

Team Contest Reference

ChaosKITs
Karlsruhe Institute of Technology

14. Februar 2016

Inhaltsverzeichnis

1	Datenstrukturen	2		
1.1	Union-Find	2		
1.2	Segmentbaum	2		
1.3	Fenwick Tree	3		
1.4	Range Minimum Query	3		
1.5	STL-Tree	3		
2	Graphen	4		
2.1	Minimale Spannbäume	4		
2.1.1	Kruskal	4		
2.2	Kürzeste Wege	4		
2.2.1	Algorithmus von DIJKSTRA	4		
2.2.2	BELLMANN-FORD-Algorithmus	5		
2.2.3	FLOYD-WARSHALL-Algorithmus	5		
2.3	Strongly Connected Components (TARJANS-Algorithmus)	5		
2.4	Artikulationspunkte und Brücken	6		
2.5	Eulertouren	7		
2.6	Lowest Common Ancestor	8		
2.7	Max-Flow	8		
2.7.1	Capacity Scaling	8		
2.7.2	Push Relabel	9		
2.7.3	Anwendungen	10		
2.8	Min-Cost-Max-Flow	10		
2.9	Maximal Cardinatlity Bipartite Matching	11		
2.10	TSP	12		
2.11	Bitonic TSP	12		
3	Geometrie	13		
3.1	Closest Pair	13		
3.2	Geraden	13		
3.3	Konvexe Hülle	14		
3.4	Formeln - std::complex	14		
4	Mathe	17		
4.1	ggT, kgV, erweiterter euklidischer Algorithmus	17		
			4.1.1	Multiplikatives Inverses von x in $\mathbb{Z}/n\mathbb{Z}$
				17
			4.2	Primzahlsieb von ERATOSTHENES
				17
			4.3	MILLER-RABIN-Primzahltest
				18
			4.4	Faktorisierung
				18
			4.5	Mod-Exponent über \mathbb{F}_p
				18
			4.6	LGS über \mathbb{F}_p
				19
			4.7	Binomialkoeffizienten
				19
			4.8	Satz von SPRAGUE-GRUNDY
				19
			4.9	Maximales Teilfeld
				20
			4.10	Polynome & FFT
				20
			4.11	Kombinatorik
				21
			4.11.1	Berühmte Zahlen
				21
			4.11.2	Verschiedenes
				22
5	Strings	22		
5.1	KNUTH-MORRIS-PRATT-Algorithmus	22		
5.2	AHO-CORASICK-Automat	22		
5.3	LEVENSHTEIN-Distanz	23		
5.4	Trie	23		
5.5	Suffix-Array	24		
5.6	Longest Common Substring	24		
5.7	Longest Common Subsequence	25		
6	Java	25		
6.1	Introduction	25		
6.2	BigInteger	25		
7	Sonstiges	26		
7.1	2-SAT	26		
7.2	Bit Operations	26		
7.3	Josephus-Problem	26		
7.4	Gemischtes	27		
8	Convenience-Methoden	27		
8.1	Zeileneingabe	27		
8.2	Template	27		
8.3	Sonstiges	28		

1 Datenstrukturen

1.1 Union-Find

```

1 // Laufzeit:  $O(n \cdot \alpha(n))$ 
2 // "height" ist obere Schranke für die Höhe der Bäume. Sobald Pfadkompression
3 // angewendet wurde, ist die genaue Höhe nicht mehr effizient berechenbar.
4 vector<int> parent // Initialisiere mit Index im Array.
5 vector<int> height; // Initialisiere mit 0.
6
7 int findSet(int n) { // Pfadkompression
8     if (parent[n] != n) parent[n] = findSet(parent[n]);
9     return parent[n];
10 }
11
12 void linkSets(int a, int b) { // Union by rank.
13     if (height[a] < height[b]) parent[a] = b;
14     else if (height[b] < height[a]) parent[b] = a;
15     else {
16         parent[a] = b;
17         height[b]++;
18     }
19 }
20
21 void unionSets(int a, int b) { // Diese Funktion aufrufen.
22     if (findSet(a) != findSet(b)) linkSets(findSet(a), findSet(b));
23 }

```

1.2 Segmentbaum

```

1 // Laufzeit: init:  $O(n)$ , query:  $O(\log n)$ , update:  $O(\log n)$ 
2 // Berechnet das Maximum im Array.
3 int a[MAX_N], m[4 * MAX_N];
4
5 int query(int x, int y, int k = 0, int X = 0, int Y = MAX_N - 1) {
6     if (x <= X && Y <= y) return m[k];
7     if (y < X || Y < x) return -1000000000; // Ein "neutrales" Element.
8     int M = (X + Y) / 2;
9     return max(query(x, y, 2 * k + 1, X, M), query(x, y, 2 * k + 2, M + 1, Y));
10 }
11
12 void update(int i, int v, int k = 0, int X = 0, int Y = MAX_N - 1) {
13     if (i < X || Y < i) return;
14     if (X == Y) {
15         m[k] = v;
16         a[i] = v;
17         return;
18     }
19     int M = (X + Y) / 2;
20     update(i, v, 2 * k + 1, X, M);
21     update(i, v, 2 * k + 2, M + 1, Y);
22     m[k] = max(m[2 * k + 1], m[2 * k + 2]);
23 }
24
25 // Einmal vor allen anderen Operationen aufrufen.
26 void init(int k = 0, int X = 0, int Y = MAX_N - 1) {
27     if (X == Y) {
28         m[k] = a[X];
29         return;
30     }
31     int M = (X + Y) / 2;
32     init(2 * k + 1, X, M);
33     init(2 * k + 2, M + 1, Y);
34     m[k] = max(m[2 * k + 1], m[2 * k + 2]);
35 }

```

update() kann so umgeschrieben werden, dass ganze Intervalle geändert werden. Dazu muss ein Offset in den inneren Knoten des Baums gespeichert werden.

1.3 Fenwick Tree

```

1  vector<int> FT; //Fenwick-Tree
2
3  //Build an Fenwick-Tree over an array a. Time Complexity: O(n*log(n))
4  buildFenwickTree(vector<int>& a) {
5      n = a.size();
6      FT.assign(n+1,0);
7      for(int i = 0; i < n; i++) updateFT(i,a[i]);
8  }
9
10 //Prefix-Sum of intervall [0..i]. Time Complexity: O(log(n))
11 int prefix_sum(int i) {
12     int sum = 0; i++;
13     while(i > 0) { sum += FT[i]; i -= (i & (-i)); }
14     return sum;
15 }
16
17 //Adds val to index i. Time Complexity O(log(n))
18 void updateFT(int i, int val) {
19     i++; while(i <= n) { FT[i] += val; i += (i & (-i)); }
20 }

```

1.4 Range Minimum Query

```

1  vector<int> data(RMQ_SIZE);
2  vector<vector<int>> rmq(floor(log2(RMQ_SIZE)) + 1, vector<int>(RMQ_SIZE));
3
4  //Runtime: O(n*log(n))
5  void initRMQ() {
6      for(int i = 0, s = 1, ss = 1; s <= RMQ_SIZE; ss=s, s*=2, i++) {
7          for(int l = 0; l + s <= RMQ_SIZE; l++) {
8              if(i == 0) rmq[0][l] = l;
9              else {
10                 int r = l + ss;
11                 rmq[i][l] = (data[rmq[i-1][l]] <= data[rmq[i-1][r]] ? rmq[i-1][l] : rmq[i-1][r]);
12             }
13         }
14     }
15 }
16 //returns index of minimum! [l, r)
17 //Runtime: O(1)
18 int queryRMQ(int l, int r) {
19     if(l >= r) return l;
20     int s = floor(log2(r-l)); r = r - (1 << s);
21     return (data[rmq[s][l]] <= data[rmq[s][r]] ? rmq[s][l] : rmq[s][r]);
22 }

```

1.5 STL-Tree

```

1  #include <bits/stdc++.h>
2  #include <ext/pb_ds/assoc_container.hpp>
3  #include <ext/pb_ds/tree_policy.hpp>
4  using namespace std; using namespace __gnu_pbds;
5  typedef tree<int, null_type, less<int>, rb_tree_tag, tree_order_statistics_node_update> Tree;
6  int main() {
7      Tree X;
8      for (int i = 1; i <= 16; i <= 1) X.insert(i); // {1, 2, 4, 8, 16}
9      cout << *X.find_by_order(3) << endl; // => 8
10     cout << X.order_of_key(10) << endl; // => 4 = successor of 10 = min i such that X[i] >= 10
11     return 0;
12 }

```

2 Graphen

2.1 Minimale Spannbäume

Benutze Algorithmus von KRUSKAL oder Algorithmus von PRIM.

Schnitteigenschaft Für jeden Schnitt C im Graphen gilt: Gibt es eine Kante e , die echt leichter ist als alle anderen Schnittkanten, so gehört diese zu allen minimalen Spannbäumen. (\Rightarrow Die leichteste Kante in einem Schnitt kann in einem minimalen Spannbaum verwendet werden.)

Kreiseigenschaft Für jeden Kreis K im Graphen gilt: Die schwerste Kante auf dem Kreis ist nicht Teil des minimalen Spannbaums.

2.1.1 Kruskal

```

1 typedef pair<int,int> ii;
2 typedef vector<pair<int,ii>> graph;
3
4 //Takes a Graph g (EdgeList!!!) with N nodes and computes the MST and Cost of it. Runtime:  $O(|E| \cdot \log(|E|))$ 
5 //Requires UnionFind-Datastructure!!!
6 pair<graph,int> buildMST(int N, graph& g) {
7     UnionFind uf(N);
8     graph mst; int mst_cost = 0; int M = g.size();
9     sort(g.begin(),g.end());
10    for(int i = 0; i < M; i++) {
11        int u = g[i].second.first, v = g[i].second.second;
12        if(uf.findSet(u) != uf.findSet(v)) {
13            mst.push_back(g[i]); mst_cost += g[i].first;
14            uf.unionSets(u,v);
15        }
16    }
17    return make_pair(mst,mst_cost);
18 }
```

2.2 Kürzeste Wege

2.2.1 Algorithmus von DIJKSTRA

Kürzeste Pfade in Graphen ohne negative Kanten.

```

1 // Laufzeit:  $O((|E|+|V|) \cdot \log |V|)$ 
2 void dijkstra(int start) {
3     priority_queue<ii, vector<ii>, greater<ii> > pq;
4     vector<int> dist, parent;
5     dist.assign(NUM_VERTICES, INF);
6     parent.assign(NUM_VERTICES, -1);
7
8     dist[start] = 0;
9     pq.push(ii(0, start));
10
11    while (!pq.empty()) {
12        ii front = pq.top(); pq.pop();
13        int curNode = front.second, curDist = front.first;
14
15        if (curDist > dist[curNode]) continue;
16
17        for (int i = 0; i < (int)adjlist[curNode].size(); i++) {
18            int nextNode = adjlist[curNode][i].first, nextDist = curDist + adjlist[curNode][i].second;
19
20            if (nextDist < dist[nextNode]) {
21                dist[nextNode] = nextDist; parent[nextNode] = curNode;
22                pq.push(ii(nextDist, nextNode));
23            }
24        }
25    }
```

26 }

2.2.2 BELLMANN-FORD-Algorithmus

Kürzestes Pfade in Graphen mit negativen Kanten. Erkennt negative Zyklen.

```

1 // Laufzeit:  $O(|V|*|E|)$ 
2 struct edge {
3     int from; int to; int cost;
4     edge () {};
5     edge (int from, int to, int cost) : from(from), to(to), cost(cost) {};
6 };
7
8 vector<edge> edges; // Kanten einfügen!
9 vector<int> dist, parent;
10
11 void bellmannFord() {
12     dist.assign(NUM_VERTICES, INF); dist[0] = 0;
13     parent.assign(NUM_VERTICES, -1);
14     for (int i = 0; i < NUM_VERTICES - 1; i++) {
15         for (int j = 0; j < (int)edges.size(); j++) {
16             if (dist[edges[j].from] + edges[j].cost < dist[edges[j].to]) {
17                 dist[edges[j].to] = dist[edges[j].from] + edges[j].cost;
18                 parent[edges[j].to] = edges[j].from;
19             }
20         }
21     }
22
23     // "dist" und "parent" sind korrekte kürzeste Pfade.
24     // Folgende Zeilen prüfen nur negative Kreise.
25     for (int j = 0; j < (int)edges.size(); j++) {
26         if (dist[edges[j].from] + edges[j].cost < dist[edges[j].to]) {
27             // Negativer Kreis gefunden.
28         }
29     }
30 }

```

2.2.3 FLOYD-WARSHALL-Algorithmus

```

1 // Laufzeit:  $O(|V|^3)$ 
2 // Initialisiere mat: mat[i][i] = 0, mat[i][j] = INF falls i & j nicht adjazent, Länge sonst.
3 void floydWarshall() {
4     for (k = 0; k < MAX_V; k++) {
5         for (i = 0; i < MAX_V; i++) {
6             for (j = 0; j < MAX_V; j++) {
7                 if (mat[i][k] != INF && mat[k][j] != INF && mat[i][k] + mat[k][j] < mat[i][j]) {
8                     mat[i][j] = mat[i][k] + mat[k][j];
9                 }
10            }
11        }
12    }
13 }

```

- Evtl. überschreibt die Eingabe die Nullen auf der Hauptdiagonalen. Nur negative Werte sollten die Nullen überschreiben.
- Von parallelen Kanten sollte nur die günstigste gespeichert werden.
- Knoten i liegt genau dann auf einem negativen Kreis, wenn $\text{dist}[i][i] < 0$ ist.
- Ein u - v -Pfad existiert nicht, wenn $\text{dist}[u][v] == \text{INF}$.
- Gibt es einen Knoten c , sodass $\text{dist}[u][c] != \text{INF} \ \&\& \ \text{dist}[c][v] != \text{INF} \ \&\& \ \text{dist}[c][c] < 0$, wird der u - v -Pfad beliebig kurz.

2.3 Strongly Connected Components (TARJANS-Algorithmus)

```

1 // Laufzeit:  $O(|V|+|E|)$ 
2 int counter, sccCounter;
3 vector<bool> visited, inStack;
4 vector< vector<int> > adjlist;
5 vector<int> d, low, sccs;
6 stack<int> s;
7
8 void visit(int v) {
9     visited[v] = true;
10    d[v] = counter; low[v] = counter; counter++;
11    inStack[v] = true; s.push(v);
12
13    for (int i = 0; i < (int)adjlist[v].size(); i++) {
14        int u = adjlist[v][i];
15        if (!visited[u]) {
16            visit(u);
17            low[v] = min(low[v], low[u]);
18        } else if (inStack[u]) {
19            low[v] = min(low[v], low[u]);
20        }
21    }
22
23    if (d[v] == low[v]) {
24        int u;
25        do {
26            u = s.top(); s.pop(); inStack[u] = false;
27            sccs[u] = sccCounter;
28        } while(u != v);
29        sccCounter++;
30    }
31 }
32
33 void scc() {
34     // Initialisiere adjlist!
35     visited.clear(); visited.assign(NUM_VERTICES, false);
36     d.clear(); d.resize(NUM_VERTICES);
37     low.clear(); low.resize(NUM_VERTICES);
38     inStack.clear(); inStack.assign(NUM_VERTICES, false);
39     sccs.clear(); sccs.resize(NUM_VERTICES);
40
41     counter = 0;
42     sccCounter = 0;
43     for (i = 0; i < NUM_VERTICES; i++) {
44         if (!visited[i]) {
45             visit(i);
46         }
47     }
48     // sccCounter ist Anzahl der starkem Zusammenhangskomponenten.
49     // sccs enthält den Index der SCC für jeden Knoten.
50 }

```

2.4 Artikulationspunkte und Brücken

```

1 vector< vector<int> > adjlist;
2 vector<int> low;
3 vector<int> d;
4 vector<bool> isArtPoint;
5 vector< vector<int> > bridges; //nur fuer Bruecken
6 int counter = 0;
7
8 void visit(int v, int parent) {
9     d[v] = low[v] = ++counter;
10    int numVisits = 0, maxlow = 0;
11
12    for (vector<int>::iterator vit = adjlist[v].begin(); vit != adjlist[v].end(); vit++) {
13        if (d[*vit] == 0) {
14            numVisits++;
15            visit(*vit, v);

```

```

16     if (low[*vit] > maxlow) {
17         maxlow = low[*vit];
18     }
19
20     if (low[*vit] > d[v]) { //nur fuer Bruecken, evtl. parent betrachten!
21         bridges[v].push_back(*vit);
22         bridges[*vit].push_back(v);
23     }
24
25     low[v] = min(low[v], low[*vit]);
26 } else {
27     if (d[*vit] < low[v]) {
28         low[v] = d[*vit];
29     }
30 }
31 }
32
33 if (parent == -1) {
34     if (numVisits > 1) isArtPoint[v] = true;
35 } else {
36     if (maxlow >= d[v]) isArtPoint[v] = true;
37 }
38 }
39
40 void findArticulationPoints() {
41     low.clear(); low.resize(adjlist.size());
42     d.clear(); d.assign(adjlist.size(), 0);
43     isArtPoint.clear(); isArtPoint.assign(adjlist.size(), false);
44     bridges.clear(); isBridge.resize(adjlist.size()); //nur fuer Bruecken
45     for (int v = 0; v < (int)adjlist.size(); v++) {
46         if (d[v] == 0) visit(v, -1);
47     }
48 }

```

2.5 Eulertouren

- Zyklus existiert, wenn jeder Knoten geraden Grad hat (ungerichtet), bzw. bei jedem Knoten Ein- und Ausgangsgrad übereinstimmen (gerichtet).
- Pfad existiert, wenn alle bis auf (maximal) zwei Knoten geraden Grad haben (ungerichtet), bzw. bei allen Knoten bis auf zwei Ein- und Ausgangsgrad übereinstimmen, wobei einer eine Ausgangskante mehr hat (Startknoten) und einer eine Eingangskante mehr hat (Endknoten).
- **Je nach Aufgabenstellung überprüfen, wie isolierte Punkte interpretiert werden sollen.**
- Der Code unten läuft in Linearzeit. Wenn das nicht notwendig ist (oder bestimmte Sortierungen verlangt werden), gehts mit einem set einfacher.

```

1 VISIT(v):
2     forall e=(v,w) in E
3         delete e from E
4         VISIT(w)
5     print e

```

Abbildung 1: Idee für Eulerzyklen

```

1 // Laufzeit: O(|V|+|E|)
2 vector< vector<int> > adjlist;
3 vector< vector<int> > otherIdx;
4 vector<int> cycle;
5 vector<int> validIdx;
6
7 void swapEdges(int n, int a, int b) { // Vertauscht Kanten mit Indizes a und b von Knoten n.
8     int neighA = adjlist[n][a];
9     int neighB = adjlist[n][b];
10    int idxNeighA = otherIdx[n][a];
11    int idxNeighB = otherIdx[n][b];
12    swap(adjlist[n][a], adjlist[n][b]);

```

```

13 swap(otherIdx[n][a], otherIdx[n][b]);
14 otherIdx[neighA][idxNeighA] = b;
15 otherIdx[neighB][idxNeighB] = a;
16 }
17
18 void removeEdge(int n, int i) { // Entfernt Kante i von Knoten n (und die zugehörige Rückwärtskante).
19     int other = adjlist[n][i];
20     if (other == n) { //Schlingen.
21         validIdx[n]++;
22         return;
23     }
24     int otherIndex = otherIdx[n][i];
25     validIdx[n]++;
26     if (otherIndex != validIdx[other]) {
27         swapEdges(other, otherIndex, validIdx[other]);
28     }
29     validIdx[other]++;
30 }
31
32 // Findet Eulerzyklus an Knoten n startend.
33 // Teste vorher, dass Graph zusammenhängend ist! Was ist mit isolierten Knoten?
34 // Teste vorher, ob Eulerzyklus überhaupt existiert!
35 void euler(int n) {
36     while (validIdx[n] < (int)adjlist[n].size()) {
37         int nn = adjlist[n][validIdx[n]];
38         removeEdge(n, validIdx[n]);
39         euler(nn);
40     }
41     cycle.push_back(n); // Zyklus am Ende in cycle (umgekehrte Reihenfolge).
42 }

```

Achtung:

- Die Ausgabe erfolgt in falscher Reihenfolge.
- Algorithmus schlägt nicht fehl, falls kein Eulerzyklus existiert. Die Existenz muss separat geprüft werden.

2.6 Lowest Common Ancestor

```

1 //RMQ muss hinzugefuegt werden!
2 vector<int> visited(2*MAX_N), first(MAX_N, 2*MAX_N), depth(2*MAX_N);
3 vector<vector<int>> graph(MAX_N);
4
5 //Runtime: O(n)
6 void initLCA(int gi, int d, int &c) {
7     visited[c] = gi, depth[c] = d, first[gi] = min(c, first[gi]), c++;
8     for(int gn : graph[gi]) {
9         initLCA(gn, d+1, c);
10         visited[c] = gi, depth[c] = d, c++;
11     }
12 }
13 //[[a, b]
14 //Runtime: O(1)
15 int getLCA(int a, int b) {
16     return visited[queryRMQ(min(first[a], first[b]), max(first[a], first[b]))];
17 }
18 //=> int c = 0; initLCA(0,0,c); initRMQ(); done! [rmq on depth]

```

2.7 Max-Flow**2.7.1 Capacity Scaling**

Gut bei dünn besetzten Graphen.

```

1 // Ford Fulkerson mit Capacity Scaling.
2 // Laufzeit: O(|E|^2*log(C))
3 struct MaxFlow { // Muss mit new erstellt werden!
4     static const int MAX_N = 500; // #Knoten, kein Einfluss auf die Laufzeit.

```



```

5  struct edge { int dest, rev; ll capacity, flow; };
6  vector<edge> adjlist[MAX_N];
7  int visited[MAX_N] = {0}, target, dfsCounter = 0;
8  ll capacity;
9
10 bool dfs(int x) {
11     if (x == target) return 1;
12     if (visited[x] == dfsCounter) return 0;
13     visited[x] = dfsCounter;
14     for (edge &e : adjlist[x]) {
15         if (e.capacity >= capacity && dfs(e.dest)) {
16             e.capacity -= capacity; adjlist[e.dest][e.rev].capacity += capacity;
17             e.flow += capacity; adjlist[e.dest][e.rev].flow -= capacity;
18             return 1;
19         }
20     }
21     return 0;
22 }
23
24 void addEdge(int u, int v, ll c) {
25     adjlist[u].push_back(edge {v, (int)adjlist[v].size(), c, 0});
26     adjlist[v].push_back(edge {u, (int)adjlist[u].size() - 1, 0, 0});
27 }
28
29 ll maxFlow(int s, int t) {
30     capacity = 1L << 62;
31     target = t;
32     ll flow = 0L;
33     while (capacity) {
34         while (dfsCounter++, dfs(s)) {
35             flow += capacity;
36         }
37         capacity /= 2;
38     }
39     return flow;
40 }
41 };

```

2.7.2 Push Relabel

Gut bei sehr dicht besetzten Graphen.

```

1  // Laufzeit:  $O(V^3)$ 
2  struct PushRelabel {
3      ll capacities[MAX_V][MAX_V], flow[MAX_V][MAX_V], excess[MAX_V];
4      int height[MAX_V], list[MAX_V - 2], seen[MAX_V], n;
5
6      PushRelabel(int n) {
7          this->n = n;
8          memset(capacities, 0L, sizeof(capacities)); memset(flow, 0L, sizeof(flow));
9          memset(excess, 0L, sizeof(excess)); memset(height, 0, sizeof(height));
10         memset(list, 0, sizeof(list)); memset(seen, 0, sizeof(seen));
11     }
12
13     inline void addEdge(int u, int v, ll c) { capacities[u][v] += c; }
14
15     void push(int u, int v) {
16         ll send = min(excess[u], capacities[u][v] - flow[u][v]);
17         flow[u][v] += send; flow[v][u] -= send;
18         excess[u] -= send; excess[v] += send;
19     }
20
21     void relabel(int u) {
22         int minHeight = INT_MAX / 2;
23         for (int v = 0; v < n; v++) {
24             if (capacities[u][v] - flow[u][v] > 0) {
25                 minHeight = min(minHeight, height[v]);
26                 height[u] = minHeight + 1;
27             }
28         }
29     }
30 };

```

```

29 void discharge(int u) {
30     while (excess[u] > 0) {
31         if (seen[u] < n) {
32             int v = seen[u];
33             if (capacities[u][v] - flow[u][v] > 0 && height[u] > height[v]) push(u, v);
34             else seen[u]++;
35         } else {
36             relabel(u);
37             seen[u] = 0;
38         }
39     }
40
41 void moveToFront(int u) {
42     int temp = list[u];
43     for (int i = u; i > 0; i--)
44         list[i] = list[i - 1];
45     list[0] = temp;
46 }
47
48 ll maxFlow(int source, int target) {
49     for (int i = 0, p = 0; i < n; i++) if (i != source && i != target) list[p++] = i;
50
51     height[source] = n;
52     excess[source] = LLONG_MAX / 2;
53     for (int i = 0; i < n; i++) push(source, i);
54
55     int p = 0;
56     while (p < n - 2) {
57         int u = list[p], oldHeight = height[u];
58         discharge(u);
59         if (height[u] > oldHeight) {
60             moveToFront(p);
61             p = 0;
62         } else p++;
63     }
64
65     ll maxflow = 0L;
66     for (int i = 0; i < n; i++) maxflow += flow[source][i];
67     return maxflow;
68 }

```

2.7.3 Anwendungen

- **Maximum Edge Disjoint Paths**

Finde die maximale Anzahl Pfade von s nach t , die keine Kante teilen.

1. Setze s als Quelle, t als Senke und die Kapazität jeder Kante auf 1.
2. Der maximale Fluss entspricht der unterschiedlichen Pfade ohne gemeinsame Kanten.

- **Maximum Independent Paths**

Finde die maximale Anzahl Pfade von s nach t , die keinen Knoten teilen.

1. Setze s als Quelle, t als Senke und die Kapazität jeder Kante *und jedes Knotens* auf 1.
2. Der maximale Fluss entspricht der unterschiedlichen Pfade ohne gemeinsame Knoten.

- **Min-Cut**

Der maximale Fluss ist gleich dem minimalen Schnitt. Bei Quelle s und Senke t , partitioniere in S und T . Zu S gehören alle Knoten, die im Residualgraphen von s aus erreichbar sind (Rückwärtskanten beachten).

2.8 Min-Cost-Max-Flow

```

1 typedef long long ll;
2 static const ll flowlimit = 1LL << 60; // Should be bigger than the max flow.
3 struct MinCostFlow { // Should be initialized with new.
4     static const int maxn = 400; // Should be bigger than the #vertices.
5     static const int maxm = 5000; // #edges.
6     struct edge { int node; int next; ll flow; ll value; } edges[maxm << 1];
7     int graph[maxn], queue[maxn], pre[maxn], con[maxn], n, m, source, target, top;
8     bool inqueue[maxn];

```

```

9  ll maxflow, mincost, dis[maxn];
10
11  MinCostFlow() { memset(graph, -1, sizeof(graph)); top = 0; }
12
13  inline int inverse(int x) { return 1 + ((x >> 1) << 2) - x; }
14
15  // Directed edge from u to v, capacity c, weight w.
16  inline int addedge(int u, int v, int c, int w) {
17      edges[top].value = w; edges[top].flow = c; edges[top].node = v;
18      edges[top].next = graph[u]; graph[u] = top++;
19      edges[top].value = -w; edges[top].flow = 0; edges[top].node = u;
20      edges[top].next = graph[v]; graph[v] = top++;
21      return top - 2;
22  }
23
24  bool SPFA() {
25      int point, node, now, head = 0, tail = 1;
26      memset(pre, -1, sizeof(pre));
27      memset(inqueue, 0, sizeof(inqueue));
28      memset(dis, 0x7F, sizeof(dis));
29      dis[source] = 0; queue[0] = source;
30      pre[source] = source; inqueue[source] = true;
31
32      while (head != tail) {
33          now = queue[head++];
34          point = graph[now];
35          inqueue[now] = false;
36          head %= maxn;
37
38          while (point != -1) {
39              node = edges[point].node;
40              if (edges[point].flow > 0 && dis[node] > dis[now] + edges[point].value) {
41                  dis[node] = dis[now] + edges[point].value;
42                  pre[node] = now; con[node] = point;
43                  if (!inqueue[node]) {
44                      inqueue[node] = true; queue[tail++] = node;
45                      tail %= maxn;
46                  }
47              }
48              point = edges[point].next;
49          }
50      }
51      return pre[target] != -1;
52  }
53
54  void extend() {
55      ll w = flowlimit;
56      for (int u = target; pre[u] != u; u = pre[u]) {
57          w = min(w, edges[con[u]].flow);
58      }
59      maxflow += w;
60      mincost += dis[target] * w;
61      for (int u = target; pre[u] != u; u = pre[u]) {
62          edges[con[u]].flow -= w;
63          edges[inverse(con[u])].flow += w;
64      }
65  }
66
67  void mincostflow() {
68      maxflow = 0;
69      mincost = 0;
70      while (SPFA()) {
71          extend();
72      }
73  }
74  };

```

2.9 Maximal Cardinality Bipartite Matching

```

1 // Laufzeit:  $O(n*(|V|+|E|))$ 
2 vector< vector<int> > adjlist; // Gerichtete Kanten, von links nach rechts.
3 vector<int> pairs; // Zu jedem Knoten der gematchte Knoten rechts, oder -1.
4 vector<bool> visited;
5
6 bool dfs(int i) {
7     if (visited[i]) return false;
8     visited[i] = true;
9     for (vector<int>::iterator vit = adjlist[i].begin(); vit != adjlist[i].end(); vit++) {
10         if (pairs[*vit] < 0 || dfs(pairs[*vit])) {
11             pairs[*vit] = i; pairs[i] = *vit; return true;
12         }
13     }
14     return false;
15 }
16
17 // n = #Knoten links (0..n-1), m = #Knoten rechts
18 int kuhn(int n, int m) {
19     pairs.assign(n + m, -1);
20     int ans = 0;
21     for (int i = 0; i < n; i++) {
22         visited.assign(n + m, false);
23         ans += dfs(i);
24     }
25     return ans; // Größe des Matchings.
26 }

```

2.10 TSP

```

1 // Laufzeit:  $O(n^2 \cdot n)$ 
2 // nodes[0] ist Start- und Endknoten.
3 vector<vector<int>>> dist;
4 vector<int> TSP() {
5     int n = dist.size(), m = 1 << n;
6     vector<vector<ii>>> dp(n, vector<ii>(m, ii(MAX_N, -1)));
7
8     for(int c = 0; c < n; c++) dp[c][m-1].first = dist[c][0], dp[c][m-1].second = 0;
9
10    for(int v = m - 2; v >= 0; v--) {
11        for(int c = n - 1; c >= 0; c--) {
12            for(int g = 0; g < n; g++) {
13                if(g != c && (((1 << g) & v) == 0)) {
14                    if((dp[g][(v | (1 << g))].first + dist[c][g]) < dp[c][v].first) {
15                        dp[c][v].first = dp[g][(v | (1 << g))].first + dist[c][g];
16                        dp[c][v].second = g;
17                    }
18                }
19            }
20        }
21    }
22
23    vector<int> res; res.push_back(0); int v = 0;
24    while(res.back() != 0 || res.size() == 1) {
25        res.push_back(dp[res.back()][(v | (1 << res.back()))].second);
26    }
27
28    return res;
29 }

```

2.11 Bitonic TSP

```

1 // Laufzeit:  $O(|V|^2)$ 
2 vector< vector<double> > dp; // Initialisiere mit -1
3 vector< vector<double> > dist; // Initialisiere mit Entfernungen zwischen Punkten.
4 vector<int> lr, rl; // Links-nach-rechts und rechts-nach-links Pfade.
5 int n; // #Knoten

```

```

6
7 // get(0, 0) gibt die Länge der kürzesten bitonischen Route.
8 double get(int p1, int p2) {
9     int v = max(p1, p2) + 1;
10    if (v == n - 1) return dist[p1][v] + dist[v][p2];
11    if (dp[p1][p2] > -0.5) return dp[p1][p2];
12    double tryLR = dist[p1][v] + get(v, p2), tryRl = dist[v][p2] + get(p1, v);
13    if (tryLR < tryRl) lr.push_back(v); // Baut die Pfade auf. Fügt v zu rl hinzu, falls beide gleich teuer.
14    else rl.push_back(v); // Ändert das, falls nötig.
15    return min(tryLR, tryRl);
16 }

```

3 Geometrie

3.1 Closest Pair

```

1 double squaredDist(point a, point b) {
2     return (a.first-b.first) * (a.first-b.first) + (a.second-b.second) * (a.second-b.second);
3 }
4
5 bool compY(point a, point b) {
6     if (a.second == b.second) return a.first < b.first;
7     return a.second < b.second;
8 }
9
10 double shortestDist(vector<point> &points) {
11     //check that points.size() > 1 and that ALL POINTS ARE DIFFERENT
12     set<point, bool(*) (point, point)> status(compY);
13     sort(points.begin(), points.end());
14     double opt = 1e30, sqrtOpt = 1e15;
15     auto left = points.begin(), right = points.begin();
16     status.insert(*right); right++;
17
18     while (right != points.end()) {
19         if (fabs(left->first - right->first) >= sqrtOpt) {
20             status.erase(*(left++));
21         } else {
22             auto lower = status.lower_bound(point(-1e20, right->second - sqrtOpt));
23             auto upper = status.upper_bound(point(-1e20, right->second + sqrtOpt));
24             while (lower != upper) {
25                 double cand = squaredDist(*right, *lower);
26                 if (cand < opt) {
27                     opt = cand;
28                     sqrtOpt = sqrt(opt);
29                 }
30                 ++lower;
31             }
32             status.insert(*(right++));
33         }
34     }
35     return sqrtOpt;
36 }

```

3.2 Geraden

```

1 struct pt { //complex<double> does not work here, because we need to set pt.x and pt.y
2     double x, y;
3     pt() {}
4     pt(double x, double y) : x(x), y(y) {}
5 };
6
7 struct line {
8     double a, b, c; //a*x+b*y+c, b=0 ==> vertical line, b=1 ==> otherwise
9 };
10

```

```

11 line pointsToLine(pt p1, pt p2) {
12     line l;
13     if (fabs(p1.x - p2.x) < EPSILON) {
14         l.a = 1; l.b = 0.0; l.c = -p1.x;
15     } else {
16         l.a = -(double)(p1.y - p2.y) / (p1.x - p2.x);
17         l.b = 1.0;
18         l.c = -(double)(l.a * p1.x) - p1.y;
19     }
20     return l;
21 }
22
23 bool areParallel(line l1, line l2) {
24     return (fabs(l1.a - l2.a) < EPSILON) && (fabs(l1.b - l2.b) < EPSILON);
25 }
26
27 bool areSame(line l1, line l2) {
28     return areParallel(l1, l2) && (fabs(l1.c - l2.c) < EPSILON);
29 }
30
31 bool areIntersect(line l1, line l2, pt &p) {
32     if (areParallel(l1, l2)) return false;
33     p.x = (l2.b * l1.c - l1.b * l2.c) / (l2.a * l1.b - l1.a * l2.b);
34     if (fabs(l1.b) > EPSILON) p.y = -(l1.a * p.x + l1.c);
35     else p.y = -(l2.a * p.x + l2.c);
36     return true;
37 }

```

3.3 Konvexe Hülle

```

1 // Laufzeit: O(n*log(n))
2 typedef pair<ll, ll> pt;
3
4 // >0 => PAB dreht gegen den Uhrzeigersinn.
5 // <0 => PAB dreht im Uhrzeigersinn.
6 // =0 => PAB sind kollinear.
7 ll cross(const pt p, const pt a, const pt b) {
8     return (a.first - p.first) * (b.second - p.second) -
9         (a.second - p.second) * (b.first - p.first);
10 }
11
12 // Punkte auf der konvexen Hülle, gegen den Uhrzeigersinn sortiert.
13 // Kollineare Punkte sind nicht enthalten. Entferne "=" im CCW-Test um sie aufzunehmen.
14 // Achtung: Der erste und letzte Punkt im Ergebnis sind gleich.
15 // Achtung: Alle Punkte müssen verschieden sein.
16 vector<pt> convexHull(vector<pt> p){
17     int n = p.size(), k = 0;
18     vector<pt> h(2 * n);
19     sort(p.begin(), p.end());
20     // Untere Hülle.
21     for (int i = 0; i < n; i++) {
22         while (k >= 2 && cross(h[k - 2], h[k - 1], p[i]) <= 0.0) k--;
23         h[k++] = p[i];
24     }
25     // Obere Hülle.
26     for (int i = n - 2, t = k + 1; i >= 0; i--) {
27         while (k >= t && cross(h[k - 2], h[k - 1], p[i]) <= 0.0) k--;
28         h[k++] = p[i];
29     }
30     h.resize(k);
31     return h;
32 }

```

3.4 Formeln - std::complex

```

1 // Komplexe Zahlen als Darstellung für Punkte.

```

```

2 // Wenn immer möglich complex<int> verwenden. Achtung: Funktionen wie abs() geben dann int zurück.
3 typedef pt complex<double>;
4
5 // Winkel zwischen Punkt und x-Achse in [0, 2 * PI), Winkel zwischen a und b.
6 double angle = arg (a), angle_a_b = arg (a - b);
7
8 // Punkt rotiert um Winkel theta.
9 pt a_rotated = a * exp (pt (0, theta));
10
11 // Mittelpunkt des Dreiecks abc.
12 pt centroid = (a + b + c) / 3.0;
13
14 // Skalarprodukt.
15 double dot(pt a, pt b) {
16     return real(conj(a) * b);
17 }
18
19 // Kreuzprodukt, 0, falls kollinear.
20 double cross(pt a, pt b) {
21     return imag(conj(a) * b);
22 }
23
24 // Flächeninhalt eines Dreiecks bei bekannten Eckpunkten.
25 double areaOfTriangle(pt a, pt b, pt c) {
26     return abs(cross(b - a, c - a)) / 2.0;
27 }
28
29 // Flächeninhalt eines Dreiecks bei bekannten Seitenlängen.
30 double areaOfTriangle(double a, double b, double c) {
31     double s = (a + b + c) / 2;
32     return sqrt(s * (s-a) * (s-b) * (s-c));
33 }
34
35 // Sind die Dreiecke a1, b1, c1, and a2, b2, c2 ähnlich?
36 // Erste Zeile testet Ähnlichkeit mit gleicher Orientierung,
37 // zweite Zeile testet Ähnlichkeit mit unterschiedlicher Orientierung
38 bool similar (pt a1, pt b1, pt c1, pt a2, pt b2, pt c2) {
39     return (
40         (b2 - a2) * (c1 - a1) == (b1 - a1) * (c2 - a2) ||
41         (b2 - a2) * (conj (c1) - conj (a1)) == (conj (b1) - conj (a1)) * (c2 - a2)
42     );
43 }
44
45 // -1 => gegen den Uhrzeigersinn, 0 => kollinear, 1 => im Uhrzeigersinn.
46 // Einschränken der Rückgabe auf [-1,1] ist sicherer gegen Overflows.
47 double orientation(pt a, pt b, pt c) {
48     double orien = cross(b - a, c - a);
49     if (abs(orien) < EPSILON) return 0; // Might need large EPSILON: ~1e-6
50     return orien < 0 ? -1 : 1;
51 }
52
53 // Test auf Streckenschnitt zwischen a-b und c-d.
54 bool lineSegmentIntersection(pt a, pt b, pt c, pt d) {
55     if (orientation(a, b, c) == 0 && orientation(a, b, d) == 0) { // Falls kollinear.
56         double dist = abs(a - b);
57         return (abs(a - c) <= dist && abs(b - c) <= dist) || (abs(a - d) <= dist && abs(b - d) <= dist);
58     }
59     return orientation(a, b, c) * orientation(a, b, d) <= 0 && orientation(c, d, a) * orientation(c, d, b) <= 0;
60 }
61
62 // Berechnet die Schnittpunkte der Strecken a-b und c-d.
63 // Enthält entweder keinen Punkt, den einzigen Schnittpunkt oder die Endpunkte der Schnittstrecke.
64 // Achtung: operator<, min, max müssen selbst geschrieben werden!
65 vector<pt> lineSegmentIntersection(pt a, pt b, pt c, pt d) {
66     vector<pt> result;
67     if (orientation(a, b, c) == 0 && orientation(a, b, d) == 0 &&
68         orientation(c, d, a) == 0 && orientation(c, d, b) == 0) {
69         pt minAB = min(a, b), maxAB = max(a, b), minCD = min(c, d), maxCD = max(c, d);
70         if (minAB < minCD && maxAB < minCD) return result;
71         if (minCD < minAB && maxCD < minAB) return result;
72         pt start = max(minAB, minCD), end = min(maxAB, maxCD);

```

```

73     result.push_back(start);
74     if (start != end) result.push_back(end);
75     return result;
76 }
77 double x1 = real(b) - real(a), y1 = imag(b) - imag(a);
78 double x2 = real(d) - real(c), y2 = imag(d) - imag(c);
79 double u1 = (-y1 * (real(a) - real(c)) + x1 * (imag(a) - imag(c))) / (-x2 * y1 + x1 * y2);
80 double u2 = (x2 * (imag(a) - imag(c)) - y2 * (real(a) - real(c))) / (-x2 * y1 + x1 * y2);
81 if (u1 >= 0 && u1 <= 1 && u2 >= 0 && u2 <= 1) {
82     double x = real(a) + u2 * x1, y = imag(a) + u2 * y1;
83     result.push_back(pt(x, y));
84 }
85 return result;
86 }
87
88 // Entfernung von Punkt p zur Gearden durch a-b.
89 double distToLine(pt a, pt b, pt p) {
90     return abs(cross(p - a, b - a)) / abs(b - a);
91 }
92
93 // Liegt p auf der Geraden a-b?
94 bool pointOnLine(pt a, pt b, pt p) {
95     return orientation(a, b, c) == 0;
96 }
97
98 // Liegt p auf der Strecke a-b?
99 bool pointOnLineSegment(pt a, pt b, pt p) {
100     if (orientation(a, b, p) != 0) return false;
101     return real(p) >= min(real(a), real(b)) && real(p) <= max(real(a), real(b)) &&
102         imag(p) >= min(imag(a), imag(b)) && imag(p) <= max(imag(a), imag(b));
103 }
104
105 // Entfernung von Punkt p zur Strecke a-b.
106 double distToSegment(pt a, pt b, pt p) {
107     if (a == b) return abs(p - a);
108     double segLength = abs(a - b);
109     double u = ((real(p) - real(a)) * (real(b) - real(a)) +
110         (imag(p) - imag(a)) * (imag(b) - imag(a))) /
111         (segLength * segLength);
112     pt projection(real(a) + u * (real(b) - real(a)), imag(a) + u * (imag(b) - imag(a)));
113     double projectionDist = abs(p - projection);
114     if (!pointOnLineSegment(a, b, projection)) projectionDist = 1e30;
115     return min(projectionDist, min(abs(p - a), abs(p - b)));
116 }
117
118 // Kürzeste Entfernung zwischen den Strecken a-b und c-d.
119 double distBetweenSegments(pt a, pt b, pt c, pt d) {
120     if (lineSegmentIntersection(a, b, c, d)) return 0.0;
121     double result = distToSegment(a, b, c);
122     result = min(result, distToSegment(a, b, d));
123     result = min(result, distToSegment(c, d, a));
124     return min(result, distToSegment(c, d, b));
125 }
126
127 // Liegt d in der gleichen Ebene wie a, b, und c?
128 bool isCoplanar(pt a, pt b, pt c, pt d) {
129     return abs((b - a) * (c - a) * (d - a)) < EPSILON;
130 }
131
132 // Berechnet den Flächeninhalt eines Polygons (nicht selbstschneidend).
133 double areaOfPolygon(vector<pt> &polygon) { // Jeder Eckpunkt nur einmal im Vektor.
134     double res = 0; int n = polygon.size();
135     for (int i = 0; i < n; i++)
136         res += real(polygon[i]) * imag(polygon[(i + 1) % n]) - real(polygon[(i + 1) % n]) * imag(polygon[i]);
137     return 0.5 * res; // Positiv, wenn Punkte gegen den Uhrzeigersinn gegeben sind. Sonst negativ.
138 }
139
140 // Testet, ob sich zwei Rechtecke (p1, p2) und (p3, p4) schneiden (jeweils gegenüberliegende Ecken).
141 bool rectIntersection(pt p1, pt p2, pt p3, pt p4) {
142     double minx12 = min(real(p1), real(p2)), maxx12 = max(real(p1), real(p2));
143     double minx34 = min(real(p3), real(p4)), maxx34 = max(real(p3), real(p4));

```



```

144 double miny12 = min(imag(p1), imag(p2)), maxy12 = max(imag(p1), imag(p2));
145 double miny34 = min(imag(p3), imag(p4)), maxy34 = max(imag(p3), imag(p4));
146 return (maxx12 >= minx34) && (maxx34 >= minx12) && (maxy12 >= miny34) && (maxy34 >= miny12);
147 }
148
149 // Testet, ob ein Punkt im Polygon liegt (beliebige Polygone).
150 bool pointInPolygon(pt p, vector<pt> &polygon) { // Jeder Eckpunkt nur einmal im Vektor.
151     pt rayEnd = p + pt(1, 1000000);
152     int counter = 0, n = polygon.size();
153     for (int i = 0; i < n; i++) {
154         pt start = polygon[i], end = polygon[(i + 1) % n];
155         if (lineSegmentIntersection(p, rayEnd, start, end)) counter++;
156     }
157     return counter & 1;
158 }

```

4 Mathe

4.1 ggT, kgV, erweiterter euklidischer Algorithmus

```

1 ll gcd(ll a, ll b) {
2     return b == 0 ? a : gcd(b, a % b);
3 }
4
5 ll lcm(ll a, ll b) {
6     return a * (b / gcd(a, b)); //Klammern gegen Overflow
7 }

```

```

1 //Accepted in Aufgabe mit Forderung: |X|+|Y| minimal (primaer) und X<=Y (sekundaer)
2 //hab aber keinen Beweis dafuer :)
3 ll x, y, d; //a * x + b * y = d = ggT(a,b)
4 void extendedEuclid(ll a, ll b) {
5     if (!b) {
6         x = 1; y = 0; d = a; return;
7     }
8     extendedEuclid(b, a % b);
9     ll x1 = y; ll y1 = x - (a / b) * y;
10    x = x1; y = y1;
11 }

```

4.1.1 Multiplikatives Inverses von x in $\mathbb{Z}/n\mathbb{Z}$

Sei $0 \leq x < n$. Definiere $d := \gcd(x, n)$.

Falls $d = 1$:

- Erweiterter euklidischer Algorithmus liefert α und β mit $\alpha x + \beta n = 1$
- Nach Kongruenz gilt $\alpha x + \beta n \equiv \alpha x \equiv 1 \pmod n$
- $x^{-1} \equiv \alpha \pmod n$

Falls $d \neq 1$: es existiert kein x^{-1}

```

1 ll multInv(ll n, ll p) { //berechnet das multiplikative Inverse von n in F_p
2     extendedEuclid(n, p); //implementierung von oben
3     x += ((x / p) + 1) * p;
4     return x % p;
5 }

```

4.2 Primzahlsieb von ERATOSTHENES

```

1 // Sieb des Eratosthenes. Laufzeit:  $O(n * \log \log n)$ 
2 #define N 100000001 // Bis  $10^8$  in unter 64MB Speicher.
3 bitset<N / 2> isPrime;
4
5 inline bool check(int x) { // Diese Methode zum Lookup verwenden.

```

```

6   if (x < 2) return false;
7   else if (x == 2) return true;
8   else if (!(x & 1)) return false;
9   else return !isPrime[x / 2];
10  }
11
12  inline int primeSieve(int n) { // Gibt die Anzahl der Primzahlen <= n zurück.
13      int counter = 1;
14      for (int i = 3; i <= min(N, n); i += 2) {
15          if (!isPrime[i / 2]) {
16              for (int j = 3 * i; j <= min(N, n); j += 2 * i) isPrime[j / 2] = 1;
17              counter++;
18          }
19      }
20      return counter;
21  }

```

4.3 MILLER-RABIN-Primzahltest

```

1  // Theoretisch: n < 318,665,857,834,031,151,167,461 (> 10^23)
2  // Praktisch: n <= 10^18 (long long)
3  // Laufzeit: O(log n)
4  bool isPrime(ll n) {
5      if(n == 2) return true;
6      if(n < 2 || n % 2 == 0) return false;
7      ll d=n-1,j=0;
8      while(d % 2 == 0) d >>= 1, j++;
9      for(int a = 2; a <= min((ll)37, n-1); a++) {
10         ll v = pow_mod(a, d, n);
11         if(v == 1 || v == n-1) continue;
12         for(int i = 1; i <= j; i++) {
13             v = mult_mod(v, v, n);
14             if(v == n-1 || v <= 1) break;
15         }
16         if(v != n-1) return false;
17     }
18     return true;
19 }

```

4.4 Faktorisierung

```

1  typedef pair<int,int> ii;
2  //Factorize a number n in its prime factors
3  //Call primeSieve-method before with N > sqrt(n)
4  //Return: Returns a vector of pairs, where the first entry in the pair is
5  //the prime factor p and the second counts how many times p divides n
6  vector<ii> factorize(ll n) {
7      vector<ii> fact; ll num = n, i = 0, c = 0;
8      while(num != 1) {
9          if(num % primes[i] == 0) {
10             c++; num /= primes[i];
11         } else {
12             if(c > 0)
13                 fact.push_back(make_pair(primes[i],c));
14             i++; c = 0;
15             if(primes[i]*primes[i] > num) break;
16         }
17     }
18     if(num != 1) fact.push_back(make_pair(num,c+1));
19     return fact;
20 }

```

4.5 Mod-Exponent über \mathbb{F}_p

```

1 //0<=a,b <=n and n <= MAX(11)/2
2 ll mult_mod(ll a, ll b, ll n) {
3     if(a == 0 || b == 0) return 0;
4     if(b == 1) return a % n;
5
6     if(b % 2 == 1) return (a + mult_mod(a, b-1, n)) % n;
7     else return mult_mod((a + a) % n, b / 2, n);
8 }
9
10 //0<=a,b<=n and n <= MAX(11)/2
11 ll pow_mod(ll a, ll b, ll n) {
12     if(b == 0) return 1;
13     if(b == 1) return a % n;
14
15     if(b % 2 == 1) return mult_mod(pow_mod(a, b-1, n), a, n);
16     else return pow_mod(mult_mod(a, a, n), b / 2, n);
17 }

```

4.6 LGS über \mathbb{F}_p

```

1 void normalLine(ll n, ll line, ll p) { //normalisiert Zeile line
2     ll factor = multInv(mat[line][line], p); //Implementierung von oben
3     for (ll i = 0; i <= n; i++) {
4         mat[line][i] *= factor;
5         mat[line][i] %= p;
6     }
7 }
8
9 void takeAll(ll n, ll line, ll p) { //zieht Vielfaches von line von allen anderen Zeilen ab
10     for (ll i = 0; i < n; i++) {
11         if (i == line) continue;
12         ll diff = mat[i][line]; //abziehen
13         for (ll j = 0; j <= n; j++) {
14             mat[i][j] -= (diff * mat[line][j]) % p;
15             while (mat[i][j] < 0) {
16                 mat[i][j] += p;
17             }
18         }
19     }
20 }
21
22 void gauss(ll n, ll p) { //n x n+1-Matrix, Koerper F_p
23     for (ll line = 0; line < n; line++) {
24         normalLine(n, line, p);
25         takeAll(n, line, p);
26     }
27 }

```

4.7 Binomialkoeffizienten

```

1 ll calc_binom(ll N, ll K) {
2     ll r = 1, d;
3     if (K > N) return 0;
4     for (d = 1; d <= K; d++) {
5         r *= N--;
6         r /= d;
7     }
8     return r;
9 }

```

4.8 Satz von SPRAGUE-GRUNDY

Weise jedem Zustand X wie folgt eine GRUNDY-Zahl $g(X)$ zu:

$$g(X) := \min\{\mathbb{Z}_0^+ \setminus \{g(Y) \mid Y \text{ von } X \text{ aus direkt erreichbar}\}\}$$

X ist genau dann gewonnen, wenn $g(X) > 0$ ist.

Wenn man k Spiele in den Zuständen X_1, \dots, X_k hat, dann ist die GRUNDY-Zahl des Gesamtzustandes $g(X_1) \oplus \dots \oplus g(X_k)$.

```

1 #Most important function!!!!11elf
2 bool WinNimm(vector<int> game) {
3     int result = 0;
4     for(int s: game) result ^= s;
5     return s > 0;
6 }

```

4.9 Maximales Teilfeld

```

1 //N := length of field
2 int maxStart = 1, maxLen = 0, curStart = 1, len = 0;
3 double maxValue = 0, sum = 0;
4 for (int pos = 0; pos < N; pos++) {
5     sum += values[pos];
6     len++;
7     if (sum > maxValue) { // neues Maximum
8         maxValue = sum; maxStart = curStart; maxLen = len;
9     }
10    if (sum < 0) { // alles zuruecksetzen
11        curStart = pos + 2; len = 0; sum = 0;
12    }
13 }
14 //maxSum := maximaler Wert, maxStart := Startposition, maxLen := Laenge der Sequenz

```

Obiger Code findet kein maximales Teilfeld, das über das Ende hinausgeht. Dazu:

1. finde maximales Teilfeld, das nicht übers Ende geht
2. berechne minimales Teilfeld, das nicht über den Rand geht (analog)
3. nimm Maximum aus gefundenem Maximalem und Allem\Minimalem

4.10 Polynome & FFT

Multipliziert Polynome A und B .

- $\deg(A * B) = \deg(A) + \deg(B)$
- Vektoren a und b müssen mindestens Größe $\deg(A * B) + 1$ haben. Größe muss eine Zweierpotenz sein.
- Für ganzzahlige Koeffizienten: $(\text{int})\text{round}(\text{real}(a[i]))$

```

1 // Laufzeit:  $O(n \log(n))$ .
2 typedef complex<double> cplx; // Eigene Implementierung ist noch deutlich schneller.
3 // s.size() muss eine Zweierpotenz sein!
4 vector<cplx> fft(const vector<cplx> &a, bool inverse = 0) {
5     int logn = 1, n = a.size();
6     vector<cplx> A(n);
7     while ((1 << logn) < n) logn++;
8     for (int i = 0; i < n; i++) {
9         int j = 0;
10        for (int k = 0; k < logn; k++) j = (j << 1) | ((i >> k) & 1);
11        A[j] = a[i];
12    }
13    for (int s = 2; s <= n; s <= 1) {
14        double angle = 2 * PI / s * (inverse ? -1 : 1);
15        cplx ws(cos(angle), sin(angle));
16        for (int j = 0; j < n; j += s) {
17            cplx w = 1;
18            for (int k = 0; k < s / 2; k++) {
19                cplx u = A[j + k], t = A[j + s / 2 + k];
20                A[j + k] = u + w * t;
21                A[j + s / 2 + k] = u - w * t;
22                if (inverse) A[j + k] /= 2, A[j + s / 2 + k] /= 2;
23                w *= ws;
24            }
25        }
26    }

```

```

27 return A;
28 }
29
30 // Polynome: a_0, a_1, ... & b_0, b_1, ...
31 vector<cplx> a = {0,0,0,0,1,2,3,4}, b = {0,0,0,0,2,3,0,1};
32 a = fft(a); b = fft(b);
33 for (int i = 0; i < (int)a.size(); i++) a[i] *= b[i];
34 a = fft(a,1); // a = a * b

```

4.11 Kombinatorik

4.11.1 Berühmte Zahlen

FIBONACCI-Zahlen	$f(0) = 0 \quad f(1) = 1 \quad f(n+2) = f(n+1) + f(n)$	Bem. 1, 2
CATALAN-Zahlen	$C_0 = 1 \quad C_n = \sum_{k=0}^{n-1} C_k C_{n-1-k} = \frac{1}{n+1} \binom{2n}{n} = \frac{2(2n-1)}{n+1} \cdot C_{n-1}$	Bem. 3, 4
EULER-Zahlen (I)	$\left\langle \begin{matrix} n \\ 0 \end{matrix} \right\rangle = \left\langle \begin{matrix} n \\ n-1 \end{matrix} \right\rangle = 1 \quad \left\langle \begin{matrix} n \\ k \end{matrix} \right\rangle = (k+1) \left\langle \begin{matrix} n-1 \\ k \end{matrix} \right\rangle + (n-k) \left\langle \begin{matrix} n-1 \\ k-1 \end{matrix} \right\rangle$	Bem. 5
EULER-Zahlen (II)	$\left\langle\left\langle \begin{matrix} n \\ 0 \end{matrix} \right\rangle\right\rangle = 1 \quad \left\langle\left\langle \begin{matrix} n \\ n \end{matrix} \right\rangle\right\rangle = 0 \quad \left\langle\left\langle \begin{matrix} n \\ k \end{matrix} \right\rangle\right\rangle = (k+1) \left\langle\left\langle \begin{matrix} n-1 \\ k \end{matrix} \right\rangle\right\rangle + (2n-k-1) \left\langle\left\langle \begin{matrix} n-1 \\ k-1 \end{matrix} \right\rangle\right\rangle$	Bem. 6
STIRLING-Zahlen (I)	$\begin{bmatrix} 0 \\ 0 \end{bmatrix} = 1 \quad \begin{bmatrix} n \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ n \end{bmatrix} = 0 \quad \begin{bmatrix} n \\ k \end{bmatrix} = \begin{bmatrix} n-1 \\ k-1 \end{bmatrix} + (n-1) \begin{bmatrix} n-1 \\ k \end{bmatrix}$	Bem. 7
STIRLING-Zahlen (II)	$\left\{ \begin{matrix} n \\ 1 \end{matrix} \right\} = \left\{ \begin{matrix} n \\ n \end{matrix} \right\} = 1 \quad \left\{ \begin{matrix} n \\ k \end{matrix} \right\} = k \left\{ \begin{matrix} n-1 \\ k \end{matrix} \right\} + \left\{ \begin{matrix} n-1 \\ k-1 \end{matrix} \right\}$	Bem. 8
Integer-Partitions	$f(1,1) = 1 \quad f(n,k) = 0 \text{ für } k > n \quad f(n,k) = f(n-k,k) + f(n,k-1)$	Bem. 9

Bemerkung 1 $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} f_n \\ f_{n+1} \end{pmatrix}$

Bemerkung 2 (ZECKENDORFS Theorem) Jede positive natürliche Zahl kann eindeutig als Summe einer oder mehrerer verschiedener FIBONACCI-Zahlen geschrieben werden, sodass keine zwei aufeinanderfolgenden FIBONACCI-Zahlen in der Summe vorkommen.

Lösung: Greedy, nimm immer die größte FIBONACCI-Zahl, die noch hineinpasst.

Bemerkung 3 • Die erste und dritte angegebene Formel sind relativ sicher gegen Overflows.
• Die erste Formel kann auch zur Berechnung der CATALAN-Zahlen bezüglich eines Moduls genutzt werden.

Bemerkung 4 Die CATALAN-Zahlen geben an: $C_n =$

- Anzahl der Binärbäume mit n Knoten
- Anzahl der validen Klammerausdrücke mit n Klammerpaaren
- Anzahl der korrekten Klammerungen von $n+1$ Faktoren
- Anzahl der Möglichkeiten ein konvexes Polygon mit $n+2$ Ecken in Dreiecke zu zerlegen.
- Anzahl der monotonen Pfade in einem $n \times n$ -Gitter, die nicht die Diagonale kreuzen. (zwischen gegenüberliegenden Ecken)

Bemerkung 5 (EULER-Zahlen 1. Ordnung) Die Anzahl der Permutationen von $\{1, \dots, n\}$ mit genau k Anstiegen.

Begründung: Für die n -te Zahl gibt es n mögliche Positionen zum Einfügen. Dabei wird entweder ein Anstieg in zwei gesplitted oder ein Anstieg um n ergänzt.

Bemerkung 6 (EULER-Zahlen 2. Ordnung) Die Anzahl der Permutationen von $\{1, 1, \dots, n, n\}$ mit genau k Anstiegen.

Bemerkung 7 (STIRLING-Zahlen 1. Ordnung) Die Anzahl der Permutationen von $\{1, \dots, n\}$ mit genau k Zyklen.

Begründung: Es gibt zwei Möglichkeiten für die n -te Zahl. Entweder sie bildet einen eigenen Zyklus, oder sie kann an jeder Position in jedem Zyklus einsortiert werden.

Bemerkung 8 (STIRLING-Zahlen 2. Ordnung) Die Anzahl der Möglichkeiten n Elemente in k nichtleere Teilmengen zu zerlegen.

Begründung: Es gibt k Möglichkeiten die n in eine $n-1$ -Partition einzuordnen. Dazu kommt der Fall, dass die n in ihrer eigenen Teilmenge (alleine) steht.

Bemerkung 9 Anzahl der Teilmengen von \mathbb{N} , die sich zu n aufaddieren mit maximalem Element $\leq k$.

4.11.2 Verschiedenes

Hanoi Towers (min steps)	$T_n = 2^n - 1$
#regions by n lines	$n(n+1)/2 + 1$
#bounded regions by n lines	$(n^2 - 3n + 2)/2$
#labeled rooted trees	n^{n-1}
#labeled unrooted trees	n^{n-2}

5 Strings

5.1 KNUTH-MORRIS-PRATT-Algorithmus

```

1 //Preprocessing Substring sub for KMP-Search
2 vector<int> kmp_preprocessing(string& sub) {
3     vector<int> b(sub.size() + 1);
4     b[0] = -1;
5     int i = 0, j = -1;
6     while(i < sub.size()) {
7         while(j >= 0 && sub[i] != sub[j])
8             j = b[j];
9         i++; j++;
10        b[i] = j;
11    }
12    return b;
13 }
14
15 //Searching after Substring sub in s
16 vector<int> kmp_search(string& s, string& sub) {
17     vector<int> pre = kmp_preprocessing(sub);
18     vector<int> result;
19     int i = 0, j = -1;
20     while(i < s.size()) {
21         while(j >= 0 && s[i] != sub[j])
22             j = pre[j];
23         i++; j++;
24         if(j == sub.size()) {
25             result.push_back(i-j);
26             j = pre[j];
27         }
28     }
29     return result;
30 }

```

5.2 AHO-CORASICK-Automat

```

1 // Laufzeit:  $O(n + m + z)$ ,  $n$  = Suchstringlänge,  $m$  = Summe der Patternlängen,  $z$  = #Matches
2 // Findet mehrere Patterns gleichzeitig in einem String.
3 // 1) Wurzel erstellen: vertex *automaton = new vertex();
4 // 2) Mit addString(automaton, s, idx); Patterns hinzufügen.
5 // 3) finishAutomaton aufrufen.
6 // 4) Mit automaton = go(automaton, c) in nächsten Zustand wechseln. Wenn patterns-Vektor nicht leer ist:
7 // Hier enden alle anhaltenen Patterns.
8 struct vertex {
9     vertex *next[ALPHABET_SIZE], *failure;
10    char character;
11    vector<int> patterns; // Indizes der Patterns, die hier enden.
12    vertex() { for (int i = 0; i < ALPHABET_SIZE; i++) next[i] = NULL; }
13 };
14
15 void addString(vertex *v, string &pattern, int patternIdx) {
16     for (int i = 0; i < (int)pattern.length(); i++) {
17         if (!v->next[(int)pattern[i]]) {
18             vertex *w = new vertex();
19             w->character = pattern[i];
20             v->next[(int)pattern[i]] = w;

```

```

21     }
22     v = v->next[(int)pattern[i]];
23 }
24 v->patterns.push_back(patternIdx);
25 }
26
27 void finishAutomaton(vertex *v) {
28     for (int i = 0; i < ALPHABET_SIZE; i++)
29         if (!v->next[i]) v->next[i] = v;
30
31     queue<vertex*> q;
32     for (int i = 0; i < ALPHABET_SIZE; i++) {
33         if (v->next[i] != v) {
34             v->next[i]->failure = v;
35             q.push(v->next[i]);
36         }
37     }
38     while (!q.empty()) {
39         vertex *r = q.front(); q.pop();
40         for (int i = 0; i < ALPHABET_SIZE; i++) {
41             if (r->next[i]) {
42                 q.push(r->next[i]);
43                 vertex *f = r->failure;
44                 while (!f->next[i]) f = f->failure;
45                 r->next[i]->failure = f->next[i];
46                 for (int j = 0; j < (int)f->next[i]->patterns.size(); j++) {
47                     r->next[i]->patterns.push_back(f->next[i]->patterns[j]);
48                 }
49             }
50         }
51     }
52 }
53
54 vertex* go(vertex *v, char c) {
55     if (v->next[(int)c]) return v->next[(int)c];
56     else return go(v->failure, c);
57 }

```

5.3 LEVENSHTAIN-Distanz

```

1 int levenshtein(string& s1, string& s2) {
2     int len1 = s1.size(), len2 = s2.size();
3     vector<int> col(len2 + 1), prevCol(len2 + 1);
4     for (int i = 0; i < len2 + 1; i++) prevCol[i] = i;
5     for (int i = 0; i < len1; i++) {
6         col[0] = i + 1;
7         for (int j = 0; j < len2; j++)
8             col[j+1] = min(min(prevCol[j+1] + 1, col[j] + 1), prevCol[j] + (s1[i]==s2[j] ? 0 : 1));
9         col.swap(prevCol);
10    }
11    return prevCol[len2];
12 }

```

5.4 Trie

```

1 //nur fuer Kleinbuchstaben!
2 struct node {
3     node *(e)[26];
4     int c = 0; //anzahl der woerter die an dem node enden.
5     node() { for(int i = 0; i < 26; i++) e[i] = NULL; }
6 };
7
8 void insert(node *root, string *txt, int s) {
9     if(s >= txt->length()) root->c++;
10    else {
11        int idx = (int)((*txt).at(s) - 'a');
12        if(root->e[idx] == NULL) {
13            root->e[idx] = new node();
14        }
15        insert(root->e[idx], txt, s+1);
16    }

```

```

17 }
18
19 int contains(node *root, string *txt, int s) {
20     if(s >= txt->length()) return root->c;
21     int idx = (int)((*txt).at(s) - 'a');
22     if(root->e[idx] != NULL) {
23         return contains(root->e[idx], txt, s+1);
24     } else return 0;
25 }

```

5.5 Suffix-Array

```

1 //longest common substring in one string (overlapping not excluded)
2 //contains suffix array:-----
3 int cmp(string &s,vector<vector<int>> &v, int i, int vi, int u, int l) {
4     int vi2 = (vi + 1) % 2, u2 = u + i / 2, l2 = l + i / 2;
5     if(i == 1) return s[u] - s[l];
6     else if (v[vi2][u] != v[vi2][l]) return (v[vi2][u] - v[vi2][l]);
7     else { //beide groesser tift nicht mehr ein, da ansonsten vorher schon unterschied in Laenge
8         if(u2 >= s.length()) return -1;
9         else if(l2 >= s.length()) return 1;
10        else return v[vi2][u2] - v[vi2][l2];
11    }
12 }
13
14 string lcsb(string s) {
15     if(s.length() == 0) return "";
16     vector<int> a(s.length());
17     vector<vector<int>> v(2, vector<int>(s.length(), 0));
18     int vi = 0;
19     for(int k = 0; k < a.size(); k++) a[k] = k;
20     for(int i = 1; i <= s.length(); i *= 2, vi = (vi + 1) % 2) {
21         sort(a.begin(), a.end(), [&] (const int &u, const int &l) {
22             return cmp(s, v, i, vi, u, l) < 0;
23         });
24         v[vi][a[0]] = 0;
25         for(int z = 1; z < a.size(); z++) v[vi][a[z]] = v[vi][a[z-1]] + (cmp(s, v, i, vi, a[z], a[z-1]) == 0 ? 0 : 1);
26     }
27 //-----
28     int r = 0, m=0, c=0;
29     for(int i = 0; i < a.size() - 1; i++) {
30         c = 0;
31         while(a[i]+c < s.length() && a[i+1]+c < s.length() && s[a[i]+c] == s[a[i+1]+c]) c++;
32         if(c > m) r=i, m=c;
33     }
34     return m == 0 ? "" : s.substr(a[r], m);
35 }

```

5.6 Longest Common Substring

```

1 //longest common substring.
2 struct lcse {
3     int i = 0, s = 0;
4 };
5 string lcp(string s[2]) {
6     if(s[0].length() == 0 || s[1].length() == 0) return "";
7     vector<lcse> a(s[0].length()+s[1].length());
8     for(int k = 0; k < a.size(); k++) a[k].i=(k < s[0].length() ? k : k - s[0].length()), a[k].s = (k < s[0].length() ? 0 : 1);
9     sort(a.begin(), a.end(), [&] (const lcse &u, const lcse &l) {
10         int ui = u.i, li = l.i;
11         while(ui < s[u.s].length() && li < s[l.s].length()) {
12             if(s[u.s][ui] < s[l.s][li]) return true;
13             else if(s[u.s][ui] > s[l.s][li]) return false;
14             ui++; li++;

```



```

15     }
16     return !(ui < s[u.s].length());
17 });
18 int r = 0, m=0, c=0;
19 for(int i = 0; i < a.size() - 1; i++) {
20     if(a[i].s == a[i+1].s) continue;
21     c = 0;
22     while(a[i].i+c < s[a[i].s].length() && a[i+1].i+c < s[a[i+1].s].length() && s[a[i].s][a[i].i+c] == s[a[i+1].s][a[i+1].i+c]) c++;
23     if(c > m) r=i, m=c;
24 }
25 return m == 0 ? "" : s[a[r].s].substr(a[r].i, m);
26 }

```

5.7 Longest Common Subsequence

```

1 string lcsm(string &a, string &b) {
2     int m[a.length() + 1][b.length() + 1], x=0, y=0;
3     memset(m, 0, sizeof(m));
4     for(int y = a.length() - 1; y >= 0; y--) {
5         for(int x = b.length() - 1; x >= 0; x--) {
6             if(a[y] == b[x]) m[y][x] = 1 + m[y+1][x+1];
7             else m[y][x] = max(m[y+1][x], m[y][x+1]);
8         }
9     } //for length only: return m[0][0];
10    string res;
11    while(x < b.length() && y < a.length()) {
12        if(a[y] == b[x]) res += a[y++], x++;
13        else if(m[y][x+1] > m[y+1][x+1]) x++;
14        else y++;
15    }
16    return res;
17 }

```

6 Java

6.1 Introduction

- Compilen: `javac main.java`
- Ausführen: `java main < sample.in`
- Eingabe:

```

1 Scanner in = new Scanner(System.in); //java.util.Scanner
2 String line = in.nextLine(); //reads the next line of the input
3 int num = in.nextInt(); //reads the next token of the input as an int
4 double num2 = in.nextDouble(); //reads the next token of the input as a double

```

- Ausgabe:

```

1 //Ausgabe in StringBuilder schreiben und am Ende alles auf einmal ausgeben -> viel schneller
2 StringBuilder sb = new StringBuilder(); //java.lang.StringBuilder
3 sb.append("Hallo Welt");
4 System.out.print(sb.toString());

```

6.2 BigInteger

Hier ein kleiner Überblick über die Methoden der Klasse `BigInteger`:

```

1 //Returns this +,*,/,- val
2 BigInteger add(BigInteger val), multiply(BigInteger val), divide(BigInteger val), subtract(BigInteger val)
3
4 //Returns this^e
5 BigInteger pow(BigInteger e)
6

```

```

7 //Bit-Operations
8 BigInteger and(BigInteger val), or(BigInteger val), xor(BigInteger val), not(), shiftLeft(int n), shiftRight(
   int n)
9
10 //Returns the greatest common divisor of abs(this) and abs(val)
11 BigInteger gcd(BigInteger val)
12
13 //Returns this mod m, this^-1 mod m, this^e mod m
14 BigInteger mod(BigInteger m), modInverse(BigInteger m), modPow(BigInteger e, BigInteger m)
15
16 //Returns the next number that is greater than this and that is probably a prime.
17 BigInteger nextProbablePrime()
18
19 //Converting BigInteger. Attention: If the BigInteger is to big the lowest bits were choosen which fits into
   the converted data-type.
20 int intValue(), long longValue(), float floatValue(), double doubleValue()

```

7 Sonstiges

7.1 2-SAT

1. Bedingungen in 2-CNF formulieren.
2. Implikationsgraph bauen, $(a \vee b)$ wird zu $\neg a \Rightarrow b$ und $\neg b \Rightarrow a$.
3. Finde die starken Zusammenhangskomponenten.
4. Genau dann lösbar, wenn keine Variable mit ihrer Negation in einer SCC liegt.

7.2 Bit Operations

```

1 //lsb: 0-th bit, msb: n-th bit
2 //get j-th bit
3 (a & (1 << j)) != 0
4 //set j-th bit
5 a |= (1 << j)
6 //clear j-th bit
7 a &= ~(1 << j)
8 //toggle j-th bit
9 a ^= (1 << j)
10 //get value of least significant bit set
11 (a & -a)
12 //turn on all bits
13 a = -1
14 //turn on first n bits (be aware of overflows)
15 a = (1 << n) - 1

```

7.3 Josephus-Problem

n Personen im Kreis, jeder k -te wird erschossen.

Spezialfall $k = 2$: Betrachte Binärdarstellung von n . Für $n = 1b_1b_2b_3..b_n$ ist $b_1b_2b_3..b_n1$ die Position des letzten Überlebenden. (Rotiere n um eine Stelle nach links)

```

1 int rotateLeft(int n) { //returns the number of the last survivor (1 based)
2     for (int i = 31; i >= 0; i--)
3         if (n & (1 << i)) {
4             n &= ~(1 << i);
5             break;
6         }
7     n <<= 1; n++; return n;
8 }

```

Allgemein: Sei $F(n, k)$ die Position des letzten Überlebenden. Nummeriere die Personen mit $0, 1, \dots, n-1$. Nach Erschießen der k -ten Person, hat der Kreis noch Größe $n-1$ und die Position des Überlebenden ist jetzt $F(n-1, k)$. Also: $F(n, k) = (F(n-1, k) + k) \% n$. Basisfall: $F(1, k) = 0$.

```

1 int josephus(int n, int k) { //returns the number of the last survivor (0 based)
2   if (n == 1) return 0;
3   return (josephus(n - 1, k) + k) % n;
4 }

```

Beachte bei der Ausgabe, dass die Personen im ersten Fall von $1, \dots, n$ nummeriert sind, im zweiten Fall von $0, \dots, n-1$!

7.4 Gemischtes

- JOHNSONS *Reweighting Algorithmus*: Füge neue Quelle s hinzu, mit Kanten mit Gewicht 0 zu allen Knoten. Nutze BELLMANN-FORD zum Betsimmen der Entfernungen $d[i]$ von s zu allen anderen Knoten. Stoppe, wenn es negative Zyklen gibt. Sonst ändere die gewichte von allen Kanten (u, v) im ursprünglichen Graphen zu $d[u] + w[u, v] - d[v]$. Dann sind alle Kantengewichte nichtnegativ, DIJKSTRA kann angewendet werden.
- Für ein System von Differenzbeschränkungen: Ändere alle Bedingungen in die Form $a - b \leq c$. Für jede Bedingung füge eine Kante (b, a) mit Gweight c ein. Füge Quelle s hinzu, mit Kanten zu allen Knoten mit Gewicht 0. Nutze BELLMANN-FORD, um die kürzesten Pfade von s aus zu finden. $d[v]$ ist mögliche Lösung für v .
- Min-Weight-Vertex-Cover im bipartiten Graph: Partitioniere in A , B und füge Kanten $s \rightarrow A$ mit Gewicht $w(A)$ und Kanten $B \rightarrow t$ mit Gewicht $w(B)$ hinzu. Füge Kanten mit Kapazität ∞ von A nach B hinzu, wo im originalen Graphen Kanten waren. Max-Flow ist die Lösung.
Im Residualgraphen:
 - Das Vertex-Cover sind die Knoten inzident zu den Brücken. *oder*
 - Die Knoten in A , die *nicht* von s erreichbar sind und die Knoten in B , die von t erreichbar sind.
- Allgemeiner Graph: Das Komplement eines Vertex-Cover ist ein Independent Set. \Rightarrow Max Weight Independent Set ist Komplement von Min Weight Vertex Cover.
- Bipartiter Graph: Min Vertex Cover (kleinste Menge Kanten, die alle Knoten berühren) = Max Matching.
- Bipartites Matching mit Gewichten auf linken Knoten. Minimiere Matchinggewicht. Lösung: Sortiere Knoten links aufsteigend nach Gewicht, danach nutze normlen Algorithmus (KUHN, Seite 11)
- Tobj, cool down!

8 Convenience-Methoden

8.1 Zeileneingabe

```

1 vector<string> split(string &s, string delim) { //zerlegt s anhand aller Zeichen in delim
2   vector<string> result; char *token;
3   token = strtok((char*)s.c_str(), (char*)delim.c_str());
4   while (token != NULL) {
5     result.push_back(string(token));
6     token = strtok(NULL, (char*)delim.c_str());
7   }
8   return result;
9 }

```

8.2 Template

```

1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 void solve() {}

```

```
6
7 int main() {
8     solve();
9     return 0;
10 }
```

8.3 Sonstiges

```
1 // Setzt das deutsche Tastaturlayout.
2 setxkbmap de
3
4 // Schnelle Ein-/Ausgabe mit cin/cout.
5 ios::sync_with_stdio(false);
6
7 // Set mit eigener Sortierfunktion. Typ muss nicht explizit angegeben werden.
8 set<point2, decltype(comp)> set1(comp);
```