteerd is. Al deze functies nemen nog een derde template argument `Cmp` en derde argument `Cmp cmp`. Deze geeft een alternatieve <-relatie.

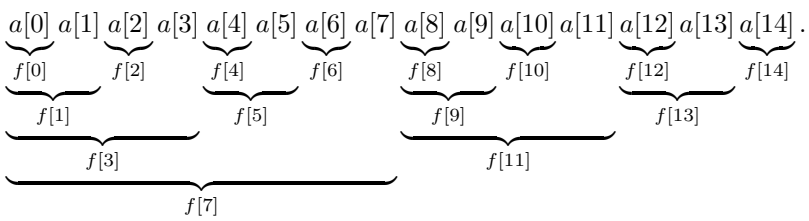
```
// True desda element bestaat dat niet groter danwel kleiner is.
bool binary_search (begin, end, x);
// Laatste zodat alle strict links strict kleiner.
it lower_bound (begin, end, x);
// Laatste zodat alle strict links kleiner gelijk.
it upper_bound (begin, end, x);
// Combinatie van lower_bound en upper_bound.
pair<it, it> equal_range (begin, end, x);
```

3 Binary Search

Deze implementatie zorgt dat `lo` steeds in het 'toegestane' gebied zit en `hi` in het 'verboden' gebied. In het begin stellen we ze op ongeldige waarden in. Dat maakt niet uit, want als hierdoor `mid` op een ongeldige waarde uitkomt, dan schelen `lo` en `hi` maar 1 en waren we al buiten de lus. We retourneren de hoogste index die mag.

```
lo = -1; // 1 voor begin
hi = N; // 1 na begin
while (hi-lo > 1) {
    mid = (lo+hi)/2;
    (mag (mid) ? lo : hi) = mid;
}
return lo;
```

4 Fenwick tree



Bedenkend dat `i |= i+1` de minst-significante 0 van i op 1 zet, en dat `k &= k-1` de minst-significante 1 van k op 0 zet, vinden we deze code:

```
void inc (unsigned i, int delta) {
    while (i < (unsigned)n) { f[i] += delta; i |= i+1; }
}
int exclsum (unsigned k) { // sum_{i=0}^{k-1} a[i]
    int ret = 0;
    while (k > 0) { ret += f[k-1]; k &= k-1; }
    return ret;
}
```

Voor gerelateerde toepassingen kan het makkelijker zijn om een binaire boom met $2n - 1$ knopen en n bladeren te gebruiken.

5 Grafen

5.1 Depth First Search

DFS heeft vele toepassingen. Bij sommige van deze dingen willen we knopen als actief danwel gedaan markeren. Dan kunnen we ook buren overslaan.

- Gerichte cykels: als een reeds actieve knoop wordt ontwikkeld.
- Ongerichte cykels: als een reeds actieve, of voltooide knoop wordt ontwikkeld.

5.2 Topologisch sorteren

Laat de globale variabele `n` het aantal punten van een graaf zijn. De volgende code registreert de finishvolgorde van de knopen in een globale variable `vector<int> finished`. Hiermee is `finished[n-1]` $< \cdots <$ `finished[0]` een topologische sortering van de knooppunten.

5.3 2-SAT

De code gaat ervanuit dat er `n` knopen zijn, waarbij knopen $2i$ en $2i + 1$ elkaars negatie zijn. De functie retourneert of er een oplossing is. Als er een oplossing is, dan is `val[comp[i]]` de valuatie van knoop i in een oplossing.

5.4 Code 2-SAT, SCC en DFS

```
bool mark[MAX_NODES];

template <typename S, typename F>
void dfs_visit (int cur, vector<int> buren[], S start, F finish) {
    if (mark[cur]) { return; }
    mark[cur] = true;

    start (cur);
    for (vector<int>::iterator it = buren[cur].begin(); it != buren[cur].end(); ++it) {
        dfs_visit (*it, buren, start, finish);
    }
    finish (cur);
}

void noop (int) { }
```

```
vector<int> finished;

void finish (int c) { finished.push_back (c); }

void finish_sort (vector<int> buren[]) {
    finished.clear(); fill_n (mark, n, false);
    for (int i = 0; i < n; i++) { dfs_visit (i, buren, noop, finish); }
}
```

```
int comp[MAX_NODES];
int comp_n;

void label (int cur) { comp[cur] = comp_n; }

void scc_and_top_sort () {
    finish_sort (heen); fill_n (mark, n, false); comp_n = 0;
    for (vector<int>::reverse_iterator it = finished.rbegin();
         it != finished.rend(); ++it) {
        if (mark[*it]) { continue; }
        dfs_visit (*it, terug, label, noop);
        comp_n++;
    }
}
```

```
int opp[MAX_NODES];

bool two_sat () {
    scc_and_top_sort ();
    for (int i = 0; i < n; i++) { opp[comp[i]] = comp[i^1]; }
    for (int i = 0; i < comp_n; i++) { if (opp[i] == i) { return false; } }
    fill_n (val, comp_n, false);
    for (int i = 0; i < comp_n; i++) { if (!val[i]) { val[opp[i]] = true; } }
    return true;
}
```

5.5 Biconnected components

De volgende code deelt de takken van de graaf op in componenten die verbonden blijven als er 1 knoop verwijderd wordt. In het commentaar staat hoe je de punten bepaald die de graaf splitsen als je ze verwijderd.

```
bool visited[MAX_NODES];
int low[MAX_NODES];
int count;
int d[MAX_NODES];
vector<pair<int, int> > st;


void mark(int a, int b) {
  pair<int, int> e;
  do {
    e = st.back();
    st.pop_back();
    // doe iets met de tak
  } while((e.first != a || e.second != b) && (e.first != b || e.second != a));
}

void dfs(int n, int parent) {
  visited[n] = true;
  low[n] = d[n] = ++count;
  for(unsigned i = 0; i <buren[n].size(); ++i) {
    int v = buren[n][i];
    if(!visited[v]) {
      st.push_back(make_pair(n,v));
      dfs(v, n);
      if(low[v] >= d[n]) {
        mark(n,v); // als n niet de root is dan n is een cut vertex
      }
      low[n] = min(low[n], low[v]);
    } else if(parent != v && d[v] < d[n]) {
      st.push_back(make_pair(n,v));
      low[n] = min(low[n], d[v]);
    }
  }
  // root == cut vertex <=> als er 2+ kinderen direct vanuit de root visited zijn.
}


void bicon() {
  count = 0;
  st.clear();
  for(unsigned i = 0; i < N; ++i)  { visited[i] = false;  }
  for(unsigned i = 0; i < N; ++i) {
    if(!visited[i])
      dfs(i, -1);
  }
}
```

5.6 Kortste pad

5.6.1 Dijkstra's algoritme

Dijkstra's algoritme werkt alleen voor grafen met niet-negatieve gewichten.

```
typedef pair<double, unsigned> halfpijl; // gewicht en bestemming

vector<halfpijl> buren[MAX_NODES];
double dist[MAX_NODES];

double dijkstra (unsigned s, unsigned t) {
```

```
7       priority_queue <halfpijl,
8                       vector <halfpijl>,
9                       greater <halfpijl> > q;
10      halfpijl cur;
11
12      // misschien is MAX_NODES een beetje overdreven
13      for (unsigned i = 0; i < MAX_NODES; i++) {
14          dist[i] = INFINITY;
15      }
16
17      dist[s] = 0;
18      q.push (make_pair (dist[s], s));
19
20      while (!q.empty ()) {
21          cur = q.top ();
22          q.pop ();
23
24          if (dist[cur.second] < cur.first) { continue; }
25
26          if (cur.second == t) {
27              return dist[t];
28          }
29
30          for (vector<halfpijl>::iterator it = buren[cur.second].begin ();
31               it != buren[cur.second].end ();
32               ++it) {
33              if (cur.first + it->first < dist[it->second]) {
34                  dist[it->second] = cur.first + it->first;
35                  q.push (make_pair (dist[it->second], it->second));
36              }
37          }
38      }
39      return INFINITY; // == dist[t]
40  }
```

In principe kan een knoop met ingraad i hier i-maal aan de priority-queue worden toegevoegd.

5.6.2 Bellman-Ford

Nota Bene: deze implementatie ondersteunt slechts paden van hoogstens `INT_MAX/4` !

```
1   struct pijl {
2       unsigned a,b;
3       int l; // signed!
4   };
5
6   unsigned n, m; // n nodes, m arches
7   int dist[MAX_NODES]; // signed!
8   pijl pijlen[MAX_ARCHES];
9
10  bool bellmanford (unsigned s) {
11      fill_n(dist, n, INT_MAX/2);
12      dist[s] = 0;
13
14      for (unsigned i = 0; i < n; i++) {
15          for (unsigned j = 0; j < m; j++) {
16              dist[pijlen[j].b] = min(dist[pijlen[j].b], dist[pijlen[j].a] + pijlen[j].l);
17          }
18      }
```

```
19
20      // return alleen false bij negatieve kringen die bereikt kunnen worden
21      for (unsigned j = 0; j < m; j++) {
22          if (dist[pijlen[j].a] < INT_MAX/4 &&
23               dist[pijlen[j].b] > dist[pijlen[j].a] + pijlen[j].l)
24              return false;
25      }
26
27      return true;
28  }
```

5.6.3 Floyd-Warshall

Floyd-Warshall vindt de lengtes van de paden van elke knoop naar elke andere knoop.

```
1  for k = 1 to n {
2      for i = 1 to n {
3          for j = 1 to n {
4              dist[i][j] = min (dist[i][j], dist[i][k] + dist[k][j]);
5          }
6      }
7  }
```

5.6.4 Afwijkende applicaties

Een aantal andere problemen is op te lossen met (aanpassingen van) kortste pad algoritmen:

- Vind het pad met de maximale capaciteit tussen S en T. De capaciteit is het minimum van de capaciteiten van de kanten in het pad.

  We passen het algoritme van Dijkstra aan. We kiezen nu telkens het punt met maximale capaciteit, in plaats van het punt met minimale afstand. Het is niet zinnig om hiervoor Bellman-ford te gebruiken.

- Een soortgelijke aanpassing van het kortste pad probleem kan gebruikt worden om een pad met een maximale waarschijnlijkheid/betrouwbaarheid/... te vinden. Oftewel een pad waarbij "waarde" het product is van de gewichten van de kanten. Vervang elk gewicht w door $-\log(w)$. Merk op dat voor waardes groter dan 1 deze afstanden negatief kunnen worden.

5.7 Flow netwerken

5.7.1 Residual capacity graaf

```
1  const cap_t cap_max = numeric_limits<cap_t>::max();
2
3  vector<int> out[MAX_NODES];     // de pijlen vanuit de knopen
4  int node_count = 0;
5
6  cap_t arr_cap[2*MAX_PIPES];   // de capaciteit op de pijlen
7  int   arr_to[2*MAX_PIPES];    // waar de pijlen heen gaan
8  int   arr_count = 0;
9
10 int add_node (void) { return node_count++; }
11
12 int add_arr (int f, cap_t c, int t) {
13     out[f].push_back (arr_count); arr_cap[arr_count] = c; arr_to[arr_count] = t;
14     return arr_count++;
15 }
```

```
16
17 // voeg pijp toe met met a->b-capaciteit cap en b->a-capaciteit revcap
18 void add_pipe (int a, int b, cap_t cap, cap_t revcap) {
19     add_arr (a, cap, b); add_arr (b, revcap, a);
20 }
21
22 void update_cap (int arr, cap_t delta) { // cap_t moet signed zijn!
23     // cap_max gebruiken we als infty, dus niet updaten
24     if (arr_cap[arr] < cap_max) { arr_cap[arr] += delta; }
25 }
26
27 void reset (void) { // leeg de graaf
28     for (int i = 0; i < node_count; i++) { out[i].clear (); }
29     node_count = arr_count = 0;
30 }
```

5.7.2 Ford-Fulkerson

```
1  cap_t ford_fulkerson (int s, int t) { // s != t
2      cap_t total = 0;
3      for (int i = 1; true ; i++) { // iteratienummer voor find
4          cap_t flow = find_bfs (s, t, i);
5          if (flow == 0) { break; }
6
7          // pad in omgekeerde richting doorlopen en capaciteiten aanpassen
8          for (int cur = t; cur != s; cur = arr_to[pred[cur]^1]) {
9              update_cap (pred[cur], -flow); update_cap (pred[cur]^1, flow);
10         }
11         total += flow;
12     }
13     return total;
14 }
```

Er zijn verschillende opties voor de functie `find`. Voor grafen met geheeltallige gewichten is de benodige tijd met breadth-first en depth-first find $\mathcal{O}(Ef^*)$ met f^* de gevonden maximal flow. De breadth-first variant heet ook wel Edmonds-Karp en draait in $\mathcal{O}(VE^2)$-tijd. De breadth-first variant is hieronder afgedrukt:

```
1  void schedule (int p, int to, cap_t c, queue<int> & q, int i)
2  { pred[to] = p; mark[to] = i; avail[to] = c; q.push (to); }
3
4  cap_t find_bfs (int s, int t, int i) { // i is iteratienummer
5      if (i == 1) { fill_n (mark, node_count, 0); } // reset int(!) mark[] in iteratie 1
6
7      if (s == t) { return cap_max; }
8
9      queue<int> q; // queue voor Breadth-First-Search
10     schedule (-1, s, cap_max, q, i);
11     while (!q.empty ()) {
12         int cur = q.front (); q.pop ();
13         for (vector<int>::iterator it = out[cur].begin (); it != out[cur].end (); ++it) {
14             if (mark[arr_to[*it]] == i || arr_cap[*it] == 0) { continue; }
15
16             schedule (*it, arr_to[*it], min (avail[cur], arr_cap[*it]), q, i);
17             if (arr_to[*it] == t) { return avail[t]; }
18         }
19     }
20     return 0;
21 }
```

5.8 Min-cost flow

Met een aangepaste functie voor het vinden van verbeterende paden kunnen we ook de min-cost max-flow vinden. De volgende code draait in $O(m \log n)$:

```
int potential[MAX_NODES];
int dist[MAX_NODES];
int pred[MAX_NODES];
typedef pair<int, int> halfpijl;

int find_dijkstra(int s, int t, int i, int& cost) {
    if(i == 1) fill_n(potential, node_count, 0);

    fill_n(dist, node_count, INT_MAX/2);
    dist[s] = 0;
    priority_queue<halfpijl, vector<halfpijl>, greater<halfpijl> > q;
    q.push(make_pair(0, s));
    while(!q.empty()) {
        halfpijl p = q.top(); q.pop();
        if(p.first > dist[p.second]) continue;
        for(vector<int>::iterator it = out[p.second].begin(); it != out[p.second].end();
            ++it) {

            int c = arr_cost[*it] + potential[p.second] - potential[arr_to[*it]];
            if(arr_cap[*it] && dist[arr_to[*it]] > p.first + c) {
                pred[arr_to[*it]] = *it;
                dist[arr_to[*it]] = p.first + c;
                q.push(make_pair(p.first + c,  arr_to[*it]));
            }
        }
    }
    if(dist[t] == INT_MAX/2) return 0;
    for(int i = 0; i < node_count; ++i) potential[i] += dist[i];

    int flow = INT_MAX;
    for(int i = t; i != s; i = arr_to[pred[i]^1]) flow = min(flow, arr_cap[pred[i]]);
    cost = potential[t] * flow;
    return flow;
}
```

Als je vanaf het begin al takken met negatieve kosten en strikt positieve capaciteit hebt moet je `potential` initialiseren met Bellman-Ford. Dit algoritme ondersteunt geen negatieve cykels.

5.9 Min-cut

De min-cut wordt geïnduceerd door de verzameling punten die bereikbaar zijn vanuit de bron. Bedenk dat dit in principe al in de mark array staat na `ford_fulkerson`.

5.10 Minimal spanning tree

5.10.1 Kruskals algoritme

```
int N, M;
struct edge { int u, v; int weight; } edges[MAX_EDGES];

bool cmp (const edge & e, const edge & f) { return e.weight < f.weight; }

void kruskal (void) {
    sort (edges, edges + M, cmp); // sorteer kanten
```

```
    for (int i = 0; i < N; i++) { init (i); } // elke knoop een eigen component

    for (int i = 0; i < M; i++) { // itereer in oplopende volgorde
        int u = representant (edges[i].u), // verkrijg representanten van de
            v = representant (edges[i].v); // componenten van edges[j].{u,v}

        if (u != v) { // als verschillende componenten: merge
            add (edges[i]); // voeg toe aan minimum spanning tree in wording
            merge (u, v);
        }
    }
}
```

Voor een efficiënte implementatie van `init`, `merge` en `representant` is een of andere Disjoint-Set datastructuur handig. Bijvoorbeeld een Disjoint-Set Forest (met amortized $\mathcal{O}(\alpha(n))$-tijd per operatie, met α de inverse van de Ackermann functie):

```
int parent[MAX_VERTICES], rank[MAX_VERTICES];

void init (int v) { parent[v] = v; rank[v] = 0; }

int representant (int v) { // met path reduction
    return (parent[v] == v) ? v : (parent[v] = representant (parent[v]));
}

void merge (int u, int v) { // met union by rank
    assert (parent[u] == u && parent[v] == v);
    if (rank[u] < rank[v]) { parent[u] = v; }
    else if (rank[u] > rank[v]) { parent[v] = u; }
    else { parent[u] = v; rank[v]++; }
}
```

6 Computational Geometry

```
struct vect { coord_t x, y; vect (coord_t _x, coord_t _y) : x(_x), y(_y) { } };
struct punt {
    coord_t x, y; naam_t naam;
    punt (coord_t _x, coord_t _y, coord_t _n) : x(_x), y(_y), naam(_n) { }
};

vect operator- (punt p, punt q) { return vect (q.x-p.x, q.y-p.y); }

ostream &operator<< (ostream &o, punt p) { o << p.naam; return o; }
ostream &operator<< (ostream &o, vect v) { o << v.x << ',' << v.y; return o; }
```

6.1 Projecties

```
coord_t dot (vect u, vect v) { return u.x*v.x + u.y*v.y; }
```

De projectie $p(x, y)$ van $x \in \mathbb{R}^n$ op $y \in \mathbb{R}^n$ wordt gegeven door:

$$p(x,y) = \frac{x \cdot y}{|y|}\hat{y} = \frac{x \cdot y}{|y|^2}y = \frac{x \cdot y}{y \cdot y}y, \qquad \text{waarbij } \hat{y} = \frac{y}{|y|}.$$

Dit is een scalair veelvoud van $\hat{y}$. Let op dat $x \cdot y$ een inproduct is en geen scalair product!

6.2 Rotaties

Om een vector $(x, y) \in \mathbb{R}^2$ precies θ graden in positieve richting te draaien:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}.$$

6.3 Hoekformules

Er geldt voor het inproduct dat

$$\langle p, q \rangle = |p||q| \cos(\theta)$$

met θ de hoek tussen de vectoren.Voor het uitproduct geldt dat

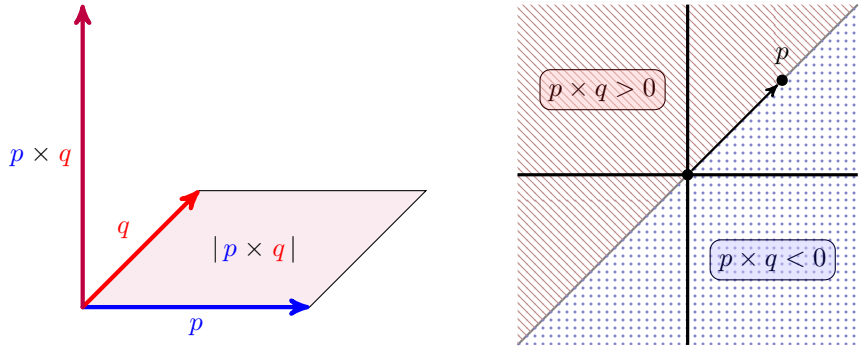
$$p \times q = |p||q| \sin(\theta) n$$

Met n de genormaliseerde normaal.

6.4 Uit-product

```
1 coord_t cross (vect u, vect v) { return u.x*v.y - u.y*v.x; }
```

Het uitproduct $p \times q = \det \left(\begin{smallmatrix} x_1 & x_2 \\ y_1 & y_2 \end{smallmatrix} \right) = x_1 y_2 - x_2 y_1 = -q \times p$ is gelijk aan de oppervlakte van het parallellogram geïnduceerd door p_1 en p_2 (zie figuur (a)).



(a) (b)

In de $\mathbb{R}^3$ is het uitproduct:

$$\begin{pmatrix} y_1 z_2 - y_2 z_1 \\ z_1 x_2 - z_2 x_1 \\ x_1 y_2 - x_2 y_1 \end{pmatrix}$$

6.4.1 Hoek tussen vectoren

Het teken van $p \times q$ vertelt ons iets over de hoek tussen de vectoren p en q (zie figuur (b)).

- Als q in positieve richting (tegen de klok in) van p ligt, geldt $p \times q > 0$.
- Als q in negatieve richting (met de klok mee) ligt, geldt $p \times q < 0$.
- Als p en q dezelfde hoek hebben (modulo 180°), geldt $p \times q = 0$. In dit geval is p een reëel veelvoud van q of andersom.

6.4.2 Bocht naar links of rechts?

```
1 // tegen overflows bij vermenigvuldigen van uitvoer van direction
2 int sgn (coord_t i) { return (i > 0) ? 1 : ((i < 0) ? -1 : 0); }
3
4 // Geef de orientatie van het lijnstuk van a naar c
5 // ten opzichte van het lijnstuk van a naar b.
6 int direction (punt a, punt b, punt c) { return sgn (cross (b-a, c-a)); }
```

Bepalen of de aaneengesloten vectoren $\overrightarrow{p_1p_2}$ en $\overrightarrow{p_2p_3}$ een bocht naar links of rechts maken in p_2:

- Als $(p_2 - p_1) \times (p_3 - p_1) > 0$ geldt, dan is de bocht naar links.
- Als $(p_2 - p_1) \times (p_3 - p_1) < 0$ geldt, dan is de bocht naar rechts.
- Als $(p_2 - p_1) \times (p_3 - p_1) = 0$ geldt, dan is er geen bocht.

6.4.3 Snijden twee lijnstukken?

Er zijn een paar vervelende randgevallen. Onderstaande code (functie `segmentsIntersect`) werkt. Deze functie onderzoekt of de lijnstukken $\overrightarrow{p_1p_2}$ en $\overrightarrow{p_3p_4}$ elkaar snijden. De code werkt ook als de coördinaten in `float` of `double` zijn.

```
// Check of c op het lijnstuk van a naar b ligt,
// gegeven het feit dat c op de lijn door a en b ligt.
bool onSegment(punt a, punt b, punt c) {
    return (min(a.x, b.x) <= c.x && max(a.x, b.x) >= c.x
         && min(a.y, b.y) <= c.y && max(a.y, b.y) >= c.y);
}

bool segmentsIntersect (punt p1, punt p2, punt p3, punt p4) {
    int d1 = direction(p3, p4, p1);
    int d2 = direction(p3, p4, p2);
    int d3 = direction(p1, p2, p3);
    int d4 = direction(p1, p2, p4);

    if (d1 * d2 < 0 && d3 * d4 < 0) return true;
    if (d1 == 0 && onSegment(p3, p4, p1)) return true;
    if (d2 == 0 && onSegment(p3, p4, p2)) return true;
    if (d3 == 0 && onSegment(p1, p2, p3)) return true;
    if (d4 == 0 && onSegment(p1, p2, p4)) return true;
    return false;
}
```

6.5 Oppervlakte van een veelhoek

$$\text{opp} = \frac{1}{2}\left| \sum_{(x,y)\to(x',y')\in A} xy' - x'y \right|$$

6.6 Zwaartepunt van een veelhoek

Het zwaartepunt kan bepaald worden met

$$C_x = \frac{1}{6A} \sum_{(x,y)\to(x',y')} (x + x')(xy' - x'y)$$

En

$$C_x = \frac{1}{6A} \sum_{(x,y)\to(x',y')} (y + y')(xy' - x'y)$$

Met A de oppervlakte van de veelhoek. Deze methode werkt niet als de veelhoek zichzelf doorsnijdt.

6.7 Convex hull

6.7.1 Graham scan

Deze code produceert uit `vector<punt> punten` een lijst van halfopen zijden van de convex hull, waarbij de hoekpunten steeds als laatste punt in de zijde voorkomen. Zo geeft punten $= \{(x, y) : 1 \leq x \leq y \leq 3\}$ een cyclische permutatie van

$$[[(1,1),(0,0)],[(1,0),(2,0)],[(2,1),(2,2)]] \quad \text{of} \quad [[(0,1),(0,0)],[(1,1),(2,2)],[(2,1),(2,0)]],$$

afhankelijk van de richting waarin de convex hull afgelopen wordt. Algoritme werkt niet betrouwbaar als punten meerdere keren voorkomen. Algoritme werkt niet als het aantal punten 1 of minder is. Als alle punten op een lijn liggen, dan krijgen we eindpunten eenmaal en interne punten tweemaal.

```
bool lexi_cmp (punt p, punt q) { return (p.y != q.y) ? (p.y < q.y) : (p.x < q.x); }

struct graham_cmp {
    punt o; graham_cmp (punt _o) : o(_o) { }
    bool operator() (punt p, punt q) const {
        int d = direction (o, p, q);
        return (d != 0) ? (d > 0) : (dot(p-o, p-o) < dot(q-o, q-o));
    }
};

void graham_scan (vector<punt> &punten, list< list<punt> > &hull) {
    // neem extreem punt
    punt o = *min_element (punten.begin(), punten.end(), lexi_cmp);
    // sorteren op hoek, dan afstand tot "o"
    sort (punten.begin(), punten.end(), graham_cmp (o));
    // "o" komt op eerste positie
    assert (punten.front().x == o.x && punten.front().y == o.y);

    // richting van bochten in de convex hull die we gaan bouwen.
    // (Is altijd hetzelfde (behalve als alle punten op een lijn).)
    // Hangt af van volgorde in "cross" en basispunt in "direction".
    int d = direction (o, *(punten.begin()+1), punten.back());

    hull.clear();

    // laatste halfopen zijde van convex hull (NB: loop doet back en front niet)
    hull.push_back(list<punt>());
    for (vector<punt>::reverse_iterator it = punten.rbegin()+1;
         it != punten.rend()-1 && 0 == direction (punten.back(), *it, o); ++it) {
        hull.back().push_back(*it);
    }
    hull.back().push_back (o);

    // eerste punt van eerste halfopen zijde van convex hull
    hull.push_back (list<punt>());
    hull.back().push_back (*(punten.begin()+1));

    for (vector<punt>::iterator it = punten.begin()+2; it != punten.end(); ++it) {
        // Zolang richting echt fout, gooi vorig weg.
        while (-1 == d * direction ((++hull.rbegin())->back(),
                                    hull.back().back(), *it)) {
            hull.pop_back();
        }
        // echt bocht om? Begin dan een nieuwe zijde.
        if (0 != direction ((++hull.rbegin())->back(), hull.back().back(), *it)) {
```

```
46              hull.push_back (list<punt>());
47          }
48          hull.back().push_back(*it);
49      }
50  }
```

7 Strings

7.1 String Matching

Het algoritme van Knuth-Morris-Pratt gaat uit van de prefix functie π, die wordt gegeven door

$$\pi^*[q] = \{k < q \text{ en } p_0 \dots p_k \text{ is een suffix voor } p_0 \dots p_q\};$$
$$\pi[q] = \max \pi^*[q].$$

Een belangrijke eigenschap van deze prefix functie is dat voor $q = 1, \dots, m$ met $\pi[q] \geq 0$ geldt dat

$$\pi[q] - 1 \in \pi^*[q-1].$$

De complexiteit hiervan is $O(n+k)$ met n en k de lengtes van de strings.

```
1  int *prefix;
2  char *str;     // of: string str
3  char *pattern; // of: string pattern
4
5  void compute_prefix_function(void) {
6      int m = strlen(pattern);
7
8      prefix[0] = -1;
9      for (int q = 1; q < m; q++) {
10         int k = prefix[q-1];
11         while (k >= 0 && pattern[k + 1] != pattern[q]) {
12             k = prefix[k];
13         }
14         if (pattern[k + 1] == pattern[q]) { /* match gevonden! */
15             prefix[q] = k + 1;
16         } else {
17             prefix[q] = -1;
18         }
19     }
20 }
21
22 void KMP_matcher() {
23     int n = strlen(str);
24     int m = strlen(pattern);
25     int q;
26
27     compute_prefix_function();
28
29     q = -1;
30     for (int i = 0; i < n; i++) {
31         while (q >= 0 && pattern[q + 1] != str[i]) {
32             q = prefix[q];
33         }
34         if (pattern[q + 1] == str[i]) {
35             q++;
36         }
```

```
        if (q + 1 == m) {
            // We have a match at i + 1 - m
            // for example: cout << (i + 1 - m) << endl;
            // if appropriate: return;
            cout << (i + 1 - m) << endl;
            q = prefix[q];
        }
    }
    // if appropriate: cout << "No matches found." << endl;
    // however this only works if we return when we do find a match
    cout << endl;
}
```

7.2 Suffix Array

De volgende code kan de suffix array van een string bepalen en de least commmon parent oftwel de lengte van de gemeenschappelijke prefix van twee suffixes bepalen.

```
struct Entry {
  int nr[2];
  int p;
  bool operator<(const Entry& o) const {
    if(nr[0] != o.nr[0]) return nr[0] < o.nr[0];
    if(nr[1] != o.nr[1]) return nr[1] < o.nr[1];
    return p < o.p;
  }
};

vector<vector<int> > calculateSA(const string& s) {
  vector<vector<int> > P(ceil(log2(s.size()))+2, vector<int>(s.size()));
  vector<Entry> ve(s.size());
  for(unsigned i = 0; i < s.size(); ++i)
    P[0][i] = s[i] - 'a';
  for(int stp=1, cnt=1; cnt >> 1 < (int)s.size(); ++stp, cnt <<=1) {
    for(unsigned i = 0; i < s.size(); ++i) {
      ve[i].nr[0] = P[stp-1][i];
      ve[i].nr[1] = i + cnt < s.size() ? P[stp-1][i+cnt] : -1;
      ve[i].p = i;
    }
    sort(ve.begin(), ve.end());
    for(unsigned i = 0; i < s.size(); ++i)
      P[stp][ve[i].p] = (i > 0 && ve[i].nr[0] == ve[i-1].nr[0] &&
                                  ve[i].nr[1] == ve[i-1].nr[1]) ? P[stp][ve[i-1].p] : i;
  }
  return P;
}

vector<unsigned> calculateSuffixArray(const string& s) {
    vector<vector<int> > order = calculateSA(s);
    vector<pair<int, unsigned> > v(s.size());
    for(unsigned i = 0; i < s.size(); ++i) v[i] = make_pair(order.back()[i], i);
    sort(v.begin(), v.end());
    vector<unsigned> sa(s.size());
    for(unsigned i = 0; i < s.size(); ++i) sa[i] = v[i].second;
    return sa;
}


```

```
int lcp(const vector<vector<int> >& P, unsigned x, unsigned y) {
  if(x == y) return P.front().size() - x;
  int ret = 0;
  for(int k = P.size()-1; k >= 0 && x < P.front().size() &&
                                    y < P.front().size(); --k)
    if(P[k][x] == P[k][y])
      x += 1 << k, y += 1 << k, ret += 1 << k;
  return ret;
}
```

7.3 Longest common subsequence

De uitdrukking $c[i,j]$ hieronder is de lengte van de longest common subsequence van strings $x_1 \cdots x_i$ en $y_1 \cdots y_j$:

$$c[i,j] = \begin{cases} 0 & \text{als } i = 0 \text{ of } j = 0 \\ c[i-1][j-1] + 1 & \text{als } i > 0 \text{ en } j > 0 \text{ en } x_i = y_j \\ \max(c[i,j-1], c[i-1,j]) & \text{als } i > 0 \text{ en } j > 0 \text{ en } x_i \neq y_j \end{cases} .$$

7.4 Longest increasing subsequence

```
vector <int> lijst;

bool cmp(int a, int b) {
    assert(a >= 0 && b >= 0);
    return lijst[a] < lijst[b];
}

list<int> longest_increasing_subsequence (void) {
    int n = lijst.size ();
    int L = 0;
    // P[i] is the predecessor i in a longest increasing subsequence of
    // lijst[0..i] containing element i.
    int P[n];
    // M[k] is de index van het eind van de sequence met lengte k + 1
    // met laagste eindwaarde.
    int M[n];
    list<int> seq;

    if (n == 0) { return seq; }

    M[0] = 0;
    P[0] = -1;
    L = 1;

    for (int i = 1; i < n; i++) {
        int j; // de lengte van de langste sequence waar lijst[i] achter kan
        if (cmp(M[0], i)) {
            j = lower_bound (M, M + L, i, cmp) - M;
        } else {
            j = 0;
        }

        P[i] = (j > 0) ? M[j - 1] : -1;

        if (j == L) {
            M[L++] = i;
```

```
        } else if (cmp(i, M[j])) {
            M[j] = i;
        }
    }

    for (int a = M[L - 1]; a >= 0; a = P[a]) {
        seq.push_front (lijst[a]);
    }

    return seq;
}
```

7.5 Levenšteinafstand

Het minimal aantal operaties nodig om een string in een andere te transformeren, met toegestane operaties invoeging, verwijdering en subsitutie van karakters. Hieronder is de expressie $d[i,j]$ de Levenšteinafstand tussen $x_1 \cdots x_i$ en $y_1 \cdots y_j$:

$$d[i,j] = \begin{cases} i+j & \text{als } i=0 \text{ of } j=0 \\ \min \begin{pmatrix} d[i-1,j]+1, \\ d[i,j-1]+1, \\ d[i-1,j-1]+1_{\{x_i \neq y_j\}} \end{pmatrix} & \text{als } i>0 \text{ en } j>0 \end{cases}$$

8 Getaltheorie

8.1 Uitgebreide Euclidische algoritme

Met dit algoritme kan zowel de ggd van twee getallen a en b bepaald worden als een paar (x,y) waarvoor geldt $ax+by = \text{ggd}(a,b)$.

```
pair <int, pair <int, int> > uggd(int a, int b) {
    int x, lastx, y, lasty;
    int q;

    x = lasty = 0;
    y = lastx = 1;

    while (b != 0) {
        q = a / b;

        a %= b;
        swap(a,b);

        lastx -= q*x;
        swap(x, lastx);

        lasty -= q*y;
        swap(y, lasty);
    }

    return make_pair (a, make_pair (lastx, lasty));
}
```

```
unsigned ggd(unsigned a, unsigned b) {
    while(b) {
        a %= b;
        swap(a,b);
    }
    return a;
}
```

8.2 Priemtest

De volgende code implementeert een deterministische versie van Miller-Rabin:

```
__int128_t power(__int128_t a, unsigned long long e, unsigned long long n) {
```

```
    if(e == 1) return a;
    if(e & 1) return (a * power(a, e-1, n) % n);
    __int128_t v = power(a, e/2, n);
    return (v*v) % n;
}

bool isprime(unsigned long long n) {
    if(n <= 1) return false;
    unsigned long long d = n-1;
    unsigned s = 0;
    while(d % 2 == 0) { d /= 2; ++s; }
    // BELANGRIJK: typ deze getallen goed over!
    const unsigned a_arr[] = {2, 325, 9375, 28178, 450775, 9780504, 1795265022};
    for(unsigned i = 0; i < sizeof(a_arr)/sizeof(unsigned) && a_arr[i] < n; ++i) {
        unsigned a = a_arr[i];
        __int128_t v = power(a, d, n);
        if(v == 1) continue;
        bool composite = true;
        for(unsigned p = 0; p < s; ++p) {
            if(v == n-1) { composite = false; break; }
            v = (v * v) % n;
        }
        if(composite) return false;
    }
    return true;
}
```

8.3 Partitiefunctie

De partitiefunctie geeft het aantal manieren dat een getal n opgedeeld kan worden in positieve getallen als volgorde niet uitmaakt, zodanig dat de som weer n is.

```
long long parts[P][P]; // [m][n] = # opdelingen van n die overal <= m zijn

long long partition(int upper) {
    parts[0][0] = 1;
    fill_n (parts[0] + 1, upper, 0);
    for (int n = 1; n <= upper; n++) {
        parts[n][0] = 1;
        for (int sum = 1; sum <= upper; sum++) {
            parts[n][sum] = parts[n - 1][sum];
            if (sum >= n)
                parts[n][sum] += parts[n][sum - n];
        }
    }
    return parts[upper][upper];
}
```

Overflow

type	iteraties veilig
s32	121
u32	127
s64	405
u64	416
s128	1437
u128	1458

9 Lineaire stelsels oplossen

We kunnen een stelsel lineaire vergelijkingen oplossen met het vegen van een matrix. De volgende code doet dat als de matrix inverteerbaar is en geeft `false` als de matrix niet inverteerbaar is.

```


bool linsolve(vector<vector<double> > A, vector<double> b, vector<double>& x,
            unsigned rows, unsigned cols) {
```

```
5       unsigned done = 0;
6       for(unsigned i = 0; i < cols; ++i) {
7           int r = -1;
8           for(unsigned j = done; j < rows; ++j)
9               if(fabs(A[j][i]) > 1e-12) { r = j; break; }
10          if(r == -1) continue;
11          swap(A[done], A[r]);
12          swap(b[done], b[r]);
13          double dd = 1.0 / A[done][i];
14          for(unsigned j = 0; j < cols; ++j)
15              A[done][j] *= dd;
16          b[done] *= dd;
17          for(unsigned j = 0; j < rows; ++j) {
18              if(done == j) continue;
19              double d = A[j][i] / A[done][i];
20              for(unsigned k = 0; k < cols; ++k)
21                  A[j][k] -= d * A[done][k];
22              b[j] -= d* b[done];
23          }
24          done++;
25      }
26      if(done == cols && rows == cols) {
27          for(unsigned i = 0; i < cols; ++i)
28              x[i] = b[i] / A[i][i];
29          return true;
30      } else
31          return false;
32  }
```

9.1 Determinant berekenen

We kunnen de determinant van een matrix bepalen door via Gauss-eliminatie een matrix te maken waarvan alles onder de diagonaal 0 is. Vervolgens nemen we het product van de diagonaal en dat is dan de determinant. Let op dat hoewel het optellen van rijen de determinant niet veranderd, het schalen dat wel doet en dat je dus in het bijzonder niet alle pivots naar 1 moet normaliseren.

10 Tips

10.1 Mogelijke algoritmes, inspiratie

- Probeer te denken vanuit de grenzen.
- Is het te doorzoeken met min/max binair/ternair zoeken? (monotoon?)
- Brute Force (met pruning)
- Brute Force small instances to discover patterns or test algorithm
- Probeer Greedy
- Dynamic Programming
- (Mincost)Maxflow
- Kan je de graaf bipartiet krijgen?
- Precalculeren
- Inclusion/Exclusion

- Neigt het naar NIM?
- Line sweep / radial sweep
- Coordinaten comprimeren
- Vervang strings met iets snellers.

10.2 Bugs

- Initialiseer alle variabelen voor elk testcase!!!
- Kijk uit voor variabelen met dezelfde naam.
- Geef je altijd de juiste uitvoer? (e.g.: geen pad, print ¨impossible¨)
- Ergens een overflow? char buf[1010]?
- Lees het probleem nog eens.
- Laat een teamgenoot het probleem herlezen.
- Gebruik de juiste epsilons om doubles te vergelijken.
- Maak je eigen testcases.