

ChaosKITs  
Karlsruhe Institute of Technology

2.8	Min-Cost-Max-Flow . . . . .	7
2.9	Maximal Cardinatlity Bipartite Mat- ching . . . . .	8
2.10	TSP . . . . .	8
2.11	Bitonic TSP . . . . .	9
<b>Geometrie</b>		<b>9</b>
3.1	Closest Pair . . . . .	9
3.2	Geraden . . . . .	9
3.3	Konvexe Hülle . . . . .	10
3.4	Formeln - std::complex . . . . .	10
<b>Mathe</b>		<b>12</b>
4.1	ggT, kgV, erweiterter euklidischer Al- gorithmus . . . . .	12
4.1.1	Multiplikatives Inverses von $x$ in $\mathbb{Z}/n\mathbb{Z}$ . . . . .	12
4.2	Mod-Exponent über $\mathbb{F}_p$ . . . . .	12
4.3	LGS über $\mathbb{F}_p$ . . . . .	12
4.4	Chinesischer Restsatz . . . . .	13
4.5	Primzahlsieb von ERATOSTHENES . . . . .	13
4.6	MILLER-RABIN-Primzahltest . . . . .	13
4.7	Binomialkoeffizienten . . . . .	14
4.8	Maximales Teilfeld . . . . .	14
4.9	Polynome & FFT . . . . .	14
4.10	Kombinatorik . . . . .	15

4.10.1	Berühmte Zahlen . . . . .	15
4.10.2	Verschiedenes . . . . .	15
4.11	Satz von SPRAGUE-GRUNDY . . . . .	15
4.12	3D-Kugeln . . . . .	16
4.13	Big Integers . . . . .	16
<b>Strings</b>		<b>18</b>
5.1	KNUTH-MORRIS-PRATT-Algorithmus . .	18
5.2	AHO-CORASICK-Automat . . . . .	18
5.3	LEVENSHTEIN-Distanz . . . . .	19
5.4	Trie . . . . .	19
5.5	Suffix-Array . . . . .	19
5.6	Longest Common Substring . . . . .	20
5.7	Longest Common Subsequence . . . .	20
<b>Java</b>		<b>20</b>
6.1	Introduction . . . . .	20
6.2	BigInteger . . . . .	21
<b>Sonstiges</b>		<b>21</b>
7.1	2-SAT . . . . .	21
7.2	Zeileneingabe . . . . .	21
7.3	Bit Operations . . . . .	21
7.4	Josephus-Problem . . . . .	21
7.5	Gemischtes . . . . .	22
7.6	Sonstiges . . . . .	22

# Datenstrukturen

## 1.1 Union-Find

```

1 // Laufzeit: O(n*alpha(n))
2 // "height" ist obere Schranke für die Höhe der Bäume. Sobald
3 // Pfadkompression angewendet wurde, ist die genaue Höhe nicht mehr
4 // effizient berechenbar.
5 vector<int> parent // Initialisiere mit Index im Array.
6 vector<int> height; // Initialisiere mit 0.
7
8 int findSet(int n) { // Pfadkompression
9     if (parent[n] != n) parent[n] = findSet(parent[n]);
10    return parent[n];
11 }
12
13 void linkSets(int a, int b) { // Union by rank.
14     if (height[a] < height[b]) parent[a] = b;
15     else if (height[b] < height[a]) parent[b] = a;
16     else {
17         parent[a] = b;
18         height[b]++;
19     }
20 }
21
22 void unionSets(int a, int b) { // Diese Funktion aufrufen.
23     if (findSet(a) != findSet(b)) linkSets(findSet(a), findSet(b));
24 }

```

## 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 -INF; // 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) { m[k] = v; a[i] = v; return; }
15     int M = (X + Y) / 2;
16     update(i, v, 2 * k + 1, X, M);
17     update(i, v, 2 * k + 2, M + 1, Y);
18     m[k] = max(m[2 * k + 1], m[2 * k + 2]);
19 }
20
21 void init(int k = 0, int X = 0, int Y = MAX_N - 1) {
22     if (X == Y) { m[k] = a[X]; return; }

```

```

23     int M = (X + Y) / 2;
24     init(2 * k + 1, X, M);
25     init(2 * k + 2, M + 1, Y);
26     m[k] = max(m[2 * k + 1], m[2 * k + 2]);
27 }

```

Mit update() können ganze Intervalle geändert werden. Dazu: Offset in den inneren Knoten des Baums speichern.

## 1.3 Fenwick Tree

```

1 vector<int> FT; // Fenwick-Tree
2 int n;
3
4 // Addiert val zum Element an Index i. O(log(n)).
5 void updateFT(int i, int val) {
6     i++; while(i <= n) { FT[i] += val; i += (i & (-i)); }
7 }
8
9 // Baut Baum auf. O(n*log(n)).
10 void buildFenwickTree(vector<int>& a) {
11     n = a.size();
12     FT.assign(n+1, 0);
13     for(int i = 0; i < n; i++) updateFT(i, a[i]);
14 }
15
16 // Präfix-Summe über das Intervall [0..i]. O(log(n)).
17 int prefix_sum(int i) {
18     int sum = 0; i++;
19     while(i > 0) { sum += FT[i]; i -= (i & (-i)); }
20     return sum;
21 }

```

## 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 // Baut Struktur auf. 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] = 1;
9             else {
10                 int r = l + ss;
11                 rmq[i][l] = (data[rmq[i-1][l]] <= data[rmq[i-1][r]]) ?
12                     rmq[i-1][l] : rmq[i-1][r];
13             }
14         }
15     }
16 }
17
18 // Gibt den Index des Minimums im Intervall [l,r] zurück. O(1).
19 int queryRMQ(int l, int r) {

```

```

17 if(l >= r) return l;
18 int s = floor(log2(r-l)); r = r - (1 << s);
19 return (data[rmq[s][l]] <= data[rmq[s][r]] ? rmq[s][l] : rmq[s][r]);
20 }

```

## 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,
6     tree_order_statistics_node_update> Tree;
7
8 int main() {
9     Tree X;
10    for (int i = 1; i <= 16; i <= 1) X.insert(i); // {1, 2, 4, 8, 16}
11    cout << *X.find_by_order(3) << endl; // => 8
12    cout << X.order_of_key(10) << endl; // => 4 = min i, mit X[i] >= 10
13    return 0;
14 }

```

## 2 Graphen

### 2.1 Minimale Spannbäume

**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 // Union-Find Implementierung von oben. Laufzeit:  $O(n \log(n))$ 
2 sort(edges.begin(), edges.end());
3 vector<ii> mst; int cost = 0;
4 for (auto &e : edges) {
5     if (findSet(e.from) != findSet(e.to)) {
6         unionSets(e.from, e.to);
7         mst.push_back(ii(e.from, e.to));
8         cost += e.cost;
9     }
10 }

```

## 2.2 Kürzeste Wege

### 2.2.1 Algorithmus von DIJKSTRA

Kürzeste Pfade in Graphen ohne negative Kanten.

```

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

```

### 2.2.2 BELLMANN-FORD-Algorithmus

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

```

1 // Laufzeit:  $O(|V| \cdot |E|)$ 
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 }

```

```

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]
8                     < mat[i][j]) {
9                     mat[i][j] = mat[i][k] + mat[k][j];
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].
13        end(); vit++) {
14        if (d[*vit] == 0) {
15            numVisits++;
16            visit(*vit, v);
17            if (low[*vit] > maxlow) {
18                maxlow = low[*vit];
19            }
20
21            if (low[*vit] > d[v]) { //nur fuer Bruecken, evtl. parent
22                betrachten!
23                bridges[v].push_back(*vit);
24                bridges[*vit].push_back(v);
25            }
26
27            low[v] = min(low[v], low[*vit]);
28        } else {
29            if (d[*vit] < low[v]) {
30                low[v] = d[*vit];
31            }
32        }
33
34    if (parent == -1) {
35        if (numVisits > 1) isArtPoint[v] = true;
36    } else {
37        if (maxlow >= d[v]) isArtPoint[v] = true;
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

```

```

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

```

- 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 }
19 //=> 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
5     Laufzeit.
6     struct edge { int dest, rev; ll capacity, flow; };
7     vector<edge> adjlist[MAX_N];
8     int visited[MAX_N] = {0}, target, dfsCounter = 0;
9     ll capacity;

```

```

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 +=
17                 capacity;
18             e.flow += capacity; adjlist[e.dest][e.rev].flow -= capacity;
19             return 1;
20         }
21     }
22     return 0;
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(
9             flow));
10        memset(excess, 0L, sizeof(excess)); memset(height, 0, sizeof(height))
11        ;
12        memset(list, 0, sizeof(list)); memset(seen, 0, sizeof(seen));
13
14        inline void addEdge(int u, int v, ll c) { capacities[u][v] += c; }

```

```

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 void discharge(int u) {
31     while (excess[u] > 0) {
32         if (seen[u] < n) {
33             int v = seen[u];
34             if (capacities[u][v] - flow[u][v] > 0 && height[u] > height[v])
35                 push(u, v);
36             else seen[u]++;
37         } else {
38             relabel(u);
39             seen[u] = 0;
40         }
41     }
42
43 void moveToFront(int u) {
44     int temp = list[u];
45     for (int i = u; i > 0; i--)
46         list[i] = list[i - 1];
47     list[0] = temp;
48 }
49
50 ll maxFlow(int source, int target) {
51     for (int i = 0, p = 0; i < n; i++) if (i != source && i != target)
52         list[p++] = i;
53
54     height[source] = n;
55     excess[source] = LLONG_MAX / 2;
56     for (int i = 0; i < n; i++) push(source, i);
57
58     int p = 0;
59     while (p < n - 2) {
60         int u = list[p], oldHeight = height[u];
61         discharge(u);
62         if (height[u] > oldHeight) {
63             moveToFront(p);
64             p = 0;
65         } else p++;
66     }
67
68     ll maxflow = 0L;
69     for (int i = 0; i < n; i++) maxflow += flow[source][i];
70     return maxflow;
71 }

```

```

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].
41                 value) {
42                 dis[node] = dis[now] + edges[point].value;
43                 pre[node] = now; con[node] = point;
44                 if (!inqueue[node]) {
45                     inqueue[node] = true; queue[tail++] = node;
46                     tail %= maxn;
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 Cardinatlity Bipartite Matching

```

1 // Laufzeit: O(n*(|V|+|E|))
2 vector< vector<int> > adjlist; // Gerichtete Kanten, von links nach
3   rechts.
4 vector<int> pairs; // Zu jedem Knoten der gematchte Knoten rechts, oder
5   -1.
6 vector<bool> visited;
7
8 bool dfs(int v) {
9     if (visited[v]) return false;
10    visited[v] = true;
11    for (auto w : adjlist[v]) if (pairs[w] < 0 || dfs(pairs[w])) {
12        pairs[w] = v; pairs[v] = w; return true;
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     // Greedy Matching. Optionale Beschleunigung.
22     for (int i = 0; i < n; i++) for (auto w : adjlist[i]) if (pairs[w] ==
23         -1) {
24         pairs[i] = w; pairs[w] = i; ans++; break;
25     }
26     for (int i = 0; i < n; i++) if (pairs[i] == -1) {
27         visited.assign(n + m, false);
28         ans += dfs(i);
29     }
30     return ans; // Größe des Matchings.

```

## 2.10 TSP

```

1 // Laufzeit: O(n^2*2^n)
2 vector<vector<int>> dist; // Entfernung zwischen je zwei Punkten.
3 vector<int> TSP() {
4     int n = dist.size(), m = 1 << n;
5     vector<vector<ii>> dp(n, vector<ii>(m, ii(INF, -1)));
6
7     for(int c = 0; c < n; c++) dp[c][m-1].first = dist[c][0], dp[c][m-1].
8         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++) {

```



```

12         if(g != c && !((1 << g) & v)) {
13             if((dp[g][v | (1 << g)].first + dist[c][g]) < dp[c][v].first)
14             {
15                 dp[c][v].first = dp[g][v | (1 << g)].first + dist[c][g];
16                 dp[c][v].second = g;
17             }
18         }
19         vector<int> res; res.push_back(0); int v = 0;
20         while(res.back() != 0 || res.size() == 1) {
21             res.push_back(dp[res.back()][v | (1 << res.back())].second);
22         }
23         return res; // Enthält Knoten 0 zweimal. An erster und letzter Position

```

## 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
4 // Punkten.
5 vector<int> lr, rl; // Links-nach-rechts und rechts-nach-links Pfade.
6 int n; // #Knoten
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,
13    v);
14    if (tryLR < tryRL) lr.push_back(v); // Baut die Pfade auf. Fügt v zu rl
15    hinzu, falls beide gleich teuer.
16    else rl.push_back(v); // Ändere das, falls nötig.
17    return min(tryLR, tryRL);

```

## 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
3     .second-b.second);
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 -
23             sqrtOpt));
24             auto upper = status.upper_bound(point(-1e20, right->second +
25             sqrtOpt));
26             while (lower != upper) {
27                 double cand = squaredDist(*right, *lower);
28                 if (cand < opt) {
29                     opt = cand;
30                     sqrtOpt = sqrt(opt);
31                 }
32                 ++lower;
33             }
34             status.insert(*(right++));
35         }
36     }
37     return sqrtOpt;

```

### 3.2 Geraden

```

1 struct pt { //complex<double> does not work here, because we need to set
2     pt.x and pt.y
3     double x, y;
4     pt() {};
5     pt(double x, double y) : x(x), y(y) {};
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
14 // sie aufzunehmen.
15 // Achtung: Der erste und letzte Punkt im Ergebnis sind gleich.
16 // Achtung: Alle Punkte müssen verschieden sein.
17 vector<pt> convexHull(vector<pt> p){
18     int n = p.size(), k = 0;
19     vector<pt> h(2 * n);
20     sort(p.begin(), p.end());
21     // Untere Hülle.
22     for (int i = 0; i < n; i++) {
23         while (k >= 2 && cross(h[k - 2], h[k - 1], p[i]) <= 0.0) k--;
24         h[k++] = p[i];
25     }
26     // Obere Hülle.
27     for (int i = n - 2, t = k + 1; i >= 0; i--) {
28         while (k >= t && cross(h[k - 2], h[k - 1], p[i]) <= 0.0) k--;
29         h[k++] = p[i];
30     }
31     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
3 // () geben dann int zurück.
4 typedef pt complex<double>;
5
6 // Winkel zwischen Punkt und x-Achse in [0, 2 * PI), Winkel zwischen a
7 // und b.
8 double angle = arg (a), angle_a_b = arg (a - b);
9
10 // Punkt rotiert um Winkel theta.
11 pt a_rotated = a * exp (pt (0, theta));
12
13 // Mittelpunkt des Dreiecks abc.
14 pt centroid = (a + b + c) / 3.0;
15
16 // Skalarprodukt.
17 double dot(pt a, pt b) {
18     return real(conj(a) * b);
19 }
20
21 // Kreuzprodukt, 0, falls kollinear.
22 double cross(pt a, pt b) {
23     return imag(conj(a) * b);
24 }
25
26 // Flächeninhalt eines Dreiecks bei bekannten Eckpunkten.
27 double areaOfTriangle(pt a, pt b, pt c) {
28     return abs(cross(b - a, c - a)) / 2.0;
29 }
30
31 // Flächeninhalt eines Dreiecks bei bekannten Seitenlängen.
32 double areaOfTriangle(double a, double b, double c) {
33     double s = (a + b + c) / 2;
34     return sqrt(s * (s-a) * (s-b) * (s-c));
35 }
36
37 // Sind die Dreiecke a1, b1, c1, and a2, b2, c2 ähnlich?
38 // Erste Zeile testet Ähnlichkeit mit gleicher Orientierung,
39 // zweite Zeile testet Ähnlichkeit mit unterschiedlicher Orientierung
40 bool similar (pt a1, pt b1, pt c1, pt a2, pt b2, pt c2) {
41     return (
42         (b2 - a2) * (c1 - a1) == (b1 - a1) * (c2 - a2) ||
43         (b2 - a2) * (conj (c1) - conj (a1)) == (conj (b1) - conj (a1)) * (c2
44             - a2)
45     );
46 }
47
48 // -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
56         kollinear.
57         double dist = abs(a - b);
58         return (abs(a - c) <= dist && abs(b - c) <= dist) || (abs(a - d) <=
59             dist && abs(b - d) <= dist);
60     }
61     return orientation(a, b, c) * orientation(a, b, d) <= 0 && orientation(
62         c, d, a) * orientation(c, d, b) <= 0;
63 }
64 // Berechnet die Schnittpunkte der Strecken a-b und c-d.
65 // Enthält entweder keinen Punkt, den einzigen Schnittpunkt oder die
66 // Endpunkte der Schnittstrecke.
67 // Achtung: operator<, min, max müssen selbst geschrieben werden!
68 vector<pt> lineSegmentIntersection(pt a, pt b, pt c, pt d) {
69     vector<pt> result;
70     if (orientation(a, b, c) == 0 && orientation(a, b, d) == 0 &&
71         orientation(c, d, a) == 0 && orientation(c, d, b) == 0) {
72         pt minAB = min(a, b), maxAB = max(a, b), minCD = min(c, d), maxCD =
73             max(c, d);
74         if (minAB < minCD && maxAB < minCD) return result;
75         if (minCD < minAB && maxCD < minAB) return result;
76         pt start = max(minAB, minCD), end = min(maxAB, maxCD);
77         result.push_back(start);
78         if (start != end) result.push_back(end);
79         return result;
80     }
81     double x1 = real(b) - real(a), y1 = imag(b) - imag(a);
82     double x2 = real(d) - real(c), y2 = imag(d) - imag(c);
83     double u1 = (-y1 * (real(a) - real(c)) + x1 * (imag(a) - imag(c))) / (-
84         x2 * y1 + x1 * y2);
85     double u2 = (x2 * (imag(a) - imag(c)) - y2 * (real(a) - real(c))) / (-
86         x2 * y1 + x1 * y2);
87     if (u1 >= 0 && u1 <= 1 && u2 >= 0 && u2 <= 1) {
88         double x = real(a) + u2 * x1, y = imag(a) + u2 * y1;
89         result.push_back(pt(x, y));
90     }
91     return result;
92 }
93 // Entfernung von Punkt p zur Geraden a-b.
94 double distToLine(pt a, pt b, pt p) {
95     return abs(cross(p - a, b - a)) / abs(b - a);
96 }
97 // Liegt p auf der Geraden a-b?
98 bool pointOnLine(pt a, pt b, pt p) {
99     return orientation(a, b, p) == 0;
100 }
101 // Liegt p auf der Strecke a-b?
102 bool pointOnLineSegment(pt a, pt b, pt p) {
103     if (orientation(a, b, p) != 0) return false;
104     return real(p) >= min(real(a), real(b)) && real(p) <= max(real(a), real
105         (b)) &&
106         imag(p) >= min(imag(a), imag(b)) && imag(p) <= max(imag(a), imag(b)
107             );
108 }
109 // Entfernung von Punkt p zur Strecke a-b.
110 double distToSegment(pt a, pt b, pt p) {
111     if (a == b) return abs(p - a);
112     double segLength = abs(a - b);
113     double u = ((real(p) - real(a)) * (real(b) - real(a)) +
114         (imag(p) - imag(a)) * (imag(b) - imag(a))) /
115         (segLength * segLength);
116     pt projection(real(a) + u * (real(b) - real(a)), imag(a) + u * (imag(
117         b) - imag(a)));
118     double projectionDist = abs(p - projection);
119     if (!pointOnLineSegment(a, b, projection)) projectionDist = 1e30;
120     return min(projectionDist, min(abs(p - a), abs(p - b)));
121 }
122 // Kürzeste Entfernung zwischen den Strecken a-b und c-d.
123 double distBetweenSegments(pt a, pt b, pt c, pt d) {
124     if (lineSegmentIntersection(a, b, c, d)) return 0.0;
125     double result = distToSegment(a, b, c);
126     result = min(result, distToSegment(a, b, d));
127     result = min(result, distToSegment(c, d, a));
128     return min(result, distToSegment(c, d, b));
129 }
130 // Liegt d in der gleichen Ebene wie a, b, und c?
131 bool isCoplanar(pt a, pt b, pt c, pt d) {
132     return abs((b - a) * (c - a) * (d - a)) < EPSILON;
133 }
134 // Berechnet den Flächeninhalt eines Polygons (nicht selbstschneidend).
135 double areaOfPolygon(vector<pt> &polygon) { // Jeder Eckpunkt nur einmal
136     im Vektor.
137     double res = 0; int n = polygon.size();
138     for (int i = 0; i < n; i++)
139         res += real(polygon[i]) * imag(polygon[(i + 1) % n]) - real(polygon[(
140             i + 1) % n]) * imag(polygon[i]);
141     return 0.5 * res; // Positiv, wenn Punkte gegen den Uhrzeigersinn
142         gegeben sind. Sonst negativ.
143 }
144 // Testet, ob sich zwei Rechtecke (p1, p2) und (p3, p4) schneiden (
145 // jeweils gegenüberliegende Ecken).
146 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 // Laufzeiten: O(log(a) + log(b))
2 ll gcd(ll a, ll b) {
3     return b == 0 ? a : gcd(b, a % b);
4 }
5
6 ll lcm(ll a, ll b) {
7     return a * (b / gcd(a, b)); // Klammern gegen Overflow.
8 }

```

```

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  $\alpha$  und  $\beta$  mit  $\alpha x + \beta n = 1$ .
- Nach Kongruenz gilt  $\alpha x + \beta n \equiv \alpha x \equiv 1 \pmod{n}$ .
- $x^{-1} := \alpha \pmod{n}$

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

```

1 // Laufzeit: O(log(n) + log(p))
2 ll multInv(ll n, ll p) { // Berechnet das multiplikative Inverse von n in
    F_p.
3     extendedEuclid(n, p); // Implementierung von oben.
4     x += ((x / p) + 1) * p;
5     return x % p;
6 }

```

### 4.2 Mod-Exponent über $\mathbb{F}_p$

```

1 // Laufzeit: O(log(b))
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 // Laufzeit: O(log(b))
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 }

```

### 4.3 LGS über $\mathbb{F}_p$

```

1 // Laufzeit: O(n^3)
2 void normalLine(ll n, ll line, ll p) { // Normalisiert Zeile line.
3     ll factor = multInv(mat[line][line], p); // Implementierung von oben.
4     for (ll i = 0; i <= n; i++) {
5         mat[line][i] *= factor;
6         mat[line][i] %= p;
7     }
8 }
9 }

```

```

10 void takeAll(ll n, ll line, ll p) { // Zieht Vielfaches von line von
    allen anderen Zeilen ab.
11     for (ll i = 0; i < n; i++) {
12         if (i == line) continue;
13         ll diff = mat[i][line];
14         for (ll j = 0; j <= n; j++) {
15             mat[i][j] -= (diff * mat[line][j]) % p;
16             while (mat[i][j] < 0) {
17                 mat[i][j] += p;
18             }
19         }
20     }
21 }
22
23 void gauss(ll n, ll p) { // nx(n+1)-Matrix, Koerper F_p.
24     for (ll line = 0; line < n; line++) {
25         normalline(n, line, p);
26         takeAll(n, line, p);
27     }
28 }

```

#### 4.4 Chinesischer Restsatz

- Extrem anfällig gegen Overflows. Evtl. häufig 128-Bit Integer verwenden.
- Direkte Formel für zwei Kongruenzen  $x \equiv a \pmod n, x \equiv b \pmod m$ :

$$x \equiv a - y * n * \frac{a - b}{d} \pmod{\frac{mn}{d}} \quad \text{mit} \quad d := \text{ggT}(n, m) = yn + zm$$

Formel kann auch für nicht teilerfremde Moduli verwendet werden.

- Sind die Moduli nicht teilerfremd, existiert genau dann eine Lösung, wenn  $a_i \equiv a_j \pmod{\text{gcd}(m_i, m_j)}$ . In diesem Fall sind keine Faktoren auf der linken Seite erlaubt.

```

1 // Laufzeit: O(n * log(n)), n := Anzahl der Kongruenzen
2 // Nur für teilerfremde Moduli.
3 // Berechnet das kleinste, nicht negative x, das die Kongruenzen simultan
  löst.
4 // Alle Lösungen sind kongruent zum kgV der Moduli (Produkt, falls alle
  teilerfremd sind).
5 struct ChineseRemainder {
6     typedef __int128 lll;
7     vector<lll> lhs, rhs, module, inv;
8     lll M; // Produkt über die Moduli. Kann leicht überlaufen.
9
10    ll g(vector<lll> &vec) {
11        lll res = 0;
12        for (int i = 0; i < (int)vec.size(); i++) {
13            res += (vec[i] * inv[i]) % M;
14            res %= M;

```

```

15    }
16    return res;
17 }
18
19 // Fügt Kongruenz l * x = r (mod m) hinzu.
20 void addEquation(ll l, ll r, ll m) {
21     lhs.push_back(l);
22     rhs.push_back(r);
23     module.push_back(m);
24 }
25
26 // Löst das System.
27 ll solve() {
28     M = accumulate(module.begin(), module.end(), 1ll(1), multiplies<lll>
        >());
29     inv.resize(lhs.size());
30     for (int i = 0; i < (int)lhs.size(); i++) {
31         lll x = (M / module[i]) % module[i];
32         inv[i] = (multInvers(x, module[i]) * (M / module[i]));
33     }
34     return (multInvers(g(lhs), M) * g(rhs)) % M;
35 }
36 };

```

#### 4.5 Primzahlsieb von ERATOSTHENES

```

1 // Laufzeit: O(n * log log n)
2 #define N 1000000001 // 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 zur
    rück.
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 }

```

#### 4.6 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 }

```

## 4.7 Binomialkoeffizienten

Vorberechnen, wenn häufig benötigt.

```

1 // Laufzeit: O(k)
2 ll calc_binom(ll n, ll k) {
3     ll r = 1, d;
4     if (k > n) return 0;
5     for (d = 1; d <= k; d++) {
6         r *= n--;
7         r /= d;
8     }
9     return r;
10 }

```

## 4.8 Maximales Teilfeld

```

1 // N := Länge des Feldes.
2 // Laufzeit: O(N)
3 int maxStart = 1, maxLen = 0, curStart = 1, len = 0;
4 double maxValue = 0, sum = 0;
5 for (int pos = 0; pos < N; pos++) {
6     sum += values[pos];
7     len++;
8     if (sum > maxValue) { // Neues Maximum.
9         maxValue = sum; maxStart = curStart; maxLen = len;
10    }
11    if (sum < 0) { // Alles zurücksetzen.
12        curStart = pos + 2; len = 0; sum = 0;
13    }

```

```

14 }
15 // maxSum := maximaler Wert, maxStart := Startposition, maxLen := Länge
    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 Maximalen und Allem ohne dem Minimalen.

## 4.9 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: `(int)round(real(a[i]))`

```

1 // Laufzeit: O(n log(n)).
2 typedef complex<double> cplx; // Eigene Implementierung ist noch deutlich
    schneller.
3 vector<cplx> fft(const vector<cplx> &a, bool inverse = 0) { // a.size()
    muss eine Zweierpotenz sein!
4     int logn = 1, n = a.size();
5     vector<cplx> A(n);
6     while ((1 << logn) < n) logn++;
7     for (int i = 0; i < n; i++) {
8         int j = 0;
9         for (int k = 0; k < logn; k++) j = (j << 1) | ((i >> k) & 1);
10        A[j] = a[i];
11    }
12    for (int s = 2; s <= n; s <= 1) {
13        double angle = 2 * PI / s * (inverse ? -1 : 1);
14        cplx ws(cos(angle), sin(angle));
15        for (int j = 0; j < n; j += s) {
16            cplx w = 1;
17            for (int k = 0; k < s / 2; k++) {
18                cplx u = A[j + k], t = A[j + s / 2 + k];
19                A[j + k] = u + w * t;
20                A[j + s / 2 + k] = u - w * t;
21                if (inverse) A[j + k] /= 2, A[j + s / 2 + k] /= 2;
22                w *= ws;
23            }
24        }
25    }
26    return A;
27 }
28

```



```

29 // Polynome: a[0] = a_0, a[1] = a_1, ... und b[0] = b_0, b[1] = b_1, ...
30 // Integer-Koeffizienten: Runde beim Auslesen der Koeffizienten: (int)
   round(a[i].real())
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

```

## 4.10 Kombinatorik

### 4.10.1 Berühmte Zahlen

FIBONACCI-Zahlen	$f(0) = 0$	$f(1) = 1$	$f(n+2) = f(n+1) + f(n)$	
CATALAN-Zahlen	$C_0 = 1$	$C_n = \sum_{k=0}^{n-1} C_k C_{n-1-k}$	$= \frac{1}{n+1} \binom{2n}{n} = \frac{2(2n-1)}{n+1} \cdot C_{n-1}$	
EULER-Zahlen (I)	$\langle n \rangle_0 = \langle n \rangle_{n-1} = 1$	$\langle n \rangle_k = (k+1) \langle n-1 \rangle_k + (n-k) \langle n-1 \rangle_{k-1}$		
EULER-Zahlen (II)	$\langle\langle n \rangle\rangle_0 = 1$	$\langle\langle n \rangle\rangle_k = 0$	$\langle\langle n \rangle\rangle_k = (k+1) \langle\langle n-1 \rangle\rangle_k + (2n-k-1) \langle\langle n-1 \rangle\rangle_{k-1}$	
STIRLING-Zahlen (I)	$\begin{bmatrix} 0 \\ 0 \end{bmatrix} = 1$	$\begin{bmatrix} n \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ n \end{bmatrix} = 0$	$\begin{bmatrix} n \\ k \end{bmatrix} = \begin{bmatrix} n-1 \\ k-1 \end{bmatrix} + (n-1) \begin{bmatrix} n-1 \\ k \end{bmatrix}$	
STIRLING-Zahlen (II)	$\left\{ \begin{matrix} n \\ 1 \end{matrix} \right\} = \left\{ \begin{matrix} n \\ n \end{matrix} \right\} = 1$	$\left\{ \begin{matrix} n \\ k \end{matrix} \right\} = k \left\{ \begin{matrix} n-1 \\ k \end{matrix} \right\} + \left\{ \begin{matrix} n-1 \\ k-1 \end{matrix} \right\}$		
Integer-Partitions	$f(1,1) = 1$	$f(n,k) = 0$ für $k > n$	$f(n,k) = f(n-k,k) + f(n,k-1)$	

**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  $n$  nicht unterscheidbaren 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 (zwischen gegenüberliegenden Ecken) in einem  $n \times n$ -Gitter, die nicht die Diagonale kreuzen.

**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 gesplittet 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  $\leq k$ .

### 4.10.2 Verschiedenes

Türme von Hanoi, minimale Schrittzahl:	$T_n = 2^n - 1$
#Regionen zwischen $n$ Geraden	$n(n+1)/2 + 1$
#Abgeschlossene Regionen zwischen $n$ Geraden	$(n^2 - 3n + 2)/2$
#Markierte, gewurzelte Bäume	$n^{n-1}$
#Markierte, nicht gewurzelte Bäume	$n^{n-2}$

## 4.11 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 // Laufzeit: O(#game)
2 bool WinNimm(vector<int> game) {
3     int result = 0;
4     for(int s: game) result ^= s;
5     return s > 0;
6 }

```

## 4.12 3D-Kugeln

```

1 // Great Circle Distance mit Längen- und Breitengrad.
2 double gcDist(double pLat, double pLon, double qLat, double qLon, double
  radius) {
3     pLat *= PI / 180; pLon *= PI / 180; qLat *= PI / 180; qLon *= PI / 180;
4     return radius * acos(cos(pLat) * cos(pLon) * cos(qLat) * cos(qLon) +
5                           cos(pLat) * sin(pLon) * cos(qLat) * sin(qLon) +
6                           sin(pLat) * sin(qLat));
7 }
8
9 // Great Circle Distance mit kartesischen Koordinaten.
10 double gcDist(point p, point q) {
11     return acos(p.x * q.x + p.y * q.y + p.z * q.z);
12 }
13
14 // 3D Punkt in kartesischen Koordinaten.
15 struct point{
16     double x, y, z;
17     point() {}
18     point(double x, double y, double z) : x(x), y(y), z(z) {}
19     point(double lat, double lon) {
20         lat *= PI / 180.0; lon *= PI / 180.0;
21         x = cos(lat) * sin(lon); y = cos(lat) * cos(lon); z = sin(lat);
22     }
23 };

```

## 4.13 Big Integers

```

1 // Bislang keine Division. Multiplikation nach Schulmethode.
2 #define PLUS 0
3 #define MINUS 1
4 #define BASE 1000000000
5
6 struct bigint {
7     int sign;
8     vector<ll> digits;
9
10    // Initialisiert mit 0.
11    bigint(void) {
12        sign = PLUS;
13    }
14
15    // Initialisiert mit kleinem Wert.
16    bigint(ll value) {
17        if (value == 0) sign = PLUS;
18        else {
19            sign = value >= 0 ? PLUS : MINUS;
20            value = abs(value);
21            while (value) {
22                digits.push_back(value % BASE);
23                value /= BASE;

```

```

24    }
25    }
26 }
27
28 // Initialisiert mit C-String. Kann nicht mit Vorzeichen umgehen.
29 bigint(char *str, int length) {
30     int base = 1;
31     ll digit = 0;
32     for (int i = length - 1; i >= 0; i--) {
33         digit += base * (str[i] - '0');
34         if (base * 10 == BASE) {
35             digits.push_back(digit);
36             digit = 0;
37             base = 1;
38         } else base *= 10;
39     }
40     if (digit != 0) digits.push_back(digit);
41     sign = PLUS;
42 }
43
44 // Löscht führende Nullen und macht -0 zu 0.
45 void trim() {
46     while (digits.size() > 0 && digits[digits.size() - 1] == 0) digits.
47         pop_back();
48     if (digits.size() == 0 && sign == MINUS) sign = PLUS;
49 }
50
51 // Gibt die Zahl aus.
52 void print() {
53     if (digits.size() == 0) {
54         printf("0");
55         return;
56     }
57     if (sign == MINUS) printf("-");
58     printf("%lld", digits[digits.size() - 1]);
59     for (int i = digits.size() - 2; i >= 0; i--) {
60         printf("%09lld", digits[i]);
61     }
62 }
63
64 // Kleiner-oder-gleich-Vergleich.
65 bool operator<=(bigint &a, bigint &b) {
66     if (a.digits.size() == b.digits.size()) {
67         int idx = a.digits.size() - 1;
68         while (idx >= 0) {
69             if (a.digits[idx] < b.digits[idx]) return true;
70             else if (a.digits[idx] > b.digits[idx]) return false;
71             idx--;
72         }
73         return true;
74     }
75     return a.digits.size() < b.digits.size();
76 }
77

```



```

78 // Kleiner-Vergleich.
79 bool operator<(bigint &a, bigint &b) {
80     if (a.digits.size() == b.digits.size()) {
81         int idx = a.digits.size() - 1;
82         while (idx >= 0) {
83             if (a.digits[idx] < b.digits[idx]) return true;
84             else if (a.digits[idx] > b.digits[idx]) return false;
85             idx--;
86         }
87         return false;
88     }
89     return a.digits.size() < b.digits.size();
90 }
91
92 void sub(bigint *a, bigint *b, bigint *c);
93
94 // a+b=c. a, b, c dürfen gleich sein.
95 void add(bigint *a, bigint *b, bigint *c) {
96     if (a->sign == b->sign) c->sign = a->sign;
97     else {
98         if (a->sign == MINUS) {
99             a->sign ^= 1;
100             sub(b, a, c);
101             a->sign ^= 1;
102         } else {
103             b->sign ^= 1;
104             sub(a, b, c);
105             b->sign ^= 1;
106         }
107         return;
108     }
109
110     c->digits.resize(max(a->digits.size(), b->digits.size()));
111     ll carry = 0;
112     int i = 0;
113     for (; i < (int)min(a->digits.size(), b->digits.size()); i++) {
114         ll sum = carry + a->digits[i] + b->digits[i];
115         c->digits[i] = sum % BASE;
116         carry = sum / BASE;
117     }
118     if (i < (int)a->digits.size()) {
119         for (; i < (int)a->digits.size(); i++) {
120             ll sum = carry + a->digits[i];
121             c->digits[i] = sum % BASE;
122             carry = sum / BASE;
123         }
124     } else {
125         for (; i < (int)b->digits.size(); i++) {
126             ll sum = carry + b->digits[i];
127             c->digits[i] = sum % BASE;
128             carry = sum / BASE;
129         }
130     }
131     if (carry) {
132         c->digits.push_back(carry);

```

```

133     }
134 }
135
136 // a-b=c. c darf a oder b sein. a und b müssen verschieden sein.
137 void sub(bigint *a, bigint *b, bigint *c) {
138     if (a->sign == MINUS || b->sign == MINUS) {
139         b->sign ^= 1;
140         add(a, b, c);
141         b->sign ^= 1;
142         return;
143     }
144
145     if (a < b) {
146         sub(b, a, c);
147         c->sign = MINUS;
148         c->trim();
149         return;
150     }
151
152     c->digits.resize(a->digits.size());
153     ll borrow = 0;
154     int i = 0;
155     for (; i < (int)b->digits.size(); i++) {
156         ll diff = a->digits[i] - borrow - b->digits[i];
157         if (a->digits[i] > 0) borrow = 0;
158         if (diff < 0) {
159             diff += BASE;
160             borrow = 1;
161         }
162         c->digits[i] = diff % BASE;
163     }
164     for (; i < (int)a->digits.size(); i++) {
165         ll diff = a->digits[i] - borrow;
166         if (a->digits[i] > 0) borrow = 0;
167         if (diff < 0) {
168             diff += BASE;
169             borrow = 1;
170         }
171         c->digits[i] = diff % BASE;
172     }
173     c->trim();
174 }
175
176 // Ziffernmultiplikation a*b=c. b und c dürfen gleich sein. a muss
    kleiner BASE sein.
177 void digitMul(ll a, bigint *b, bigint *c) {
178     if (a == 0) {
179         c->digits.clear();
180         c->sign = PLUS;
181         return;
182     }
183     c->digits.resize(b->digits.size());
184     ll carry = 0;
185     for (int i = 0; i < (int)b->digits.size(); i++) {
186         ll prod = carry + b->digits[i] * a;

```

```

187     c->digits[i] = prod % BASE;
188     carry = prod / BASE;
189 }
190 if (carry) c->digits.push_back(carry);
191 c->sign = (a > 0) ? b->sign : 1 ^ b->sign;
192 c->trim();
193 }
194
195 // Zifferndivision b/a=c. b und c dürfen gleich sein. a muss kleiner BASE
    sein.
196 void digitDiv(ll a, bigint *b, bigint *c) {
197     c->digits.resize(b->digits.size());
198     ll carry = 0;
199     for (int i = (int)b->digits.size() - 1; i >= 0; i--) {
200         ll quot = (carry * BASE + b->digits[i]) / a;
201         carry = carry * BASE + b->digits[i] - quot * a;
202         c->digits[i] = quot;
203     }
204     c->sign = b->sign ^ (a < 0);
205     c->trim();
206 }
207
208 // a*b=c. c darf weder a noch b sein. a und b dürfen gleich sein.
209 void mult(bigint *a, bigint *b, bigint *c) {
210     bigint row = *a;
211     bigint tmp;
212     c->digits.clear();
213     for (int i = 0; i < (int)b->digits.size(); i++) {
214         digitMul(b->digits[i], &row, &tmp);
215         add(&tmp, c, c);
216         row.digits.insert(row.digits.begin(), 0);
217     }
218     c->sign = a->sign != b->sign;
219     c->trim();
220 }
221
222 // Berechnet eine kleine Zehnerpotenz.
223 inline ll pow10(int n) {
224     ll res = 1;
225     for (int i = 0; i < n; i++) res *= 10;
226     return res;
227 }
228
229 // Berechnet eine große Zehnerpotenz.
230 void power10(ll e, bigint *out) {
231     out->digits.assign(e / 9 + 1, 0);
232     if (e % 9) out->digits[out->digits.size() - 1] = pow10(e % 9);
233     else out->digits[out->digits.size() - 1] = 1;
234 }
235
236 // Nimmt eine Zahl module einer Zehnerpotenz 10^e.
237 void mod10(int e, bigint *a) {
238     int idx = e / 9;
239     if ((int)a->digits.size() < idx + 1) return;
240     if (e % 9) {

```

```

241         a->digits.resize(idx + 1);
242         a->digits[idx] %= pow10(e % 9);
243     } else {
244         a->digits.resize(idx);
245     }
246     a->trim();
247 }

```

## 5 Strings

### 5.1 KNUTH-MORRIS-PRATT-Algorithmus

```

1 // Laufzeit: O(n + m), n = #Text, m = #Pattern
2 vector<int> kmp_preprocessing(string &sub) {
3     vector<int> b(sub.length() + 1);
4     b[0] = -1;
5     int i = 0, j = -1;
6     while (i < (int)sub.length()) {
7         while (j >= 0 && sub[i] != sub[j]) j = b[j];
8         i++; j++;
9         b[i] = j;
10    }
11    return b;
12 }
13
14 vector<int> kmp_search(string &s, string &sub) {
15     vector<int> pre = kmp_preprocessing(sub);
16     vector<int> result;
17     int i = 0, j = 0;
18     while (i < (int)s.length()) {
19         while (j >= 0 && s[i] != sub[j]) j = pre[j];
20         i++; j++;
21         if (j == (int)sub.length()) {
22             result.push_back(i - j);
23             j = pre[j];
24         }
25     }
26     return result;
27 }

```

### 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(automaton) aufrufen.

```

```

6 // 4) Mit automaton = go(automaton, c) in nächsten Zustand wechseln.
  DANACH: Wenn patterns-Vektor nicht leer
7 // ist: Hier enden alle enthaltenen Patterns.
8 // ACHTUNG: Die Zahlenwerte der auftretenden Buchstaben müssen zusammenhängend sein und bei 0 beginnen!
9 struct vertex {
10     vertex *next[ALPHABET_SIZE], *failure;
11     char character;
12     vector<int> patterns; // Indizes der Patterns, die hier enden.
13     vertex() { for (int i = 0; i < ALPHABET_SIZE; i++) next[i] = NULL; }
14 };
15
16 void addString(vertex *v, string &pattern, int patternIdx) {
17     for (int i = 0; i < (int)pattern.length(); i++) {
18         if (!v->next[(int)pattern[i]]) {
19             vertex *w = new vertex();
20             w->character = pattern[i];
21             v->next[(int)pattern[i]] = w;
22         }
23         v = v->next[(int)pattern[i]];
24     }
25     v->patterns.push_back(patternIdx);
26 }
27
28 void finishAutomaton(vertex *v) {
29     for (int i = 0; i < ALPHABET_SIZE; i++)
30         if (!v->next[i]) v->next[i] = v;
31
32     queue<vertex*> q;
33     for (int i = 0; i < ALPHABET_SIZE; i++) {
34         if (v->next[i] != v) {
35             v->next[i]->failure = v;
36             q.push(v->next[i]);
37         }
38     }
39     while (!q.empty()) {
40         vertex *r = q.front(); q.pop();
41         for (int i = 0; i < ALPHABET_SIZE; i++) {
42             if (r->next[i]) {
43                 q.push(r->next[i]);
44                 vertex *f = r->failure;
45                 while (f->next[i]) f = f->failure;
46                 r->next[i]->failure = f->next[i];
47                 for (int j = 0; j < (int)f->next[i]->patterns.size(); j++) {
48                     r->next[i]->patterns.push_back(f->next[i]->patterns[j]);
49                 }
50             }
51         }
52     }
53     vertex* go(vertex *v, char c) {
54         if (v->next[(int)c]) return v->next[(int)c];
55         else return go(v->failure, c);
56     }
57 }

```

### 5.3 LEVENSHTAIN-Distanz

```

1 // Laufzeit: O(nm), Speicher: O(m), n = #s1, m = #s2
2 int levenshtein(string& s1, string& s2) {
3     int len1 = s1.size(), len2 = s2.size();
4     vector<int> col(len2 + 1), prevCol(len2 + 1);
5     for (int i = 0; i < len2 + 1; i++) prevCol[i] = i;
6     for (int i = 0; i < len1; i++) {
7         col[0] = i + 1;
8         for (int j = 0; j < len2; j++)
9             col[j+1] = min(min(prevCol[j+1] + 1, col[j] + 1), prevCol[j] + (s1[i] == s2[j] ? 0 : 1));
10        col.swap(prevCol);
11    }
12    return prevCol[len2];
13 }

```

### 5.4 Trie

```

1 // Implementierung für Kleinbuchstaben.
2 struct node {
3     node *e[26];
4     int c = 0; // Anzahl der Wörter, die an diesem 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) { // Laufzeit: O(|txt|)
9     if(s == (int)txt.size()) root->c++;
10    else {
11        int idx = (int)(txt[s] - 'a');
12        if(root->e[idx] == NULL) root->e[idx] = new node();
13        insert(root->e[idx], txt, s+1);
14    }
15 }
16
17 int contains(node *root, string &txt, int s) { // Laufzeit: O(|txt|)
18     if(s == txt.size()) return root->c;
19     int idx = (int)(txt[s] - 'a');
20     if(root->e[idx] != NULL) return contains(root->e[idx], txt, s + 1);
21     else return 0;
22 }

```

### 5.5 Suffix-Array

```

1 //longest common substring in one string (overlapping not excluded)
2 //contains suffix array
3 :-----
4
5 int cmp(string &s, vector<vector<int>> &v, int i, int vi, int u, int l) {
6     int vi2 = (vi + 1) % 2, u2 = u + i / 2, l2 = l + i / 2;
7     if(i == 1) return s[u] - s[l];
8     else if (v[vi2][u] != v[vi2][l]) return (v[vi2][u] - v[vi2][l]);
9 }

```

```

7  else { //beide groesser tiff 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(
26             s, v, i, vi, a[z], a[z-1]) == 0 ? 0 : 1);
27     }
28
29     int r = 0, m=0, c=0;
30     for(int i = 0; i < a.size() - 1; i++) {
31         c = 0;
32         while(a[i]+c < s.length() && a[i+1]+c < s.length() && s[a[i]+c] == s[
33             a[i+1]+c]) c++;
34         if(c > m) r=i, m=c;
35     }
36     return m == 0 ? "" : s.substr(a[r], m);
37 }

```

## 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
9          [0].length()), a[k].s = (k < s[0].length() ? 0 : 1);
10     sort(a.begin(), a.end(), [&] (const lcse &u, const lcse &l) {
11         int ui = u.i, li = l.i;
12         while(ui < s[u.s].length() && li < s[l.s].length()) {
13             if(s[u.s][ui] < s[l.s][li]) return true;
14             else if(s[u.s][ui] > s[l.s][li]) return false;
15             ui++; li++;
16         }
17     }
18 }

```

```

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].
23         length() && s[a[i].s][a[i].i+c] == s[a[i+1].s][a[i+1].i+c]) c++;
24     if(c > m) r=i, m=c;
25 }
26 return m == 0 ? "" : s[a[r].s].substr(a[r].i, m);
27 }

```

## 5.7 Longest Common Subsequence

```

1  string lcsc(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: Scanner ist sehr langsam. Bei großen Eingaben muss ein Buffered Reader verwendet werden.

```

1  Scanner in = new Scanner(System.in); // java.util.Scanner
2  String line = in.nextLine(); // Liest die nächste Zeile.
3  int num = in.nextInt(); // Liest das nächste Token als int.
4  double num2 = in.nextDouble(); // Liest das nächste Token als 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

```

1 // Berechnet this +,*,/,- val.
2 BigInteger add(BigInteger val), multiply(BigInteger val), divide(
  BigInteger val), subtract(BigInteger val)
3
4 // Berechnet this^e.
5 BigInteger pow(BigInteger e)
6
7 // Bit-Operationen.
8 BigInteger and(BigInteger val), or(BigInteger val), xor(BigInteger val),
  not(), shiftLeft(int n), shiftRight(int n)
9
10 // Berechnet den ggT von abs(this) und abs(val).
11 BigInteger gcd(BigInteger val)
12
13 // Berechnet 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 // Berechnet die nächste Zahl, die größer und wahrscheinlich prim ist.
17 BigInteger nextProbablePrime()
18
19 // Berechnet int/long/float/double-Wert. Ist die Zahl zu groß werden
  // die niedrigsten Bits konvertiert.
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 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 }

```

### 7.3 Bit Operations

```

1 // Bit an Position j auslesen.
2 (a & (1 << j)) != 0
3 // Bit an Position j setzen.
4 a |= (1 << j)
5 // Bit an Position j löschen.
6 a &= ~(1 << j)
7 // Bit an Position j umkehren.
8 a ^= (1 << j)
9 // Wert des niedrigsten gesetzten Bits.
10 (a & -a)
11 // Setzt alle Bits auf 1.
12 a = -1
13 // Setzt die ersten n Bits auf 1. Achtung: Overflows.
14 a = (1 << n) - 1

```

### 7.4 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) { // Gibt Index des letzten Überlebenden zurück
  // ck, 1-basiert.
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) { // Gibt Index des letzten Überlebenden
  zurück, 0-basiert.
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  $0, \dots, n-1$ !

## 7.5 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 Gewicht  $c$  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  $\infty$  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 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 8)
- Tobi, cool down!

## 7.6 Sonstiges

```

1 // Alles-Header.
2 #include <bits/stdc++.h>
3
4 // Setzt das deutsche Tastaturlayout.
5 setxkbmap de
6
7 // Schnelle Ein-/Ausgabe mit cin/cout.
8 ios::sync_with_stdio(false);
9
10 // Set mit eigener Sortierfunktion. Typ muss nicht explizit angegeben
    werden.
11 set<point2, decltype(comp)> set1(comp);
12
13 // PI
14 #define PI (2*acos(0))
15
16 // STL-Debugging, Compiler flags.
17 -D_GLIBCXX_DEBUG
18 #define _GLIBCXX_DEBUG
19
20 // 128-Bit Integer. Muss zum Einlesen/Ausgeben in einen int oder long
    long gecastet werden.
21 __int128

```