

Team Contest Reference

ChaosKITs
Karlsruhe Institute of Technology

15. Oktober 2016

Inhaltsverzeichnis

1	Datenstrukturen	2	4.6	Primzahlsieb von ERATOSTHENES	12
1.1	Union-Find	2	4.7	Primzahltest & Faktorisierung	12
1.2	Segmentbaum	2	4.8	Binomialkoeffizienten	13
1.3	Fenwick Tree	2	4.9	EULERSche φ -Funktion	13
1.4	Range Minimum Query	2	4.10	Primitivwurzeln	13
1.5	STL-Tree	3	4.11	Diskreter Logarithmus	13
2	Graphen	3	4.12	Polynome & FFT	13
2.1	Minimale Spannbäume	3	4.13	3D-Kugeln	14
2.1.1	Kruskal	3	4.14	Kombinatorik	14
2.2	Kürzeste Wege	3	4.14.1	Berühmte Zahlen	14
2.2.1	Algorithmus von DIJKSTRA	3	4.14.2	Verschiedenes	15
2.2.2	BELLMANN-FORD-Algorithmus	3	4.15	Satz von SPRAGUE-GRUNDY	15
2.2.3	FLOYD-WARSHALL-Algorithmus	4	4.16	Big Integers	15
2.3	Strongly Connected Components (TARJANS-Algorithmus)	4	5	Strings	17
2.4	Artikulationspunkte und Brücken	4	5.1	KNUTH-MORRIS-PRATT-Algorithmus	17
2.5	Eulertouren	4	5.2	AHO-CORASICK-Automat	18
2.6	Lowest Common Ancestor	5	5.3	Trie	18
2.7	Max-Flow	5	5.4	Suffix-Array	18
2.7.1	Capacity Scaling	5	5.5	Suffix-Automaton	19
2.7.2	Push Relabel	6	5.6	Longest Common Subsequence	19
2.7.3	Anwendungen	6	5.7	Rolling Hash	20
2.8	Min-Cost-Max-Flow	7	6	Java	20
2.9	Maximal Cardinatlity Bipartite Matching	7	6.1	Introduction	20
3	Geometrie	8	6.2	BigInteger	20
3.1	Closest Pair	8	7	Sonstiges	20
3.2	Geraden	8	7.1	2-SAT	20
3.3	Konvexe Hülle	8	7.2	Zeileneingabe	21
3.4	Formeln - std::complex	9	7.3	Bit Operations	21
4	Mathe	10	7.4	Josephus-Problem	21
4.1	ggT, kgV, erweiterter euklidischer Algorithmus	10	7.5	Gemischtes	21
4.2	Mod-Exponent über \mathbb{F}_p	10	7.6	Sonstiges	22
4.3	LGS über \mathbb{F}_p	11			

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:  $O(|E| \cdot \log(|E|))$ 
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     }

```

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(NUM_VERTICES, INF), parent(NUM_VERTICES, -1);
5     dist[start] = 0; pq.push(ii(0, start));
6
7     while (!pq.empty()) {
8         ii front = pq.top(); pq.pop();
9         int curNode = front.second, curDist = front.first;
10        if (curDist > dist[curNode]) continue; // WICHTIG!
11
12        for (auto n : adjlist[curNode]) {
13            int nextNode = n.first, nextDist = curDist + n.second;
14            if (nextDist < dist[nextNode]) {
15                dist[nextNode] = nextDist; parent[nextNode] = curNode;
16                pq.push(ii(nextDist, nextNode));
17            }

```

2.2.2 BELLMANN-FORD-Algorithmus

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

```

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

```

2.2.3 FLOYD-WARSHALL-Algorithmus

```

1 // Initialisiere mat: mat[i][i] = 0, mat[i][j] = INF falls i & j nicht
2 // adjazent, Länge sonst. Laufzeit: O(|V|^3)
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    }
14 }

```

- Nur negative Werte sollten die Nullen überschreiben.
- Von parallelen Kanten sollte nur die günstigste gespeichert werden.
- i liegt genau dann auf einem negativen Kreis, wenn $\text{dist}[i][i] < 0$ ist.
- Wenn für c gilt, dass $\text{dist}[u][c] \neq \text{INF}$ && $\text{dist}[c][v] \neq \text{INF}$ && $\text{dist}[c][c] < 0$, 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; // sccs enthält den Index der SCC pro Knoten.
6 stack<int> s;
7
8 void visit(int v) {
9     visited[v] = true;
10    d[v] = low[v] = counter++;
11    s.push(v); inStack[v] = true;
12
13    for (auto u : adjlist[v]) {
14        if (!visited[u]) {
15            visit(u);
16            low[v] = min(low[v], low[u]);
17        } else if (inStack[u]) {
18            low[v] = min(low[v], low[u]);
19        }
20    }
21
22    if (d[v] == low[v]) {
23        int u;
24        do {
25            u = s.top(); s.pop(); inStack[u] = false;
26            sccs[u] = sccCounter;
27        } while (u != v);
28        sccCounter++;
29    }
30 }

```

```

29
30 void scc() {
31     visited.assign(adjlist.size(), false);
32     d.assign(adjlist.size(), -1);
33     low.assign(adjlist.size(), -1);
34     inStack.assign(adjlist.size(), false);
35     sccs.resize(adjlist.size(), -1);
36
37     counter = sccCounter = 0;
38     for (int i = 0; i < (int)adjlist.size(); i++) {
39         if (!visited[i]) {
40             visit(i);
41         }
42     }
43 }

```

2.4 Artikulationspunkte und Brücken

```

1 // Laufzeit: O(|V|+|E|)
2 vector< vector<int> > adjlist;
3 vector<bool> isArt;
4 vector<int> d, low;
5 int counter, root; // root >= 2 <=> Wurzel Artikulationspunkt
6 vector<ii> bridges; // Nur fuer Brücken.
7
8 void dfs(int v, int parent) { // Mit parent=-1 aufrufen.
9     d[v] = low[v] = counter++;
10    if (parent == 0) root++;
11
12    for (auto w : adjlist[v]) {
13        if (!d[w]) {
14            dfs(w, v);
15            if (low[w] >= d[v]) isArt[v] = true;
16            if (low[w] > d[v]) bridges.push_back(ii(v, w));
17            low[v] = min(low[v], low[w]);
18        } else if (w != parent) {
19            low[v] = min(low[v], d[w]);
20        }
21    }
22
23 void findArticulationPoints() {
24     counter = 1; // Nicht auf 0 setzen!
25     low.resize(adjlist.size());
26     d.assign(adjlist.size(), 0);
27     isArt.assign(adjlist.size(), false);
28     bridges.clear(); //nur fuer Bruecken
29     for (int v = 0; v < (int)adjlist.size(); v++) if (!d[v]) visit(v, -1);
30 }

```

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.
- Algorithmus schlägt nicht fehl, falls kein Eulerzyklus existiert. Die Existenz muss separat geprüft werden.

```

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, otherIdx;
3 vector<int> cycle, validIdx;
4
5 // Vertauscht Kanten mit Indizes a und b von Knoten n.
6 void swapEdges(int n, int a, int b) {
7     int neighA = adjlist[n][a], neighB = adjlist[n][b];
8     int idxNeighA = otherIdx[n][a], idxNeighB = otherIdx[n][b];
9     swap(adjlist[n][a], adjlist[n][b]);
10    swap(otherIdx[n][a], otherIdx[n][b]);
11    otherIdx[neighA][idxNeighA] = b;
12    otherIdx[neighB][idxNeighB] = a;
13 }
14
15 // Entfernt Kante i von Knoten n (und die zugehörige Rückwärtskante).
16 void removeEdge(int n, int i) {
17     int other = adjlist[n][i];
18     if (other == n) { //Schlingen.
19         validIdx[n]++;
20         return;
21     }
22     int otherIndex = otherIdx[n][i];
23     validIdx[n]++;
24     if (otherIndex != validIdx[other]) {
25         swapEdges(other, otherIndex, validIdx[other]);
26     }
27     validIdx[other]++;
28 }
29
30 // Findet Eulerzyklus an Knoten n startend.
31 // Teste vorher, dass Graph zusammenhängend ist! Isolierten Knoten?
32 // Teste vorher, ob Eulerzyklus überhaupt existiert!
33 void euler(int n) {

```

```

34 while (validIdx[n] < (int)adjlist[n].size()) {
35     int nn = adjlist[n][validIdx[n]];
36     removeEdge(n, validIdx[n]);
37     euler(nn);
38 }
39 cycle.push_back(n); // Zyklus in cycle in umgekehrter Reihenfolge.
40 }

```

2.6 Lowest Common Ancestor

```

1 vector<int> visited(2*MAX_N), first(MAX_N, 2*MAX_N), depth(2*MAX_N);
2 vector<vector<int>> graph(MAX_N);
3
4 void initLCA(int gi, int d, int &c) { // Laufzeit: O(n)
5     visited[c] = gi, depth[c] = d, first[gi] = min(c, first[gi]), c++;
6     for(int gn : graph[gi]) {
7         initLCA(gn, d+1, c);
8         visited[c] = gi, depth[c] = d, c++;
9     }
10
11 int getLCA(int a, int b) { // Laufzeit: O(1)
12     return visited[queryRMQ(
13         min(first[a], first[b]), max(first[a], first[b]))];
14 }
15
16 // Benutzung:
17 int c = 0;
18 initLCA(0, 0, c);
19 initRMQ(); // Ersetze das data im RMQ-Code von oben durch depth.

```

2.7 Max-Flow

2.7.1 Capacity Scaling

Gut bei dünn besetzten Graphen.

```

1 // Ford Fulkerson mit Capacity Scaling. Laufzeit: O(|E|^2*log(C))
2 struct MaxFlow { // Muss mit new erstellt werden!
3     static const int MAX_N = 500; // #Knoten, egal für die Laufzeit.
4     struct edge { int dest, rev; ll cap, flow; };
5     vector<edge> adjlist[MAX_N];
6     int visited[MAX_N] = {0}, target, dfsCounter = 0;
7     ll capacity;
8
9     bool dfs(int x) {
10         if (x == target) return 1;
11         if (visited[x] == dfsCounter) return 0;
12         visited[x] = dfsCounter;
13         for (edge &e : adjlist[x]) {
14             if (e.cap >= capacity && dfs(e.dest)) {
15                 e.cap -= capacity; adjlist[e.dest][e.rev].cap += capacity;

```

```

16         e.flow += capacity; adjlist[e.dest][e.rev].flow -= capacity;
17         return 1;
18     }}
19     return 0;
20 }
21
22 void addEdge(int u, int v, ll c) {
23     adjlist[u].push_back(edge {v, (int)adjlist[v].size(), c, 0});
24     adjlist[v].push_back(edge {u, (int)adjlist[u].size() - 1, 0, 0});
25 }
26
27 ll maxFlow(int s, int t) {
28     capacity = 1L << 62;
29     target = t;
30     ll flow = 0L;
31     while (capacity) {
32         while (dfsCounter++, dfs(s)) flow += capacity;
33         capacity /= 2;
34     }
35     return flow;
36 }
37 };

```

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));
9         memset(flow, 0L, sizeof(flow));
10        memset(excess, 0L, sizeof(excess));
11        memset(height, 0, sizeof(height));
12        memset(list, 0, sizeof(list));
13        memset(seen, 0, sizeof(seen));
14    }
15
16    inline void addEdge(int u, int v, ll c) { capacities[u][v] += c; }
17
18    void push(int u, int v) {
19        ll send = min(excess[u], capacities[u][v] - flow[u][v]);
20        flow[u][v] += send; flow[v][u] -= send;
21        excess[u] -= send; excess[v] += send;
22    }
23
24    void relabel(int u) {
25        int minHeight = INT_MAX / 2;
26        for (int v = 0; v < n; v++) {
27            if (capacities[u][v] - flow[u][v] > 0) {

```

```

28                minHeight = min(minHeight, height[v]);
29                height[u] = minHeight + 1;
30            }}
31
32    void discharge(int u) {
33        while (excess[u] > 0) {
34            if (seen[u] < n) {
35                int v = seen[u];
36                if (capacities[u][v] - flow[u][v] > 0 && height[u] > height[v]) {
37                    push(u, v);
38                } else seen[u]++;
39            } else {
40                relabel(u);
41                seen[u] = 0;
42            }}
43
44    void moveToFront(int u) {
45        int temp = list[u];
46        for (int i = u; i > 0; i--) 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++)
52            if (i != source && i != target) 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    }
72 };

```

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 den unterschiedlichen Pfaden ohne gemeinsame Kanten.

- **Maximum Independent Paths**

Finde die maximale Anzahl an Pfaden 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 den unterschiedlichen Pfaden 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 static const ll flowlimit = 1LL << 60; // Größer als der maximale Fluss.
2 struct MinCostFlow { // Mit new erstellen!
3     static const int maxn = 400; // Größer als die Anzahl der Knoten.
4     static const int maxm = 5000; // Größer als die Anzahl der Kanten.
5     struct edge { int node, next; ll flow, value; } edges[maxm << 1];
6     int graph[maxn], queue[maxn], pre[maxn], con[maxn];
7     int 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 &&
41                     dis[node] > dis[now] + edges[point].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                 return pre[target] != -1;
51             }
52
53     void extend() {
54         ll w = flowlimit;
55         for (int u = target; pre[u] != u; u = pre[u])
56             w = min(w, edges[con[u]].flow);
57         maxflow += w;
58         mincost += dis[target] * w;
59         for (int u = target; pre[u] != u; u = pre[u]) {
60             edges[con[u]].flow -= w;
61             edges[inverse(con[u])].flow += w;
62         }
63
64     void mincostflow() {
65         maxflow = mincost = 0;
66         while (SPFA()) extend();
67     }
68 };

```

2.9 Maximal Cardinality Bipartite Matching

```

1 // Laufzeit: O(n*min(ans^2, |E|))
2 vector< vector<int> > adjlist; // Von links nach rechts.
3 vector<int> pairs; // Der gematchte Knoten oder -1.
4 vector<bool> visited;
5
6 bool dfs(int v) {
7     if (visited[v]) return false;
8     visited[v] = true;
9     for (auto w : adjlist[v]) if (pairs[w] < 0 || dfs(pairs[w])) {
10         pairs[w] = v; pairs[v] = w; return true;
11     }
12     return false;
13 }
14
15 int kuhn(int n) { // n = #Knoten links.
16     pairs.assign(adjlist.size(), -1);
17     int ans = 0;
18     // Greedy Matching. Optionale Beschleunigung.
19     for (int i = 0; i < n; i++) for (auto w : adjlist[i])
20         if (pairs[w] == -1) pairs[i] = w; pairs[w] = i; ans++; break; }

```



```

21 for (int i = 0; i < n; i++) if (pairs[i] == -1) {
22     visited.assign(n, false);
23     ans += dfs(i);
24 }
25 return ans; // Größe des Matchings.
26 }

```

3 Geometrie

3.1 Closest Pair

```

1 double squaredDist(pt a, pt b) {
2     return (a.fst-b.fst) * (a.fst-b.fst) + (a.snd-b.snd) * (a.snd-b.snd);
3 }
4
5 bool compY(pt a, pt b) {
6     if (a.snd == b.snd) return a.fst < b.fst;
7     return a.snd < b.snd;
8 }
9
10 // points.size() > 1 und alle Punkte müssen verschieden sein!
11 double shortestDist(vector<pt> &points) {
12     set<pt, bool(*)>(pt, pt)> 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->fst - right->fst) >= sqrtOpt) {
20             status.erase(*(left++));
21         } else {
22             auto lower = status.lower_bound(pt(-1e20, right->snd - sqrtOpt));
23             auto upper = status.upper_bound(pt(-1e20, right->snd + 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 // Nicht complex<double> benutzen. Eigene struct schreiben.
2 struct line {
3     double a, b, c; // ax + by + c = 0; vertikale Line: b = 0, sonst: b = 1
4 };
5
6 line pointsToLine(pt p1, pt p2) {
7     line l;
8     if (fabs(p1.x - p2.x) < EPSILON) {
9         l.a = 1; l.b = 0.0; l.c = -p1.x;
10    } else {
11        l.a = -(double)(p1.y - p2.y) / (p1.x - p2.x);
12        l.b = 1.0;
13        l.c = -(double)(l.a * p1.x) - p1.y;
14    }
15    return l;
16 }
17
18 bool areParallel(line l1, line l2) {
19     return (fabs(l1.a - l2.a) < EPSILON) && (fabs(l1.b - l2.b) < EPSILON);
20 }
21
22 bool areSame(line l1, line l2) {
23     return areParallel(l1, l2) && (fabs(l1.c - l2.c) < EPSILON);
24 }
25
26 bool areIntersect(line l1, line l2, pt &p) {
27     if (areParallel(l1, l2)) return false;
28     p.x = (l2.b * l1.c - l1.b * l2.c) / (l2.a * l1.b - l1.a * l2.b);
29     if (fabs(l1.b) > EPSILON) p.y = -(l1.a * p.x + l1.c);
30     else p.y = -(l2.a * p.x + l2.c);
31     return true;
32 }

```

3.3 Konvexe Hülle

```

1 // Laufzeit: O(n*log(n))
2
3 ll cross(const pt p, const pt a, const pt b) {
4     return (a.x - p.x) * (b.y - p.y) - (a.y - p.y) * (b.x - p.x);
5 }
6
7 // Punkte auf der konvexen Hülle, gegen den Uhrzeigersinn sortiert.
8 // Kollineare Punkte nicht enthalten, entferne dafür "=" im CCW-Test.
9 // Achtung: Der erste und letzte Punkt im Ergebnis sind gleich.
10 // Achtung: Alle Punkte müssen verschieden sein.
11 vector<pt> convexHull(vector<pt> p){
12     int n = p.size(), k = 0;
13     vector<pt> h(2 * n);
14     sort(p.begin(), p.end());
15     for (int i = 0; i < n; i++) { // Untere Hülle.
16         while (k >= 2 && cross(h[k - 2], h[k - 1], p[i]) <= 0.0) k--;
17         h[k++] = p[i];
18     }
19 }

```



```

18 }
19 for (int i = n - 2, t = k + 1; i >= 0; i--) { // Obere Hülle.
20     while (k >= t && cross(h[k - 2], h[k - 1], p[i]) <= 0.0) k--;
21     h[k++] = p[i];
22 }
23 h.resize(k);
24 return h;
25 }

```

3.4 Formeln - std::complex

```

1 // Komplexe Zahlen als Darstellung für Punkte. Wenn immer möglich
2 // complex<int> verwenden. Funktionen wie abs() geben dann int zurück.
3 typedef pt complex<double>;
4
5 // Winkel zwischen Punkt und x-Achse in [0, 2 * PI), bzw. zwischen a, 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) { return real(conj(a) * b); }
16
17 // Kreuzprodukt, 0, falls kollinear.
18 double cross(pt a, pt b) { return imag(conj(a) * b); }
19
20 // Flächeninhalt eines Dreiecks bei bekannten Eckpunkten.
21 double areaOfTriangle(pt a, pt b, pt c) {
22     return abs(cross(b - a, c - a)) / 2.0;
23 }
24
25 // Flächeninhalt eines Dreiecks bei bekannten Seitenlängen.
26 double areaOfTriangle(double a, double b, double c) {
27     double s = (a + b + c) / 2;
28     return sqrt(s * (s-a) * (s-b) * (s-c));
29 }
30
31 // Sind die Dreiecke a1, b1, c1, and a2, b2, c2 ähnlich?
32 // Erste Zeile testet Ähnlichkeit mit gleicher Orientierung,
33 // zweite Zeile testet Ähnlichkeit mit unterschiedlicher Orientierung
34 bool similar (pt a1, pt b1, pt c1, pt a2, pt b2, pt c2) {
35     return (
36         (b2-a2) * (c1-a1) == (b1-a1) * (c2-a2) ||
37         (b2-a2) * (conj(c1)-conj(a1)) == (conj(b1)-conj(a1)) * (c2-a2)
38     );
39 }
40
41 // -1 => gegen den Uhrzeigersinn, 0 => kollinear, 1 => im Uhrzeigersinn.
42 // Einschränken der Rückgabe auf [-1,1] ist sicherer gegen Overflows.

```

```

43 double orientation(pt a, pt b, pt c) {
44     double orien = cross(b - a, c - a);
45     if (abs(orien) < EPSILON) return 0; // Braucht großes EPSILON: ~1e-6
46     return orien < 0 ? -1 : 1;
47 }
48
49 // Test auf Streckenschnitt zwischen a-b und c-d.
50 bool lineSegmentIntersection(pt a, pt b, pt c, pt d) {
51     if (orientation(a, b, c) == 0 && orientation(a, b, d) == 0) {
52         double dist = abs(a - b);
53         return (abs(a - c) <= dist && abs(b - c) <= dist) ||
54             (abs(a - d) <= dist && abs(b - d) <= dist);
55     }
56     return orientation(a, b, c) * orientation(a, b, d) <= 0 &&
57         orientation(c, d, a) * orientation(c, d, b) <= 0;
58 }
59
60 // Berechnet die Schnittpunkte der Strecken a-b und c-d. Enthält entweder
61 // keinen Punkt, den einzigen Schnittpunkt oder die Endpunkte der
62 // Schnittstrecke. operator<, min, max müssen noch geschrieben werden!
63 vector<pt> lineSegmentIntersection(pt a, pt b, pt c, pt d) {
64     vector<pt> result;
65     if (orientation(a, b, c) == 0 && orientation(a, b, d) == 0 &&
66         orientation(c, d, a) == 0 && orientation(c, d, b) == 0) {
67         pt minAB = min(a, b), maxAB = max(a, b);
68         pt minCD = min(c, d), maxCD = max(c, d);
69         if (minAB < minCD && maxAB < minCD) return result;
70         if (minCD < minAB && maxCD < minAB) return result;
71         pt start = max(minAB, minCD), end = min(maxAB, maxCD);
72         result.push_back(start);
73         if (start != end) result.push_back(end);
74         return result;
75     }
76     double x1 = real(b) - real(a), y1 = imag(b) - imag(a);
77     double x2 = real(d) - real(c), y2 = imag(d) - imag(c);
78     double u1 = (-y1 * (real(a) - real(c)) + x1 * (imag(a) - imag(c))) /
79         (-x2 * y1 + x1 * y2);
80     double u2 = (x2 * (imag(a) - imag(c)) - y2 * (real(a) - real(c))) /
81         (-x2 * y1 + x1 * y2);
82     if (u1 >= 0 && u1 <= 1 && u2 >= 0 && u2 <= 1) {
83         double x = real(a) + u2 * x1, y = imag(a) + u2 * y1;
84         result.push_back(pt(x, y));
85     }
86     return result;
87 }
88
89 // Entfernung von Punkt p zur Geraden durch a-b.
90 double distToLine(pt a, pt b, pt p) {
91     return abs(cross(p - a, b - a)) / abs(b - a);
92 }
93
94 // Liegt p auf der Geraden a-b?
95 bool pointOnLine(pt a, pt b, pt p) {
96     return orientation(a, b, c) == 0;
97 }

```

```

98
99 // Liegt p auf der Strecke a-b?
100 bool pointOnLineSegment(pt a, pt b, pt p) {
101     if (orientation(a, b, p) != 0) return false;
102     return real(p) >= min(real(a), real(b)) &&
103            real(p) <= max(real(a), real(b)) &&
104            imag(p) >= min(imag(a), imag(b)) &&
105            imag(p) <= max(imag(a), imag(b));
106 }
107
108 // Entfernung von Punkt p zur Strecke a-b.
109 double distToSegment(pt a, pt b, pt p) {
110     if (a == b) return abs(p - a);
111     double segLength = abs(a - b);
112     double u = ((real(p) - real(a)) * (real(b) - real(a)) +
113                (imag(p) - imag(a)) * (imag(b) - imag(a))) /
114                (segLength * segLength);
115     pt projection(real(a) + u * (real(b) - real(a)),
116                  imag(a) + u * (imag(b) - imag(a)));
117     double projectionDist = abs(p - projection);
118     if (!pointOnLineSegment(a, b, projection)) projectionDist = 1e30;
119     return min(projectionDist, min(abs(p - a), abs(p - b)));
120 }
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
131 // Liegt d in der gleichen Ebene wie a, b, und c?
132 bool isCoplanar(pt a, pt b, pt c, pt d) {
133     return abs((b - a) * (c - a) * (d - a)) < EPSILON;
134 }
135
136 // Berechnet den Flächeninhalt eines Polygons (nicht selbstschneidend).
137 // Punkte gegen den Uhrzeigersinn: positiv, sonst negativ.
138 double areaOfPolygon(vector<pt> &polygon) { // Jeder Eckpunkt nur einmal.
139     double res = 0; int n = polygon.size();
140     for (int i = 0; i < n; i++)
141         res += real(polygon[i]) * imag(polygon[(i + 1) % n]) -
142               real(polygon[(i + 1) % n]) * imag(polygon[i]);
143     return 0.5 * res;
144 }
145
146 // Schneiden sich (p1, p2) und (p3, p4) (gegenüberliegende Ecken).
147 bool rectIntersection(pt p1, pt p2, pt p3, pt p4) {
148     double minx12=min(real(p1), real(p2)), maxx12=max(real(p1), real(p2));
149     double minx34=min(real(p3), real(p4)), maxx34=max(real(p3), real(p4));
150     double miny12=min(imag(p1), imag(p2)), maxy12=max(imag(p1), imag(p2));
151     double miny34=min(imag(p3), imag(p4)), maxy34=max(imag(p3), imag(p4));
152     return (maxx12 >= minx34) && (maxx34 >= minx12) &&

```

```

153         (maxy12 >= miny34) && (maxy34 >= miny12);
154 }
155
156 // Testet, ob ein Punkt im Polygon liegt (beliebige Polygone).
157 bool pointInPolygon(pt p, vector<pt> &polygon) { // Punkte nur einmal.
158     pt rayEnd = p + pt(1, 1000000);
159     int counter = 0, n = polygon.size();
160     for (int i = 0; i < n; i++) {
161         pt start = polygon[i], end = polygon[(i + 1) % n];
162         if (lineSegmentIntersection(p, rayEnd, start, end)) counter++;
163     }
164     return counter & 1;
165 }

```

4 Mathe

4.1 ggT, kgV, erweiterter euklidischer Algorithmus

```

1 // Laufzeiten: O(log(a) + log(b))
2 ll gcd(ll a, ll b) { return b == 0 ? a : gcd(b, a % b); }
3 ll lcm(ll a, ll b) { return a * (b / gcd(a, b)); }

```

```

1 ll extendedEuclid(ll a, ll b, ll &x, ll &y) { // a*x + b*y = ggt(a, b).
2     if (a == 0) { x = 0; y = 1; return b; }
3     ll x1, y1, d = extendedEuclid(b % a, a, x1, y1);
4     x = y1 - (b / a) * x1; y = x1;
5     return d;
6 }

```

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

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

```

1 // Laufzeit: O(log(n) + log(p))
2 ll multInv(ll n, ll p) {
3     ll x, y;
4     extendedEuclid(n, p, x, y); // Implementierung von oben.
5     x += ((x / p) + 1) * p;
6     return x % p;
7 }

```

4.2 Mod-Exponent über \mathbb{F}_p

```

1 // Laufzeit:  $O(\log(b))$ 
2 ll multMod(ll a, ll b, ll n) {
3     if(a == 0 || b == 0) return 0;
4     if(b == 1) return a % n;
5     if(b % 2 == 1) return (a + multMod(a, b-1, n)) % n;
6     else return multMod((a + a) % n, b / 2, n);
7 }
8
9 // Laufzeit:  $O(\log(b))$ 
10 ll powMod(ll a, ll b, ll n) {
11     if(b == 0) return 1;
12     if(b == 1) return a % n;
13     if(b % 2 == 1) return multMod(powMod(a, b-1, n), a, n);
14     else return powMod(multMod(a, a, n), b / 2, n);
15 }

```

die Moduli nicht teilerfremd, existiert genau dann eine Lösung, wenn $a_i \equiv a_j \pmod{\gcd(m_i, m_j)}$. 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. Berechnet das kleinste, nicht negative x,
3 // das alle Kongruenzen simultan löst. Alle Lösungen sind kongruent zum
4 // kgV der Moduli (Produkt, falls alle teilerfremd sind).
5 struct ChineseRemainder {
6     typedef __int128 lll;
7     vector<lll> lhs, rhs, mods, 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        mods.push_back(m);
24    }
25
26    // Löst das System.
27    ll solve() {
28        M = accumulate(mods.begin(), mods.end(), lll(1), multiplies<lll>());
29        inv.resize(lhs.size());
30        for (int i = 0; i < (int)lhs.size(); i++) {
31            lll x = (M / mods[i]) % mods[i];
32            inv[i] = (multInv(x, mods[i]) * (M / mods[i]));
33        }
34        return (multInv(g(lhs), M) * g(rhs)) % M;
35    }
36 };

```

4.6 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) { // Rückgabe: Anzahl der Primzahlen <= n.
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;

```

4.7 Primzahltest & Faktorisierung

```

1 bool isPrime(ll n) { // Miller Rabin Primzahltest. O(log n)
2     if (n == 2) return true;
3     if (n < 2 || n % 2 == 0) return false;
4     ll d = n - 1, j = 0;
5     while (d % 2 == 0) d >>= 1, j++;
6     for (int a = 2; a <= min((ll)37, n - 1); a++) {
7         ll v = powMod(a, d, n); // Implementierung von oben.
8         if (v == 1 || v == n - 1) continue;
9         for (int i = 1; i <= j; i++) {
10             v = multMod(v, v, n); // Implementierung von oben.
11             if (v == n - 1 || v == 1) break;
12         }
13         if (v != n - 1) return false;
14     }
15     return true;
16 }
17
18 ll rho(ll n) { // Findet Faktor < n, nicht unbedingt prim.
19     if (~n & 1) return 2;
20     ll c = rand() % n, x = rand() % n, y = x, d = 1;
21     while (d == 1) {
22         x = (multMod(x, x, n) + c) % n;
23         y = (multMod(y, y, n) + c) % n;
24         y = (multMod(y, y, n) + c) % n;
25         d = __gcd(abs(x - y), n);
26     }
27     return d == n ? rho(n) : d;
28 }
29
30 void factor(ll n, map<ll, int> &facts) {
31     if (n == 1) return;
32     if (isPrime(n)) {
33         facts[n]++;
34         return;
35     }
36     ll f = rho(n);
37     factor(n/f, facts);
38     factor(f, facts);
39 }

```

4.8 Binomialkoeffizienten

Vorberechnen, wenn häufig benötigt.

```

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

4.9 EULERSche φ -Funktion

- Zählt die relativ primen Zahlen $\leq n$.
- Multiplikativ: $\gcd(a, b) = 1 \implies \varphi(a) \cdot \varphi(b) = \varphi(ab)$
- p prim, $k \in \mathbb{N}$: $\varphi(p^k) = p^k - p^{k-1}$
- $n = p_1^{a_1} \cdot \dots \cdot p_k^{a_k}$: $\varphi(n) = n \cdot \left(1 - \frac{1}{p_1}\right) \cdot \dots \cdot \left(1 - \frac{1}{p_k}\right)$ Evtl. ist es sinnvoll obgigen Code zum Faktorisieren zu benutzen und dann diese Formel anzuwenden.
- **EULER's Theorem**: Seien a und m teilerfremd. Dann: $a^{\varphi(m)} \equiv 1 \pmod m$
Falls m prim ist, liefert das den **kleinen Satz von FERMAT**: $a^m \equiv a \pmod m$

```

1 ll phi(ll n) {
2     ll result = n;
3     for(int i = 2; i * i <= n; ++i) if(n % i == 0) {
4         while(n % i == 0) n /= i;
5         result -= result / i;
6     }
7     if(n > 1) result -= result / n;
8     return result;
9 }
```

4.10 Primitivwurzeln

- Primitivwurzel modulo n existiert genau dann wenn:
 - n ist 1, 2 oder 4, oder
 - n ist Potenz einer ungeraden Primzahl, oder
 - n ist das Doppelte einer Potenz einer ungeraden Primzahl.
- Sei g Primitivwurzel modulo n . Dann gilt:
Das kleinste k , sodass $g^k \equiv 1 \pmod n$, ist $k = \varphi(n)$.

```

1 // Ist g Primitivwurzel modulo p. Teste zufällige g, um eine zu finden.
2 bool is_primitive(ll g, ll p) {
3     map<ll, int> facs;
4     factor(p - 1, facs);
5     for (auto &f : facs)
```

```

6         if (1 == powMod(g, (p - 1) / f.first, p)) return false;
7     return true;
8 }
9
10 // Alternativ: Generator zum Finden. -1 falls keine existiert.
11 ll generator(ll p) { // Laufzeit: O(ans*log(phi(n))*log(n))
12     map<ll, int> facs;
13     factor(n, facs);
14     ll phi = phi(p), n = phi;
15
16     for (ll res = 2; res <= p; res++) {
17         bool ok = true;
18         for (auto &f : facs)
19             ok &= powMod(res, phi / f.first, p) != 1;
20         if (ok) return res;
21     }
22     return -1;
23 }
```

4.11 Diskreter Logarithmus

```

1 // Bestimmt Lösung x für a^x=b mod m.
2 ll solve(ll a, ll b, ll m) { // Laufzeit: O(sqrt(m)*log(m))
3     ll n = (ll)sqrt((double)m) + 1;
4     map<ll, ll> vals;
5     for (int i = n; i >= 1; i--) vals[powMod(a, i * n, m)] = i;
6     for (int i = 0; i <= n; i++) {
7         ll cur = (powMod(a, i, m) * b) % m;
8         if (vals.count(cur)) {
9             ll ans = vals[cur] * n - i;
10            if (ans < m) return ans;
11        }
12    }
13    return -1;
14 }
```

4.12 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 schneller.
3
4 // a.size() muss eine Zweierpotenz sein!
5 vector<cplx> fft(const vector<cplx> &a, bool inverse = 0) {
6     int logn = 1, n = a.size();
7     vector<cplx> A(n);
```

```

8  while ((l < logn) < n) logn++;
9  for (int i = 0; i < n; i++) {
10     int j = 0;
11     for (int k = 0; k < logn; k++) j = (j < 1) | ((i >> k) & 1);
12     A[j] = a[i];
13 }
14 for (int s = 2; s <= n; s <= 1) {
15     double angle = 2 * PI / s * (inverse ? -1 : 1);
16     cplx ws(cos(angle), sin(angle));
17     for (int j = 0; j < n; j += s) {
18         cplx w = 1;
19         for (int k = 0; k < s / 2; k++) {
20             cplx u = A[j + k], t = A[j + s / 2 + k];
21             A[j + k] = u + w * t;
22             A[j + s / 2 + k] = u - w * t;
23             if (inverse) A[j + k] /= 2, A[j + s / 2 + k] /= 2;
24             w *= ws;
25         }
26     }
27     return A;
28 }
29 // Polynome: a[0] = a_0, a[1] = a_1, ... und b[0] = b_0, b[1] = b_1, ...
30 // Bei Integern: Runde 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

```

4.13 3D-Kugeln

```

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

```

```

24 }

```

4.14 Kombinatorik

4.14.1 Berühmte Zahlen

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

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 $\leq k$.

4.14.2 Verschiedenes

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

4.15 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 $g(X_1) \oplus \dots \oplus g(X_k)$.

4.16 Big Integers

```

1 // Bislang keine Division. Multiplikation nach Schulmethode.
2 #define PLUS 0
3 #define MINUS 1
4 #define BASE 1000000000
5 #define EXPONET 9
6
7 struct bigint {
8     int sign;
9     vector<ll> digits;
10
11     // Initialisiert mit 0.
12     bigint(void) { sign = PLUS; }
13
14     // Initialisiert mit kleinem Wert.
15     bigint(ll value) {
16         if (value == 0) sign = PLUS;
17         else {
18             sign = value >= 0 ? PLUS : MINUS;
19             value = abs(value);
20             while (value) {
21                 digits.push_back(value % BASE);
22                 value /= BASE;
23             }
24         }
25
26     // Initialisiert mit C-String. Kann nicht mit Vorzeichen umgehen.
27     bigint(char *str, int length) {
28         int base = 1;
29         ll digit = 0;
30         for (int i = length - 1; i >= 0; i--) {
31             digit += base * (str[i] - '0');
32             if (base * 10 == BASE) {
33                 digits.push_back(digit);
34                 digit = 0;
35                 base = 1;
36             } else base *= 10;
37         }
38         if (digit != 0) digits.push_back(digit);
39         sign = PLUS;
40     }
41
42     // Löscht führende Nullen und macht -0 zu 0.
43     void trim() {
44         while (digits.size() > 0 && digits[digits.size() - 1] == 0)
45             digits.pop_back();
46         if (digits.size() == 0 && sign == MINUS) sign = PLUS;
47     }
48
49     // Gibt die Zahl aus.
50     void print() {
51         if (digits.size() == 0) { printf("0"); return; }
52         if (sign == MINUS) printf("-");
53         printf("%lld", digits[digits.size() - 1]);
54         for (int i = digits.size() - 2; i >= 0; i--) {

```



```

54     printf("%09lld", digits[i]); // Anpassen, wenn andere Basis gewählt
55     wird.
56 }
57
58 // Kleiner-oder-gleich-Vergleich.
59 bool operator<=(bigint &a, bigint &b) {
60     if (a.digits.size() == b.digits.size()) {
61         int idx = a.digits.size() - 1;
62         while (idx >= 0) {
63             if (a.digits[idx] < b.digits[idx]) return true;
64             else if (a.digits[idx] > b.digits[idx]) return false;
65             idx--;
66         }
67         return true;
68     }
69     return a.digits.size() < b.digits.size();
70 }
71
72 // Kleiner-Vergleich.
73 bool operator<(bigint &a, bigint &b) {
74     if (a.digits.size() == b.digits.size()) {
75         int idx = a.digits.size() - 1;
76         while (idx >= 0) {
77             if (a.digits[idx] < b.digits[idx]) return true;
78             else if (a.digits[idx] > b.digits[idx]) return false;
79             idx--;
80         }
81         return false;
82     }
83     return a.digits.size() < b.digits.size();
84 }
85
86 void sub(bigint *a, bigint *b, bigint *c);
87
88 // a + b = c. a, b, c dürfen gleich sein.
89 void add(bigint *a, bigint *b, bigint *c) {
90     if (a->sign == b->sign) c->sign = a->sign;
91     else {
92         if (a->sign == MINUS) {
93             a->sign ^= 1;
94             sub(b, a, c);
95             a->sign ^= 1;
96         } else {
97             b->sign ^= 1;
98             sub(a, b, c);
99             b->sign ^= 1;
100         }
101         return;
102     }
103
104     c->digits.resize(max(a->digits.size(), b->digits.size()));
105     ll carry = 0;
106     int i = 0;
107     for (; i < (int)min(a->digits.size(), b->digits.size()); i++) {
108         ll sum = carry + a->digits[i] + b->digits[i];
109         c->digits[i] = sum % BASE;
110         carry = sum / BASE;
111     }
112     if (i < (int)a->digits.size()) {
113         for (; i < (int)a->digits.size(); i++) {
114             ll sum = carry + a->digits[i];
115             c->digits[i] = sum % BASE;
116             carry = sum / BASE;
117         }
118     } else {
119         for (; i < (int)b->digits.size(); i++) {
120             ll sum = carry + b->digits[i];
121             c->digits[i] = sum % BASE;
122             carry = sum / BASE;
123         }
124     }
125     if (carry) c->digits.push_back(carry);
126
127 // a - b = c. c darf a oder b sein. a und b müssen verschieden sein.
128 void sub(bigint *a, bigint *b, bigint *c) {
129     if (a->sign == MINUS || b->sign == MINUS) {
130         b->sign ^= 1;
131         add(a, b, c);
132         b->sign ^= 1;
133         return;
134     }
135
136     if (a < b) {
137         sub(b, a, c);
138         c->sign = MINUS;
139         c->trim();
140         return;
141     }
142
143     c->digits.resize(a->digits.size());
144     ll borrow = 0;
145     int i = 0;
146     for (; i < (int)b->digits.size(); i++) {
147         ll diff = a->digits[i] - borrow - b->digits[i];
148         if (a->digits[i] > 0) borrow = 0;
149         if (diff < 0) {
150             diff += BASE;
151             borrow = 1;
152         }
153         c->digits[i] = diff % BASE;
154     }
155     for (; i < (int)a->digits.size(); i++) {
156         ll diff = a->digits[i] - borrow;
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 c->trim();
165 }
166
167 // Ziffernmultiplikation a * b = c. b und c dürfen gleich sein.
168 // a muss kleiner BASE sein.
169 void digitMul(ll a, bigint *b, bigint *c) {
170     if (a == 0) {
171         c->digits.clear();
172         c->sign = PLUS;
173         return;
174     }
175     c->digits.resize(b->digits.size());
176     ll carry = 0;
177     for (int i = 0; i < (int)b->digits.size(); i++) {
178         ll prod = carry + b->digits[i] * a;
179         c->digits[i] = prod % BASE;
180         carry = prod / BASE;
181     }
182     if (carry) c->digits.push_back(carry);
183     c->sign = (a > 0) ? b->sign : 1 ^ b->sign;
184     c->trim();
185 }
186
187 // Zifferndivision b / a = c. b und c dürfen gleich sein.
188 // a muss kleiner BASE sein.
189 void digitDiv(ll a, bigint *b, bigint *c) {
190     c->digits.resize(b->digits.size());
191     ll carry = 0;
192     for (int i = (int)b->digits.size() - 1; i >= 0; i--) {
193         ll quot = (carry * BASE + b->digits[i]) / a;
194         carry = carry * BASE + b->digits[i] - quot * a;
195         c->digits[i] = quot;
196     }
197     c->sign = b->sign ^ (a < 0);
198     c->trim();
199 }
200
201 // a * b = c. c darf weder a noch b sein. a und b dürfen gleich sein.
202 void mult(bigint *a, bigint *b, bigint *c) {
203     bigint row = *a, tmp;
204     c->digits.clear();
205     for (int i = 0; i < (int)b->digits.size(); i++) {
206         digitMul(b->digits[i], &row, &tmp);
207         add(&tmp, c, c);
208         row.digits.insert(row.digits.begin(), 0);
209     }
210     c->sign = a->sign != b->sign;
211     c->trim();
212 }
213
214 // Berechnet eine kleine Zehnerpotenz.
215 inline ll pow10(int n) {
216     ll res = 1;
217     for (int i = 0; i < n; i++) res *= 10;

```

```

218     return res;
219 }
220
221 // Berechnet eine große Zehnerpotenz.
222 void power10(ll e, bigint *out) {
223     out->digits.assign(e / EXPONET + 1, 0);
224     if (e % EXPONET)
225         out->digits[out->digits.size() - 1] = pow10(e % EXPONET);
226     else out->digits[out->digits.size() - 1] = 1;
227 }
228
229 // Nimmt eine Zahl module einer Zehnerpotenz 10^e.
230 void mod10(int e, bigint *a) {
231     int idx = e / EXPONET;
232     if ((int)a->digits.size() < idx + 1) return;
233     if (e % EXPONET) {
234         a->digits.resize(idx + 1);
235         a->digits[idx] %= pow10(e % EXPONET);
236     } else {
237         a->digits.resize(idx);
238     }
239     a->trim();
240 }

```

5 Strings

5.1 KNUTH-MORRIS-PRATT-Algorithmus

```

1 // Laufzeit: O(n + m), n = #Text, m = #Pattern
2 vector<int> kmpPreprocessing(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> kmpSearch(string &s, string &sub) {
15     vector<int> pre = kmpPreprocessing(sub), result;
16     int i = 0, j = 0;
17     while (i < (int)s.length()) {
18         while (j >= 0 && s[i] != sub[j]) j = pre[j];
19         i++; j++;
20         if (j == (int)sub.length()) {
21             result.push_back(i - j);
22             j = pre[j];
23         }
24     }
25     return result;

```

25 }

5.2 AHO-CORASICK-Automat

```

1 // Laufzeit:  $O(n + m + z)$ ,  $n = \# \text{Text}$ ,  $m = \text{Summe } \# \text{Pattern}$ ,  $z = \# \text{Matches}$ 
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.
7 // DANACH: Wenn patterns-Vektor nicht leer ist: Hier enden alle
8 // enthaltenen Patterns.
9 // ACHTUNG: Die Zahlenwerte der auftretenden Buchstaben müssen
10 // zusammenhängend sein und bei 0 beginnen!
11 struct vertex {
12     vertex *next[ALPHABET_SIZE], *failure;
13     char character;
14     vector<int> patterns; // Indizes der Patterns, die hier enden.
15     vertex() { for (int i = 0; i < ALPHABET_SIZE; i++) next[i] = NULL; }
16 };
17
18 void addString(vertex *v, string &pattern, int patternIdx) {
19     for (int i = 0; i < (int)pattern.length(); i++) {
20         if (!v->next[(int)pattern[i]]) {
21             vertex *w = new vertex();
22             w->character = pattern[i];
23             v->next[(int)pattern[i]] = w;
24         }
25         v = v->next[(int)pattern[i]];
26     }
27     v->patterns.push_back(patternIdx);
28 }
29
30 void finishAutomaton(vertex *v) {
31     for (int i = 0; i < ALPHABET_SIZE; i++)
32         if (!v->next[i]) v->next[i] = v;
33
34     queue<vertex*> q;
35     for (int i = 0; i < ALPHABET_SIZE; i++) {
36         if (v->next[i] != v) {
37             v->next[i]->failure = v;
38             q.push(v->next[i]);
39         }
40     }
41     while (!q.empty()) {
42         vertex *r = q.front(); q.pop();
43         for (int i = 0; i < ALPHABET_SIZE; i++) {
44             if (r->next[i]) {
45                 q.push(r->next[i]);
46                 vertex *f = r->failure;
47                 while (!f->next[i]) f = f->failure;
48                 r->next[i]->failure = f->next[i];
49                 for (int j = 0; j < (int)f->next[i]->patterns.size(); j++) {
50                     r->next[i]->patterns.push_back(f->next[i]->patterns[j]);

```

```

50 }
51
52 vertex* go(vertex *v, char c) {
53     if (v->next[(int)c]) return v->next[(int)c];
54     else return go(v->failure, c);
55 }

```

5.3 Trie

```

1 struct node {
2     node *(e)[26]; // Implementierung für Kleinbuchstaben.
3     int c = 0; // Anzahl der Wörter, die an diesem node enden.
4     node() { for (int i = 0; i < 26; i++) e[i] = NULL; }
5 };
6
7 void insert(node *root, string &txt, int s) { // Laufzeit:  $O(|\text{txt}|)$ 
8     if (s == (int)txt.size()) root->c++;
9     else {
10         int idx = (int)(txt[s] - 'a');
11         if (root->e[idx] == NULL) root->e[idx] = new node();
12         insert(root->e[idx], txt, s+1);
13     }
14 }
15
16 int contains(node *root, string &txt, int s) { // Laufzeit:  $O(|\text{txt}|)$ 
17     if (s == txt.size()) return root->c;
18     int idx = (int)(txt[s] - 'a');
19     if (root->e[idx] != NULL) return contains(root->e[idx], txt, s + 1);
20     else return 0;

```

5.4 Suffix-Array

```

1 struct SuffixArray { // MAX_LG = ceil(log2(MAX_N))
2     static const int MAX_N = 100010, MAX_LG = 17;
3     pair<pair<int, int>, int> L[MAX_N];
4     int P[MAX_LG + 1][MAX_N], n, step, count;
5     int suffixArray[MAX_N], lcpArray[MAX_N];
6
7     SuffixArray(const string &s) : n(s.size()) { // Laufzeit:  $O(n \cdot \log^2(n))$ 
8         for (int i = 0; i < n; i++) P[0][i] = s[i];
9         suffixArray[0] = 0; // Falls n == 1.
10         for (step = 1, count = 1; count < n; step++, count <= 1) {
11             for (int i = 0; i < n; i++) L[i] =
12                 {{P[step-1][i], i+count < n ? P[step-1][i+count] : -1}, i};
13             sort(L, L + n);
14             for (int i = 0; i < n; i++) P[step][L[i].second] = i > 0 &&
15                 L[i].first == L[i-1].first ? P[step][L[i-1].second] : i;
16         }
17         for (int i = 0; i < n; i++) suffixArray[i] = L[i].second;
18         for (int i = 1; i < n; i++)

```

```

19     lcpArray[i] = lcp(suffixArray[i - 1], suffixArray[i]);
20 }
21
22 // x und y sind Indizes im String, nicht im Suffixarray.
23 int lcp(int x, int y) { // Laufzeit: O(log(n))
24     int k, ret = 0;
25     if (x == y) return n - x;
26     for (k = step - 1; k >= 0 && x < n && y < n; k--)
27         if (P[k][x] == P[k][y])
28             x += 1 << k, y += 1 << k, ret += 1 << k;
29     return ret;
30 }
31 };

```

5.5 Suffix-Automaton

```

1 #define ALPHABET_SIZE 26
2 struct SuffixAutomaton {
3     struct State {
4         int length; int link; int next[ALPHABET_SIZE];
5         State() { memset(next, 0, sizeof(next)); }
6     };
7     static const int MAX_N = 100000; // Maximale Länge des Strings.
8     State states[2 * MAX_N];
9     int size, last;
10
11     SuffixAutomaton(string &s) { // Laufzeit: O(|s|)
12         size = 1; last = 0;
13         states[0].length = 0;
14         states[0].link = -1;
15         for (auto c : s) extend(c);
16     }
17
18     void extend(char c) {
19         c -= 'a'; // Werte von c müssen bei 0 beginnen.
20         int current = size++;
21         states[current].length = states[last].length + 1;
22         int pos = last;
23         while (pos != -1 && !states[pos].next[(int)c]) {
24             states[pos].next[(int)c] = current;
25             pos = states[pos].link;
26         }
27         if (pos == -1) states[current].link = 0;
28         else {
29             int q = states[pos].next[(int)c];
30             if (states[pos].length + 1 == states[q].length) {
31                 states[current].link = q;
32             } else {
33                 int clone = size++;
34                 states[clone].length = states[pos].length + 1;
35                 states[clone].link = states[q].link;
36                 memcpy(states[clone].next, states[q].next,
37                     sizeof(states[q].next));

```

```

38         while (pos != -1 && states[pos].next[(int)c] == q) {
39             states[pos].next[(int)c] = clone;
40             pos = states[pos].link;
41         }
42         states[q].link = states[current].link = clone;
43     }
44     last = current;
45 }
46
47 // Paar mit Startposition und Länge des LCS. Index in Parameter s.
48 ii longestCommonSubstring(string &s) { // Laufzeit: O(|s|)
49     int v = 0, l = 0, best = 0, bestpos = 0;
50     for (int i = 0; i < (int)s.size(); i++) {
51         int c = s[i] - 'a';
52         while (v && !states[v].next[c]) {
53             v = states[v].link;
54             l = states[v].length;
55         }
56         if (states[v].next[c]) { v = states[v].next[c]; l++; }
57         if (l > best) { best = l; bestpos = i; }
58     }
59     return ii(bestpos - best + 1, best);
60 }
61
62 // Berechnet die Terminale des Automaten.
63 vector<int> calculateTerminals() {
64     vector<int> terminals;
65     int pos = last;
66     while (pos != -1) {
67         terminals.push_back(pos);
68         pos = states[pos].link;
69     }
70     return terminals;
71 }
72 };

```

- **Ist w Substring von s ?** Baue Automaten für s und wende ihn auf w an. Wenn alle Übergänge vorhanden sind, ist w Substring von s .
- **Ist w Suffix von s ?** Wie oben. Überprüfe am Ende, ob aktueller Zustand ein Terminal ist.
- **Anzahl verschiedener Substrings.** Jeder Pfad im Automaten entspricht einem Substring. Für einen Knoten ist die Anzahl der ausgehenden Pfade gleich der Summe über die Anzahlen der Kindknoten plus 1. Der letzte Summand ist der Pfad, der in diesem Knoten endet.
- **Wie oft taucht w in s auf?** Sei p der Zustand nach Abarbeitung von w . Lösung ist Anzahl der Pfade, die in p starten und in einem Terminal enden. Diese Zahl lässt sich wie oben rekursiv berechnen. Bei jedem Knoten darf nur dann plus 1 gerechnet werden, wenn es ein Terminal ist.

5.6 Longest Common Subsequence

```

1 // Laufzeit: O(|a|*|b|)
2 string lcsm(string &a, string &b) {
3     int m[a.length() + 1][b.length() + 1], x=0, y=0;
4     memset(m, 0, sizeof(m));
5     for(int y = a.length() - 1; y >= 0; y--) {
6         for(int x = b.length() - 1; x >= 0; x--) {
7             if(a[y] == b[x]) m[y][x] = 1 + m[y+1][x+1];
8             else m[y][x] = max(m[y+1][x], m[y][x+1]);
9         } // Für die Länge: 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 }

```

5.7 Rolling Hash

```

1 ll q = 31; // Größer als Alphabetgröße. q=31,53,311
2 struct Hasher {
3     string s;
4     ll mod;
5     vector<ll> power, pref;
6     Hasher(const string& s, ll mod) : s(s), mod(mod) {
7         power.push_back(1);
8         for (int i = 1; i < (int)s.size(); i++)
9             power.push_back(power.back() * q % mod);
10        pref.push_back(0);
11        for (int i = 0; i < (int)s.size(); i++)
12            pref.push_back((pref.back() * q % mod + s[i]) % mod);
13    }
14
15    // Berechnet hash(s[l..r]). l,r inklusive.
16    ll hash(int l, int r) {
17        return (pref[r+1] - power[r-l+1] * pref[l] % mod + mod) % mod;
18    }
19 };

```

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(); // Die nächste Zeile.
3 int num = in.nextInt(); // Das nächste Token als int.
4 double num2 = in.nextDouble(); // Das nächste Token als double.

```

- Ausgabe:

```

1 // In StringBuilder schreiben und auf einmal ausgeben ist 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),
3     divide(BigInteger val), subtract(BigInteger val)
4
5 // Berechnet this^e.
6 BigInteger pow(BigInteger e)
7
8 // Bit-Operationen.
9 BigInteger and(BigInteger val), or(BigInteger val), xor(BigInteger val),
10    not(), shiftLeft(int n), shiftRight(int n)
11
12 // Berechnet den ggT von abs(this) und abs(val).
13 BigInteger gcd(BigInteger val)
14
15 // Berechnet this mod m, this^-1 mod m, this^e mod m.
16 BigInteger mod(BigInteger m), modInverse(BigInteger m),
17    modPow(BigInteger e, BigInteger m)
18
19 // Berechnet die nächste Zahl, die größer und wahrscheinlich prim ist.
20 BigInteger nextProbablePrime()
21
22 // Berechnet int/long/float/double-Wert.
23 // Ist die Zahl zu groß werden die niedrigsten Bits konvertiert.
24 int intValue(), long longValue(),
25 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 // Zerlegt s anhand aller Zeichen in delim.
2 vector<string> split(string &s, string delim) {
3     vector<string> result; char *token;
4     token = strtok((char*)s.c_str(), (char*)delim.c_str());
5     while (token != NULL) {
6         result.push_back(string(token));
7         token = strtok(NULL, (char*)delim.c_str());
8     }
9     return result;
10 }

```

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) { // Der letzte Überlebende, 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) { // Der letzte Überlebende, 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.
- **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 ∞ 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 7)

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
```