

Team reference document

sudo ku

Universiteit Leiden

Inhoudsopgave

1 Useful tables	2	4.4 Oppervlakte van een veelhoek . .	14
2 Bomen	2	4.5 Zwaartepunt van een veelhoek . .	15
2.1 Segment tree	2	4.6 Polygon	15
2.2 Fenwick tree	3	5 Dynamic Programming	17
2.3 Minimal spanning tree	4	5.1 Knapsack	17
2.4 UFDS	4	6 Strings	17
3 Grafen	4	6.1 String Matching	17
3.1 Depth First Search	4	6.2 Levenshteinafstand	17
3.2 Breadth first search	5	7 Getaltheorie	17
3.3 Topologisch sorteren	5	7.1 Priemgetallen	17
3.4 2-SAT	5	7.2 Uitgebreide Euclidische algoritme	18
3.5 Code 2-SAT, SCC en DFS	5	7.3 CRT	18
3.6 Biconnected components	6	7.4 Priemtest	18
3.7 Kortste pad	7	7.5 Partitiefunctie	20
3.7.1 Dijkstra's algoritme	7	8 Lineaire stelsels oplossen	20
3.7.2 Negatieve cykels	8	9 Tips	21
3.7.3 Floyd-Warshall	8	9.1 Mogelijke algoritmes, inspiratie .	21
3.8 Flow netwerken	9	9.2 Bugs	21
3.9 Max-flow	10	9.3 Minijudge	21
3.10 Min-cut	11	10 Toevoegingen sudo ku	22
4 Computational Geometry	11	10.1 Binary search	22
4.1 Geometry	11	10.2 Longest increasing subsequence .	22
4.2 Projecties	14	10.3 Code snippets	23
4.3 Rotaties	14	10.4 Bit operations	24
4.3.1 Snijden twee lijnstukken?	14	10.5 gdb debugger	24

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 Useful tables

maximale complexiteit		maximale grootte		Machten van 2	
Complexiteit	max. waarde n	data type	max. waarde	2-macht	dec. waarde
$\mathcal{O}(n!)$, $\mathcal{O}(n^6)$	10	int	$3.27 \cdot 10^4$	2^{10}	$1,02 \cdot 10^3$
$\mathcal{O}(2^n \cdot n^2)$	15	unsigned	$6.55 \cdot 10^4$	2^{20}	$1,04 \cdot 10^6$
$\mathcal{O}(n^4)$	100	long	$2.14 \cdot 10^9$	2^{30}	$1,07 \cdot 10^9$
$\mathcal{O}(n^3)$	400	unsigned long	$4.29 \cdot 10^9$	2^{40}	$1,10 \cdot 10^{12}$
$\mathcal{O}(n^2 \lg n)$	2000	long long	$9.22 \cdot 10^{18}$	2^{50}	$1,13 \cdot 10^{15}$
$\mathcal{O}(n^2)$	10.000	unsigned long	$1.84 \cdot 10^{19}$	2^{60}	$1,15 \cdot 10^{18}$
$\mathcal{O}(n \lg n)$	1000.000	float	7 digits		
$\mathcal{O}(n)$	100.000.000	double	15 digits		

2 Bomen

2.1 Segment tree

Onderstaande code werkt voor het dynamische Range Minimum Query probleem. Range updates kunnen worden geïmplementeerd met lazy propagation: houd bij of een segment volledig naar een bepaalde waarde geüpdated moet worden.

```

1 struct segment{
2     int num;
3     int beg, eind;
4     int minIndex;
5 };
6
7 class SegmentTree{
8 public:
9     SegmentTree(int n){
10         ar.assign(n, inf);
11         p.resize(3*n + 5); // to be sure
12         p[1].num = 1; p[1].beg = 0; p[1].eind = n-1; p[1].minIndex = 0;
13         for(unsigned int i = 2 ; i < p.size() ; i++){
14             p[i].num = i;
15             if( i%2 == 0 ){
16                 p[i].beg = p[i/2].beg; p[i].eind = (p[i/2].beg + p[i/2].eind)/2;}
17             else{
18                 p[i].beg = (p[i/2].beg + p[i/2].eind)/2 + 1; p[i].eind = p[i/2].eind;}
19             p[i].minIndex = p[i].beg;
20         }
21     }
22     void update(int index, int newValue){
23         ar[index] = newValue;
24         update(1, index, newValue);
25     }
26     void update(int num, int index, int newValue){
27         if( p[num].beg == p[num].eind )
28             return;
29         //if( ar[p[num].minIndex] > newValue)
30         int mid = (p[num].beg + p[num].eind)/2;
31         int num2;
32         if( mid >= index )
33             num *= 2;
34         else
35             num = 2*num+1;
36         num2 = num^1;

```

```

37     update(num, index, newValue);
38     p[num/2].minIndex = index;
39     if( ar[p[num/2].minIndex] < ar[p[num/2].minIndex])
40         p[num/2].minIndex = p[num/2].minIndex;
41     if( ar[p[num].minIndex] < ar[p[num/2].minIndex] )
42         p[num/2].minIndex = p[num].minIndex;
43
44 }
45 int rmq(int num, int beg, int eind, int a, int b){
46     if( beg >= a && eind <= b )
47         return p[num].minIndex;
48     if( beg > b || eind < a )
49         return -1;
50
51     int p1 = rmq(2*num, beg, (beg+eind)/2, a, b);
52     int p2 = rmq(2*num+1, (beg + eind)/2+1, eind, a, b);
53     if(p1 == -1)
54         return p2;
55     else{
56         if( p2 == -1)
57             return p1;
58         return (ar[p1] < ar[p2] ? p1 : p2);
59     }
60 }
61 int rmq(int a, int b){
62     return rmq(1,0,ar.size()-1,a,b);
63 }
64 vector<segment> p;
65 vi ar;
66 };

```

2.2 Fenwick tree

```

1 // Fenwick tree/binary indexed tree: dynamic range queries
2 // find sum of range:      lg(n)
3 // change value at index:  lg(n)
4 struct f_tree {
5     vl data;
6     ll n;          // size
7
8     f_tree(ll n) : n(n) {
9         data.assign(n, 0);
10    }
11
12    void add(ll idx, ll delta) {    // add to item at idx
13        for (; idx < n; idx = idx | (idx + 1))
14            data[idx] += delta;
15    }
16
17    ll sum(ll idx) {    // compute sum of elements 0,...,r
18        ll ret = 0;
19        for (; idx >= 0; idx = (idx & (idx + 1)) - 1)
20            ret += data[idx];
21        return ret;
22    }
23 };

```

2.3 Minimal spanning tree

```

1  int N, M;
2  struct edge { int u, v; int weight; } edges[MAX_EDGES];
3
4  bool cmp (const edge & e, const edge & f) { return e.weight < f.weight; }
5
6  void kruskal (void) {
7      sort (edges, edges + M, cmp); // sorteer kanten
8      for (int i = 0; i < N; i++) { init (i); } // elke knoop een eigen component
9
10     for (int i = 0; i < M; i++) { // itereer in oplopende volgorde
11         int u = representant (edges[i].u), // verkrijg representanten van de
12             v = representant (edges[i].v); // componenten van edges[j].{u,v}
13
14         if (u != v) { // als verschillende componenten: merge
15             add (edges[i]); // voeg toe aan minimum spanning tree in wording
16             merge (u, v);
17         }
18     }
19 }

```

Voor een efficiënte implementatie van `init`, `merge` en `representant` is een of andere Disjoint-Set datastructuur handig.

2.4 UFDS

Een Disjoint-Set Forest (met amortized $\mathcal{O}(\alpha(n))$ -tijd per operatie, met α de inverse van de Ackermann functie):

```

1  class UFDS {
2  private: vi p, rnk;
3  public:
4      UFDS(int N) { rnk.assign(N, 0);
5          p.assign(N, 0); for (int i=0 ; i< N; i++) p[i] = i; }
6      int findSet(int i) {
7          return (p[i] == i) ? i : (p[i] = findSet(p[i])); }
8      bool isSameSet(int i, int j) { return findSet(i) == findSet(j); }
9      void unionSet(int i, int j) {
10         if (!isSameSet(i,j)) {
11             int x = findSet(i), y = findSet(j);
12             if( rnk[x] > rnk[y]) p[y] = x;
13             else { p[x] = y;
14                 if (rnk[x] == rnk[y]) rnk[y]++; }
15         } } };

```

3 Grafen

3.1 Depth First Search

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

- Gerichte cykels: als een reeds actieve knoop wordt ontwikkeld.
- Ongerichte cykels: als een reeds actieve, of voltooide knoop wordt ontwikkeld. ($\mathcal{O}(V + E)$ en $\mathcal{O}(V^2)$)

```

1  vector<vi> AdjList //Adjacency list
2
3  vi dfs_num; //set all values to UNVISITED

```

```

4 void dfs(int u){
5     ii v;
6     dfs_num[u] = VISITED;
7     for(int j = 0; j < AdjList[u].size(); j++){
8         v = AdjList[u][j];
9         if (dfs_num[v] == UNVISITED){
10            dfs(v);DFS
11        }    }

```

3.2 Breadth first search

($\mathcal{O}(V + E)$ en $\mathcal{O}(V^2)$)

```

1 vector<vi> AdjList //Adjacency list
2 vi d(V, INF); d[s] = 0;
3 queue<int> q; q.push(s);
4 int u,v;
5 while(!q.empty()){
6     u = q.front(); q.pop(); // get the first element from our queue
7     for(int j = 0; j < AdjList[u].size(); j++){
8         v = AdjList[u][j]
9         if(d[v] == INF){ //test if unvisited
10            d[v] = d[u] + 1; //mark our visit
11            q.push(v);
12        }    }

```

3.3 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 variabele `vector<int> finished`. Hiermee is `finished[n-1] < ... < finished[0]` een topologische sortering van de knooppunten.

3.4 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.

3.5 Code 2-SAT, SCC en DFS

```

1 // the ith variable corresponds to nodes 2i and 2i+1 in the graph
2 // 2i is true if the variable is true, 2i+1 if it is false
3 struct graph {
4     vector<vector<int> > heen, terug; // the arguments of 2sat
5     graph(n) : heen(2*n), terug(2*n) {} // graph with nodes for n vars
6     int add_clause(int a, int b) { // a & b are nodes, adds clause (a or b)
7         heen[a^1].push_back(b); terug[b].push_back(a^1);
8         heen[b^1].push_back(a); terug[a].push_back(b^1);
9     }
10 }; // 2sat gives output for both nodes of a variable

1 template <typename S, typename F>
2 void dfs_visit (int cur, vector<vector<int> > const& buren, S start, F finish,
3     vector<bool> &mark) {
4     if (mark[cur]) { return; }
5     mark[cur] = true;

```

```

6
7     start (cur);
8     for (auto nb : buren[cur]) {
9         dfs_visit (nb, buren, start, finish, mark);
10    }
11    finish (cur);
12 }

1 vector<int> finish_sort(vector<vector<int> > const& buren) {
2     vector<int> finished; vector<bool> mark(buren.size());
3     int n = buren.size();
4     for (int i = 0; i < n; i++) {
5         dfs_visit (i, buren, [](int){},
6                     [&](int cur){finished.push_back(cur);}, mark);
7     }
8     return finished;
9 }

1 pair<int, vector<int> >
2 scc_and_top_sort(vector<vector<int> > const& heen, vector<vector<int> > const& terug) {
3     vector<int> comp(heen.size());
4     vector<bool> mark(heen.size(), false); int comp_n = 0;
5
6     auto finished = finish_sort(heen);
7     for (auto it = finished.rbegin(); it != finished.rend(); it++) {
8         if (mark[*it]) { continue; }
9         dfs_visit(*it, terug, [&](int cur){comp[cur] = comp_n;}, [](int){}, mark);
10        comp_n++;
11    }
12    return make_pair(comp_n, comp);
13 }

1 vector<bool> two_sat(vector<vector<int> > const& heen, vector<vector<int> > const& terug)
2     int n = heen.size();
3     vector<bool> val;
4     int comp_n; vector<int> comp;
5     vector<int> opp(n);
6
7     tie(comp_n, comp) = scc_and_top_sort(heen, terug);
8     for (int i=0; i<n; i++) { opp[comp[i]] = comp[i^1]; }
9     for (int i=0; i<comp_n; i++) { if (opp[i] == i) { return val; } }
10    vector<bool> cval(comp_n, false);
11    for (int i=0; i<comp_n; i++) { if (!cval[i]) { cval[opp[i]] = true; } }
12    for (int i=0; i<n; i++) { val.push_back(cval[comp[i]]); }
13    return val;
14 }

```

3.6 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.

```

1 vector<int> buren[MAX_NODES];
2 bool visited[MAX_NODES];
3 int low[MAX_NODES];
4 int d[MAX_NODES];
5 vector<pair<int, int> > st;
6

```

```

7 void mark(int a, int b) {
8     pair<int, int> e;
9     do {
10         e = st.back();
11         st.pop_back();
12         // doe iets met de tak
13     } while((e.first != a || e.second != b) && (e.first != b || e.second != a));
14 }
15
16 void dfs(int n, int parent, int& count) {
17     visited[n] = true;
18     low[n] = d[n] = ++count;
19     for(unsigned i = 0; i < buren[n].size(); ++i) {
20         int v = buren[n][i];
21         if(!visited[v]) {
22             st.push_back(make_pair(n,v));
23             dfs(v, n, count);
24             if(low[v] >= d[n]) {
25                 mark(n,v); // als n niet de root is dan n is een cut vertex
26             }
27             low[n] = min(low[n], low[v]);
28         } else if(parent != v && d[v] < d[n]) {
29             st.push_back(make_pair(n,v));
30             low[n] = min(low[n], d[v]);
31         }
32     }
33     // root == cut vertex <=> als er 2+ kinderen direct vanuit de root visited zijn.
34 }
35
36
37 void bicon(int N) {
38     int count = 0;
39     st.clear();
40     fill_n(visited, N, false);
41     for(unsigned i = 0; i < N; ++i)
42         if(!visited[i])
43             dfs(i, -1, count);
44 }

```

3.7 Kortste pad

3.7.1 Dijkstra's algoritme

Dijkstra's algoritme werkt alleen voor grafen met niet-negatieve gewichten. Een korte implementatie die knopen meerdermaals aan de PQ toe kan voegen is:

```

1 vi dist(V, INF); dist[s] = 0;
2 priority_queue<ii, vector<ii>, greater<ii> > pq; pq.push(ii(0,s));
3 while(!pq.empty()) {
4     ii front = pq.top(); pq.pop();
5     int d = front.first, u = front.second;
6     if(d > dist[u]) continue;
7     for (int j = 0; j < (int)AdjList[u].size(); j++){
8         ii v = AdjList[u][j];
9         if(dist[u] + v.second < dist[v.first]) {
10             dist[v.first] = dist[u] + v.second;
11             pq.push(ii(dist[v.first], v.first));
12 } } }

```

Deze implementatie hoort ook te werken met negatieve gewichten, hoewel langzamer. Pas wel op voor negatieve cyclen! met eigen ordening:

```

1 struct Order
2 {
3     bool operator()(ii const& a, ii const& b) const
4     {
5         return a.first > b.first; //or any other ordering
6     }
7 };
8 //...
9 priority_queue<ii, vector<ii>, Order > pq;
```

3.7.2 Negatieve cyclen

Bellman-Ford: relax elke edge V keer (makkelijk te implementeren met edge list). Negatieve-cykeldetectie: kijk of er na het uitvoeren nog iets te relaxen valt.

Kan ook met Floyd-Warshall/

Nota Bene: deze implementatie ondersteunt slechts paden van hoogstens $\text{INT_MAX}/4$!

```

1 template<class DistType>
2 struct pijl {
3     unsigned a,b;
4     DistType l; // signed!
5 };
6
7 template<class DistType>
8 bool bellmanford (unsigned s, unsigned n, vector<pijl<DistType>> const& pijlen,
9     vector<DistType> &dist) {
10     unsigned m = pijlen.size();
11
12     dist.clear();
13     dist.resize(n, numeric_limits<DistType>::max()/2);
14     dist[s] = 0;
15
16     for (unsigned i = 0; i < n; i++) {
17         for (unsigned j = 0; j < m; j++) {
18             dist[pijlen[j].b] = min(dist[pijlen[j].b], dist[pijlen[j].a] + pijlen[j].l);
19         }
20     }
21
22     // return alleen false bij negatieve kringen die bereikt kunnen worden
23     for (unsigned j = 0; j < m; j++) {
24         if (dist[pijlen[j].a] < numeric_limits<DistType>::max()/4 &&
25             dist[pijlen[j].b] > dist[pijlen[j].a] + pijlen[j].l)
26             return false;
27     }
28
29     return true;
30 }
```

3.7.3 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 {
```



```

51         q.push(make_pair(p.first + c, edges[e].dest));
52     }
53 }
54 }
55 if(pred[t] == t)
56     return false;
57
58 potential += dist;
59 return true;
60 }
61
62 vector<vector<unsigned>> out;
63 struct Edge {
64     unsigned dest;
65     Cap cap;
66     Cost cost;
67 };
68 vector<Edge> edges;
69
70 valarray<Cost> potential, dist;
71 vector<unsigned> pred;
72 };

```

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.

3.9 Max-flow

De Edmond-Karp implementatie van Ford-Fulkerson (mbv BFS)

```

1  int res[MAX_V][MAX_V], mf, f, s, t; // global variables
2  vi p;
3
4  void augment(int v, int minEdge) { // traverse BFS spanning tree from s to t
5      if (v == s) { f = minEdge; return; } // record minEdge in a global variable f
6      else if (p[v] != -1) { augment(p[v], min(minEdge, res[p[v]][v])); // recursive
7                          res[p[v]][v] -= f; res[v][p[v]] += f; } // update
8  }
9
10
11 //inside main()
12 mf = 0; // mf stands for max_flow
13 while (1) { // O(V^2) (actually O(V^3E) Edmonds Karp's algorithm
14     f = 0;
15     // run BFS, compare with the original BFS shown in Section 4.2.2
16     vi dist(MAX_V, INF); dist[s] = 0; queue<int> q; q.push(s);
17     p.assign(MAX_V, -1); // record the BFS spanning tree, from s to t!
18     while (!q.empty()) {
19         int u = q.front(); q.pop();
20         if (u == t) break; // immediately stop BFS if we already reach sink t
21         for (int v = 0; v < MAX_V; v++) // note: this part is slow
22             if (res[u][v] > 0 && dist[v] == INF)
23                 dist[v] = dist[u] + 1, q.push(v), p[v] = u;
24     }
25     augment(t, INF); // find the min edge weight 'f' along this path, if any
26     if (f == 0) break; // we cannot send any more flow ('f' = 0), terminate
27     mf += f; // we can still send a flow, increase the max flow!
28 }

```

3.10 Min-cut

Als je na max-flow kijk naar alle knopen die je kan bereiken vanaf de source via positieve ($\neq 0$) takken kan bereiken, heb je een 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.

4 Computational Geometry

4.1 Geometry

```

1 #define INF 1e9
2 #define EPS 1e-9
3 #define PI acos(-1.0) // important constant; alternative #define PI (2.0 * acos(0.0))
4
5 double DEG_to_RAD(double d) { return d * PI / 180.0; }
6
7 double RAD_to_DEG(double r) { return r * 180.0 / PI; }
8
9 // struct point_i { int x, y; }; // basic raw form, minimalist mode
10 struct point_i { int x, y; // whenever possible, work with point_i
11     point_i() { x = y = 0; } // default constructor
12     point_i(int _x, int _y) : x(_x), y(_y) {} }; // user-defined
13
14 struct point { double x, y; // only used if more precision is needed
15     point() { x = y = 0.0; } // default constructor
16     point(double _x, double _y) : x(_x), y(_y) {} // user-defined
17     bool operator < (point other) const { // override less than operator
18         if (fabs(x - other.x) > EPS) // useful for sorting
19             return x < other.x; // first criteria, by x-coordinate
20         return y < other.y; } // second criteria, by y-coordinate
21 // use EPS (1e-9) when testing equality of two floating points
22     bool operator == (point other) const {
23         return (fabs(x - other.x) < EPS && (fabs(y - other.y) < EPS)); } };
24
25 double dist(point p1, point p2) { // Euclidean distance
26     // hypot(dx, dy) returns sqrt(dx * dx + dy * dy)
27     return hypot(p1.x - p2.x, p1.y - p2.y); } // return double
28
29 // rotate p by theta degrees CCW w.r.t origin (0, 0)
30 point rotate(point p, double theta) {
31     double rad = DEG_to_RAD(theta); // multiply theta with PI / 180.0
32     return point(p.x * cos(rad) - p.y * sin(rad),
33         p.x * sin(rad) + p.y * cos(rad)); }
34
35 struct line { double a, b, c; }; // a way to represent a line
36
37 // the answer is stored in the third parameter (pass by reference)
38 void pointsToLine(point p1, point p2, line &l) {
39     if (fabs(p1.x - p2.x) < EPS) { // vertical line is fine
40         l.a = 1.0; l.b = 0.0; l.c = -p1.x; // default values
41     } else {
42         l.a = -(double)(p1.y - p2.y) / (p1.x - p2.x);
43         l.b = 1.0; // IMPORTANT: we fix the value of b to 1.0
44         l.c = -(double)(l.a * p1.x) - p1.y;
45     } }
46

```

```

47 // not needed since we will use the more robust form: ax + by + c = 0 (see above)
48 struct line2 { double m, c; }; // another way to represent a line
49
50 int pointsToLine2(point p1, point p2, line2 &l) {
51     if (abs(p1.x - p2.x) < EPS) { // special case: vertical line
52         l.m = INF; // l contains m = INF and c = x_value
53         l.c = p1.x; // to denote vertical line x = x_value
54         return 0; // we need this return variable to differentiate result
55     }
56     else {
57         l.m = (double)(p1.y - p2.y) / (p1.x - p2.x);
58         l.c = p1.y - l.m * p1.x;
59         return 1; // l contains m and c of the line equation y = mx + c
60     } }
61
62 bool areParallel(line l1, line l2) { // check coefficients a & b
63     return (fabs(l1.a-l2.a) < EPS) && (fabs(l1.b-l2.b) < EPS); }
64
65 bool areSame(line l1, line l2) { // also check coefficient c
66     return areParallel(l1, l2) && (fabs(l1.c - l2.c) < EPS); }
67
68 // returns true (+ intersection point) if two lines are intersect
69 bool areIntersect(line l1, line l2, point &p) {
70     if (areParallel(l1, l2)) return false; // no intersection
71     // solve system of 2 linear algebraic equations with 2 unknowns
72     p.x = (l2.b * l1.c - l1.b * l2.c) / (l2.a * l1.b - l1.a * l2.b);
73     // special case: test for vertical line to avoid division by zero
74     if (fabs(l1.b) > EPS) p.y = -(l1.a * p.x + l1.c);
75     else p.y = -(l2.a * p.x + l2.c);
76     return true; }
77
78 struct vec { double x, y; // name: 'vec' is different from STL vector
79     vec(double _x, double _y) : x(_x), y(_y) {} };
80
81 vec toVec(point a, point b) { // convert 2 points to vector a->b
82     return vec(b.x - a.x, b.y - a.y); }
83
84 vec scale(vec v, double s) { // nonnegative s = [<1 .. 1 .. >1]
85     return vec(v.x * s, v.y * s); } // shorter.same.longer
86
87 point translate(point p, vec v) { // translate p according to v
88     return point(p.x + v.x, p.y + v.y); }
89
90 // convert point and gradient/slope to line
91 void pointSlopeToLine(point p, double m, line &l) {
92     l.a = -m; // always -m
93     l.b = 1; // always 1
94     l.c = -((l.a * p.x) + (l.b * p.y)); // compute this
95
96 void closestPoint(line l, point p, point &ans) {
97     line perpendicular; // perpendicular to l and pass through p
98     if (fabs(l.b) < EPS) { // special case 1: vertical line
99         ans.x = -(l.c); ans.y = p.y; return; }
100
101     if (fabs(l.a) < EPS) { // special case 2: horizontal line
102         ans.x = p.x; ans.y = -(l.c); return; }
103
104     pointSlopeToLine(p, 1 / l.a, perpendicular); // normal line

```

```

105 // intersect line l with this perpendicular line
106 // the intersection point is the closest point
107 areIntersect(l, perpendicular, ans); }
108
109 // returns the reflection of point on a line
110 void reflectionPoint(line l, point p, point &ans) {
111     point b;
112     closestPoint(l, p, b); // similar to distToLine
113     vec v = toVec(p, b); // create a vector
114     ans = translate(translate(p, v), v); // translate p twice
115
116 double dot(vec a, vec b) { return (a.x * b.x + a.y * b.y); }
117
118 double norm_sq(vec v) { return v.x * v.x + v.y * v.y; }
119
120 // returns the distance from p to the line defined by
121 // two points a and b (a and b must be different)
122 // the closest point is stored in the 4th parameter (byref)
123 double distToLine(point p, point a, point b, point &c) {
124     // formula: c = a + u * ab
125     vec ap = toVec(a, p), ab = toVec(a, b);
126     double u = dot(ap, ab) / norm_sq(ab);
127     c = translate(a, scale(ab, u)); // translate a to c
128     return dist(p, c); } // Euclidean distance between p and c
129
130 // returns the distance from p to the line segment ab defined by
131 // two points a and b (still OK if a == b)
132 // the closest point is stored in the 4th parameter (byref)
133 double distToLineSegment(point p, point a, point b, point &c) {
134     vec ap = toVec(a, p), ab = toVec(a, b);
135     double u = dot(ap, ab) / norm_sq(ab);
136     if (u < 0.0) { c = point(a.x, a.y); // closer to a
137         return dist(p, a); } // Euclidean distance between p and a
138     if (u > 1.0) { c = point(b.x, b.y); // closer to b
139         return dist(p, b); } // Euclidean distance between p and b
140     return distToLine(p, a, b, c); } // run distToLine as above
141
142 double angle(point a, point o, point b) { // returns angle aob in rad
143     vec oa = toVec(o, a), ob = toVec(o, b);
144     return acos(dot(oa, ob) / sqrt(norm_sq(oa) * norm_sq(ob))); }
145
146 double cross(vec a, vec b) { return a.x * b.y - a.y * b.x; }
147
148 //// another variant
149 //int area2(point p, point q, point r) { // returns 'twice' the area of this triangle A-B-
150 //    return p.x * q.y - p.y * q.x +
151 //        q.x * r.y - q.y * r.x +
152 //        r.x * p.y - r.y * p.x;
153 //}
154
155 // returns true if point r is on the same line as the line pq
156 bool collinear(point p, point q, point r) {
157     return fabs(cross(toVec(p, q), toVec(p, r))) < EPS; }
158
159 // note: to accept collinear points, we have to change the '> 0'
160 // returns true if point r is on the left side of line pq
161 int ccw(point p, point q, point r) {
162     if( collinear(p,q,r))

```

```

163 |         return 0;
164 |         return (cross(toVec(p, q), toVec(p, r)) > 0 ? 1 : -1);
165 |     }

```

4.2 Projecties

```

1 | 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:

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

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

4.3 Rotaties

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

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

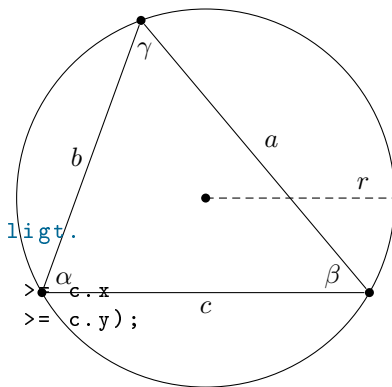
4.3.1 Snijden twee lijnstukken?

Er zijn een paar vervelende randgevallen. Onderstaande code (functie `segmentsIntersect`) werkt. Deze functie onderzoekt of de lijnstukken $\overrightarrow{p_1 p_2}$ en $\overrightarrow{p_3 p_4}$ elkaar snijden. De code werkt ook als de coördinaten in float of double zijn.

```

1 | // Check of c op het lijnstuk van a naar b ligt,
2 | // gegeven het feit dat c op de lijn door a en b ligt.
3 | bool onSegment(punt a, punt b, punt c) {
4 |     return (min(a.x, b.x) <= c.x && max(a.x, b.x) >=
5 |             && min(a.y, b.y) <= c.y && max(a.y, b.y) >=
6 |             c.x && c.y);
7 | }
8 |
9 | bool segmentsIntersect (punt p1, punt p2, punt p3, punt p4) {
10 |     int d1 = ccw(p3, p4, p1);
11 |     int d2 = ccw(p3, p4, p2);
12 |     int d3 = ccw(p1, p2, p3);
13 |     int d4 = ccw(p1, p2, p4);
14 |
15 |     if (d1 * d2 < 0 && d3 * d4 < 0) return true;
16 |     if (d1 == 0 && onSegment(p3, p4, p1)) return true;
17 |     if (d2 == 0 && onSegment(p3, p4, p2)) return true;
18 |     if (d3 == 0 && onSegment(p1, p2, p3)) return true;
19 |     if (d4 == 0 && onSegment(p1, p2, p4)) return true;
20 |     return false;

```



4.4 Oppervlakte van een veelhoek

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

4.5 Zwaartepunt van een veelhoek

Het zwaartepunt kan bepaald worden met

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

En

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

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

4.6 Polygon

```

1 // returns the perimeter, which is the sum of Euclidian distances
2 // of consecutive line segments (polygon edges)
3 double perimeter(const vector<point> &P) {
4     double result = 0.0;
5     for (int i = 0; i < (int)P.size()-1; i++) // remember that P[0] = P[n-1]
6         result += dist(P[i], P[i+1]);
7     return result; }
8
9 // returns the area, which is half the determinant
10 double area(const vector<point> &P) {
11     double result = 0.0, x1, y1, x2, y2;
12     for (int i = 0; i < (int)P.size()-1; i++) {
13         x1 = P[i].x; x2 = P[i+1].x;
14         y1 = P[i].y; y2 = P[i+1].y;
15         result += (x1 * y2 - x2 * y1);
16     }
17     return fabs(result) / 2.0; }
18
19 // returns true if we always make the same turn while examining
20 // all the edges of the polygon one by one
21 bool isConvex(const vector<point> &P) {
22     int sz = (int)P.size();
23     if (sz <= 3) return false; // a point/sz=2 or a line/sz=3 is not convex
24     bool isLeft = ccw(P[0], P[1], P[2]); // remember one result
25     for (int i = 1; i < sz-1; i++) // then compare with the others
26         if (ccw(P[i], P[i+1], P[(i+2) % sz]) != isLeft)
27             return false; // different sign -> this polygon is concave
28     return true; // this polygon is convex
29
30 // returns true if point p is in either convex/concave polygon P
31 bool inPolygon(point pt, const vector<point> &P) {
32     if ((int)P.size() == 0) return false;
33     double sum = 0; // assume the first vertex is equal to the last vertex
34     for (int i = 0; i < (int)P.size()-1; i++) {
35         if (ccw(pt, P[i], P[i+1]))
36             sum += angle(P[i], pt, P[i+1]); // left turn/ccw
37         else sum -= angle(P[i], pt, P[i+1]); // right turn/cw
38     }
39     return fabs(fabs(sum) - 2*PI) < EPS; }
40
41 // line segment p-q intersect with line A-B.
42 point lineIntersectSeg(point p, point q, point A, point B) {
43     double a = B.y - A.y;
44     double b = A.x - B.x;

```

```

44 double c = B.x * A.y - A.x * B.y;
45 double u = fabs(a * p.x + b * p.y + c);
46 double v = fabs(a * q.x + b * q.y + c);
47 return point((p.x * v + q.x * u) / (u+v), (p.y * v + q.y * u) / (u+v)); }
48
49 // cuts polygon Q along the line formed by point a -> point b
50 // (note: the last point must be the same as the first point)
51 vector<point> cutPolygon(point a, point b, const vector<point> &Q) {
52     vector<point> P;
53     for (int i = 0; i < (int)Q.size(); i++) {
54         double left1 = cross(toVec(a, b), toVec(a, Q[i])), left2 = 0;
55         if (i != (int)Q.size()-1) left2 = cross(toVec(a, b), toVec(a, Q[i+1]));
56         if (left1 > -EPS) P.push_back(Q[i]); // Q[i] is on the left of ab
57         if (left1 * left2 < -EPS) // edge (Q[i], Q[i+1]) crosses line ab
58             P.push_back(lineIntersectSeg(Q[i], Q[i+1], a, b));
59     }
60     if (!P.empty() && !(P.back() == P.front()))
61         P.push_back(P.front()); // make P's first point = P's last point
62     return P; }
63
64 point pivot;
65 bool angleCmp(point a, point b) { // angle-sorting function
66     if (collinear(pivot, a, b)) // special case
67         return dist(pivot, a) < dist(pivot, b); // check which one is closer
68     double d1x = a.x - pivot.x, d1y = a.y - pivot.y;
69     double d2x = b.x - pivot.x, d2y = b.y - pivot.y;
70     return (atan2(d1y, d1x) - atan2(d2y, d2x)) < 0; } // compare two angles
71
72 vector<point> CH(vector<point> P) { // the content of P may be reshuffled
73     int i, j, n = (int)P.size();
74     if (n <= 3) {
75         if (!P[0] == P[n-1]) P.push_back(P[0]); // safeguard from corner case
76         return P; // special case, the CH is P itself
77     }
78
79     // first, find P0 = point with lowest Y and if tie: rightmost X
80     int P0 = 0;
81     for (i = 1; i < n; i++)
82         if (P[i].y < P[P0].y || (P[i].y == P[P0].y && P[i].x > P[P0].x))
83             P0 = i;
84
85     point temp = P[0]; P[0] = P[P0]; P[P0] = temp; // swap P[P0] with P[0]
86
87     // second, sort points by angle w.r.t. pivot P0
88     pivot = P[0]; // use this global variable as reference
89     sort(++P.begin(), P.end(), angleCmp); // we do not sort P[0]
90
91     // third, the ccw tests
92     vector<point> S;
93     S.push_back(P[n-1]); S.push_back(P[0]); S.push_back(P[1]); // initial S
94     i = 2; // then, we check the rest
95     while (i < n) { // note: N must be >= 3 for this method to work
96         j = (int)S.size()-1;
97         if (ccw(S[j-1], S[j], P[i])) S.push_back(P[i++]); // left turn, accept
98         else S.pop_back(); // or pop the top of S until we have a left turn
99     }
    return S; }

```


5 Dynamic Programming

5.1 Knapsack

```

1 // max value under constraint upper bound on weight
2 int knapsack( vector<int> value, vector<int> weight, int max_weight ) {
3     int n = value.size();
4     vector<vector<int>> m( n+1, vector<int>( max_weight+1, 0 ) );
5     for( int i = 0; i < n; ++i )
6         for( int j = 0; j <= max_weight; ++j )
7             if( weight[i] > j )
8                 m[i+1][j] = m[i][j];
9             else
10                m[i+1][j] = max( m[i][j], m[i][j-weight[i]] + value[i] );
11     return m[n][max_weight];
12 }
```

6 Strings

6.1 String Matching

```

1 // s: haystack, w: needle, cb: match callback with word start
2 template<class F>
3 void match_string(string const& s, string const& w, F&& cb) {
4     assert(!w.empty());
5     vector<int> f(w.size() + 1, 0);
6     for(unsigned i = 2, c = 0; i <= w.size(); i++) {
7         if(w[i-1] == w[c]) f[i++] = ++c;
8         else if(c > 0) c = f[c];
9         else ++i;
10    }
11    for(unsigned i = 0, q = 0; i < s.size(); ++i) {
12        while(q > 0 && (q == w.size() || s[i] != w[q])) q = f[q];
13        if(w[q] == s[i]) ++q;
14        if(q == w.size()) cb(i + 1 - w.size());
15    }
16 }
```

6.2 Levenšteinafstand

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

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

7 Getaltheorie

7.1 Priemgetallen

```

1 void sieve(ll upperbound) { // create list of primes in [0..upperbound]
2     _sieve_size = upperbound + 1; // add 1 to include upperbound
3     bs.set(); // set all bits to 1
4     bs[0] = bs[1] = 0; // except index 0 and 1
```

```

5   for (ll i = 2; i <= _sieve_size; i++) if (bs[i]) {
6       // cross out multiples of i starting from i * i!
7       for (ll j = i * i; j <= _sieve_size; j += i) bs[j] = 0;
8       primes.push_back((int)i); // also add this vector containing list of primes
9   } } // call this method in main method
10
11 bool isPrime(ll N) { // a good enough deterministic prime tester
12     if (N <= _sieve_size) return bs[N]; // O(1) for small primes
13     for (int i = 0; i < (int)primes.size(); i++)
14         if (N % primes[i] == 0) return false;
15     return true; // it takes longer time if N is a large prime!
16 } // note: only work for N <= (last prime in vi "primes")^2

```

7.2 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 $ax + by = \text{ggd}(a, b)$.

```

1   template<class T>
2   pair<T, pair<T, T>> uggd(T a, T b) {
3       T x, lastx, y, lasty;
4       T q;
5
6       x = lasty = 0;
7       y = lastx = 1;
8
9       while (b != 0) {
10          q = a / b;
11
12          a %= b;
13          swap(a, b);
14
15          lastx -= q*x;
16          swap(x, lastx);
17
18          lasty -= q*y;
19          swap(y, lasty);
20      }
21
22      return make_pair (a, make_pair (lastx, lasty));
23  }

```

```

1   unsigned ggd(unsigned a, unsigned b) {
2       while(b) {
3           a %= b;
4           swap(a, b);
5       }
6       return a;
7   }

```

7.3 CRT

```

1   template<class T>
2   T crt(T a, T b, T m, T n){ //assumes m, n are coprime
3       pair<T, T> multinv = uggd(m, n).second;
4       T x = b*((multinv.first*m)%(m*n)) + a*((multinv.second*n)%(m*n));
5       x = (x%(m*n) + m*n)%(m*n);
6       return x;
7   } // returns 0 <= x < m*n with x = a (mod m), x = b (mod n)

```

7.4 Priemtest

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

```

1   __int128_t power(__int128_t a, unsigned long long e, unsigned long long n) {
2       if(e == 1) return a;
3       if(e & 1) return (a * power(a, e-1, n) % n);

```

```

4     __int128_t v = power(a, e/2, n);
5     return (v*v) % n;
6 }
7
8 bool isprime(unsigned long long n) {
9     if(n <= 1) return false;
10    unsigned long long d = n-1;
11    unsigned s = 0;
12    while(d % 2 == 0) { d /= 2; ++s; }
13    // BELANGRIJK: typ deze getallen goed over!
14    const unsigned a_arr[] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};
15    for(unsigned i = 0; i < sizeof(a_arr)/sizeof(unsigned) && a_arr[i] < n; ++i) {
16        unsigned a = a_arr[i];
17        __int128_t v = power(a, d, n);
18        if(v == 1) continue;
19        bool composite = true;
20        for(unsigned p = 0; p < s; ++p) {
21            if(v == n-1) { composite = false; break; }
22            v = (v * v) % n;
23        }
24        if(composite) return false;
25    }
26    return true;
27 }

```

Dit gebruiken we om te voorkomen dat we grote priemgetallen in Pollards-Rho gaan gooien

```

1 #define abs_val(a) (((a)>=0)?(a):- (a))
2 ll mulmod(ll a, ll b, ll c) { // returns (a * b) % c, and minimize overflow
3     ll x = 0, y = a % c;
4     while (b > 0) {
5         if (b % 2 == 1) x = (x + y) % c;
6         y = (y * 2) % c;
7         b /= 2;
8     }
9     return x % c;
10 }
11
12 ll gcd(ll a, ll b) { return !b ? a : gcd(b, a % b); } // standard gcd
13
14 ll pollard_rho(ll n) {
15     int i = 0, k = 2;
16     ll x = 3, y = 3; // random seed = 3, other values possible
17     while (1) {
18         i++;
19         x = (mulmod(x, x, n) + n - 1) % n; // generating function
20         ll d = gcd(abs_val(y - x), n); // the key insight
21         if (d != 1 && d != n) return d; // found one non-trivial factor
22         if (i == k) y = x, k *= 2;
23     } }
24
25 int main() {
26     ll n = 2063512844981574047LL; // we assume that n is not a large prime
27     ll ans = pollard_rho(n); // break n into two non trivial factors
28     if (ans > n / ans) ans = n / ans; // make ans the smaller factor
29     printf("%lld %lld\n", ans, n / ans); // should be: 1112041493 1855607779
30 } // return 0;

```

7.5 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.

```

1 long long parts[P][P]; // [m][n] = # opdelingen van n die overall <= m zijn
2
3 long long partition(int upper) {
4     parts[0][0] = 1;
5     fill_n (parts[0] + 1, upper, 0);
6     for (int n = 1; n <= upper; n++) {
7         parts[n][0] = 1;
8         for (int sum = 1; sum <= upper; sum++) {
9             parts[n][sum] = parts[n - 1][sum];
10            if (sum >= n)
11                parts[n][sum] += parts[n][sum - n];
12        }
13    }
14    return parts[upper][upper];
15 }

```

Overflow

type	iteraties veilig
s32	121
u32	127
s64	405
u64	416
sl28	1437
ul28	1458

8 Lineaire stelsels oplossen

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

```

1
2
3 bool linsolve(vector<vector<double> > A, vector<double> b, vector<double>& x,
4             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 Tips

9.1 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
- longest increasing subsequence
- Kan je de graaf bipartiet krijgen?
- Precalculeren
- Inclusion/Exclusion
- Neigt het naar NIM?
- Line sweep / radial sweep
- Coördinaten comprimeren
- Vervang strings met iets snellers.

9.2 Bugs

- Maak je aannamens die niet in de probleembeschrijving staan?
- Kijk uit voor variabelen met dezelfde naam.
- Geef je altijd de juiste uitvoer? (e.g.: geen pad, print "impossible")
- Ergens een overflow? Array out of bounds?
- *Gebruik de juiste epsilons om doubles te vergelijken.*
- Zijn er belangrijke regels in commentaar gezet?
- Gebruik je overal groot genoeg variabelen (long long)? Ook bij bit operations?
- Lees het probleem nog eens (is de input gesorteerd of niet?).
- Laat een teamgenoot het probleem herlezen.
- Maak je eigen testcases (wat gebeurt er als de input ergens 0 is?).
- Controleer je of je nodes al hebt bezocht?

9.3 Minijudge

```

1  #!/bin/bash
2
3  TESTF=~ /final_samples
4
5  c++ -g -o $1 $1.cc || {
6      echo "COMPILE ERROR"
7      exit
8  }
9
10 for file in $TESTF/$2/*.in; do
11     ./$1 < $file > out.tst || {
12         echo "RUN ERROR"
13         exit
14     }
15     diff out.tst "${file%.in}.ans" > /dev/null 2>/dev/null || {
16         echo "Output:"
17         cat $file
18         echo "Diff:"
19         diff out.tst "${file%.in}.ans" -y
20     }
21     rm ./out.tst
22 done

```

10 Toevoegingen sudo ku

10.1 Binary search

binary search (voor een functie *can* die *false* is voor $x < ans$ en *true* voor $x \geq ans$)

```

1 bool can(double x){
2 //test is x is a solution (or larger than the solution)
3 }
4 ...
5 //instide main()
6 double lo = 0.0, hi = 100000.0, mid = 0.0, ans = 0.0;
7 double eps = 0.01;
8
9 while(fabs(hi-lo) > eps){
10 mid = (lo + hi) / 2.0;
11 if(can(mid)) {ans = mid; high = mid;}
12 //save the answer, look for smaller solution
13 else {lo = mid;} //look for larger solution
14 }

```

sort using a custom function object

```

1 struct {
2 bool operator()(int a, int b) const{
3 return a < b;
4 }
5 } customLess;
6
7 array<int, 10> s = {5, 7, 4, 2, 8, 6, 1, 9, 0, 3};
8 sort(s.begin(), s.end(), customLess);

```

10.2 Longest increasing subsequence

```

1 void find_lis(vector<int> &a, vector<int> &b) {
2     vector<int> p(a.size());
3     int u, v;
4     if (a.empty()) return;
5     b.push_back(0);
6     for (size_t i = 1; i < a.size(); i++) {
7         if (a[b.back()] < a[i]) { //If next element a[i] is greater than last
8             p[i] = b.back(); // element of current longest subsequence
9             b.push_back(i); // a[b.back()], just push it at back of "b"
10            continue;
11        }
12        // search for smallest element referenced by b just bigger than a[i]
13        for (u = 0, v = b.size()-1; u < v;) {
14            int c = (u + v) / 2;
15            if (a[b[c]] < a[i]) u=c+1; else v=c;
16        }
17        if (a[i] < a[b[u]]) { // Update b if new value is smaller
18            if (u > 0) p[i] = b[u-1]; // then previously referenced value
19            b[u] = i;
20        }
21    }
22    for (u = b.size(), v = b.back(); u-- > v;) b[u] = p[v];
23 }

```

10.3 Code snippets

vimrc

```
1 | execute pathogen#infect()
2 | filetype plugin indent on
3 |
4 | set number
5 | set tabstop=4
6 | set shiftwidth=4
```

Template

```
1 | #include <bits/stdc++.h>
2 | using namespace std;
3 |
4 | #define pb push_back
5 | #define eb emplace_back
6 | #define mp make_pair
7 | #define pq priority_queue
8 |
9 | using ll = long long;           // integral value type
10 | using ff = long double;        // floating point type
11 | using vl = vector<ll>;
12 | using pl = pair<ll, ll>;
13 | using vpl = vector<pl>;
14 |
15 | int main() {
16 |     ios::sync_with_stdio(false);
17 |     cin.tie(NULL);
18 |     cout << fixed << setprecision(10);
19 |
20 |     ll N; cin >> N;
21 | }
```

zet alle elementen van een (multi-dimensionale) array naar *value*:

```
1 | memset(array, value, sizeof array);
```

of voor longs

```
1 | void memset64( void * dest, uint64_t value, uintptr_t size ){
2 |     uintptr_t i;
3 |     for( i = 0; i < (size & (~7)); i+=8 ){
4 |         memcpy( ((char*)dest) + i, &value, 8 );
5 |     }
6 |     for( ; i < size; i++ ){
7 |         ((char*)dest)[i] = ((char*)&value)[i&7];
8 |     } }
```

geef een decimale precisie voor cout ($n = 4$ geeft $\pi = 3.1415$)

```
1 | cout << setprecision(n);
```

Bekijk alle permutaties van een array, in complexiteit $\mathcal{O}(n!)$:

```
1 | int n = 4;
2 | int p[n] = {0,1,2,3};
3 |
4 | do{
5 |     //your code here
6 | }while(next_permutation(p,p+n));
```

Bekijk alle subset van een array, in complexiteit $\mathcal{O}(2^n)$:

```

1 //given an array "array" with size "arraySize"
2 unsigned int i, j, bits, i_max = 1U << arraySize;
3
4 for (i = 0; i < i_max ; ++i) {
5     for (bits = i, j = 0; bits; bits >>= 1, ++j) {
6         if (bits & 1){
7             //do something for each element in subset
8         }
5     }
4 }

```

10.4 Bit operations

Een bitset is een meer geheugenvriendelijke boolean array.

```

1 |bitset<8> b1;/* [0,0,0,0,0,0,0,0] */ bitset<8> b2(42); // [0,0,1,0,1,0,1,0]

```

<pre> 1 int S; 2 S << 1; //S*2 3 S << 2; //S*4 4 S >> 1; //S/2 5 S >> 2; //S/4 6 1 << 3; //001000 7 A B; //bitwise or </pre>	<pre> 8 A & B; //bitwise and 9 A ^ B; //bitwise xor 10 S = (1 << j); //ensure jth bit is on 11 S &= ~(1 << j); //ensure jth bit is off 12 S ^= (1 << j); //toggle jth bit 13 S & (1 << j) != 0; //test jth bit </pre>
---	--

10.5 gdb debugger

Compileer je programma met

```
g++ -g -Wall -o test test.cc
```

Open GDB met

```
gdb test
```

Dit zijn de belangrijkste gdb commando's:

- b** break REGELNUMMER / FUNCTIENAAM (stelt een breakpoint in; de executie zal worden onderbroken vóórdat de regel is uitgevoerd)
- wa** watch VARIABLE / CONDITIE (het programma zal worden onderbroken als de waarde van de variabele wordt veranderd)
- i b** info breakpoints (geeft een genummerde lijst van alle breakpoints)
- dis/en** disable/enable BREAKPOINTNUMMER (het breakpoint blijft bestaan onder hetzelfde nummer, maar zal in de executie worden genegeerd)
- r** run < infile.in > outfile.out (mag ook zonder pipes, dan van stdin/stdout)
- c** continue (tot volgende breakpoint)
- s** step (voer de volgende stap uit, gaat zonodig de functie in)
- n** next (voer de volgende regel code uit)
- u** until REGELNUMMER (ga door tot de genoemde regel)
- k** kill (maar blijf in gdb)
- q** quit (verlaat gdb)
- ba/bt** backtrace (print een stack trace, zodat je ziet in welke functie je zit)
 - f** frame NUMMER-IN-DE-STACK (verplaatst de scope van gdb naar de gegeven plek in de stack trace)
 - p** print NAAM-VAN-VARIABLE (eerste 10 elementen van array a bekijken: print *a@10)
 - l** list (print de huidige regel broncode met een paar regels context)

10.6 Priemgetallen

6.15. Stelling. *Chinese Reststelling* Laat m en n onderling ondeelbare gehele getallen zijn. Dan is de natuurlijke afbeelding

$$\begin{aligned} \psi : \mathbf{Z}/mn\mathbf{Z} &\xrightarrow{\sim} \mathbf{Z}/m\mathbf{Z} \times \mathbf{Z}/n\mathbf{Z} \\ (x \bmod mn) &\mapsto (x \bmod m, x \bmod n) \end{aligned}$$

een ringisomorfisme. De afbeelding ψ induceert een isomorfisme van eenhedengroepen

$$\psi_* : (\mathbf{Z}/mn\mathbf{Z})^* \xrightarrow{\sim} (\mathbf{Z}/m\mathbf{Z})^* \times (\mathbf{Z}/n\mathbf{Z})^*,$$

en de Euler- φ -functie voldoet aan $\varphi(mn) = \varphi(m)\varphi(n)$.

6.16. Gevolg. Voor ieder positief geheel getal n is er een natuurlijk ringisomorfisme

$$\mathbf{Z}/n\mathbf{Z} \xrightarrow{\sim} \prod_{p|n} (\mathbf{Z}/p^{\text{ord}_p(n)}\mathbf{Z}).$$

Voor de Euler- φ -functie geldt dienovereenkomstig

$$\varphi(n) = \prod_{p|n} \varphi(p^{\text{ord}_p(n)}) = \prod_{p|n} (p-1)p^{\text{ord}_p(n)-1} = n \cdot \prod_{p|n} \left(1 - \frac{1}{p}\right).$$

6.17. Stelling. Voor a en $n \geq 1$ onderling ondeelbaar geldt $a^{\varphi(n)} \equiv 1 \bmod n$. □

Het geval dat n een priemgetal is was al bestudeerd door Fermat (1601–1665), die rond 1640 het volgende speciale geval van 6.17 formuleerde.

6.18. Kleine stelling van Fermat. Voor p een priemgetal en a een geheel getal geldt

$$a^p \equiv a \bmod p.$$

