

# Team Contest Reference

ChaosKITs  
Karlsruhe Institute of Technology

23. Dezember 2014

## Inhaltsverzeichnis

|          |   |           |          |  |           |
|----------|---|-----------|----------|--|-----------|
| <b>1</b> | <b>Datenstrukturen</b>  | <b>2</b>  | 4.1.1    | Multiplikatives Inverses von $x$ in $\mathbb{Z}/n\mathbb{Z}$ . . . . . | 12        |
| 1.1      | Union-Find . . . . .  | 2         | 4.2      | Primzahlsieb von Eratosthenes . . . . .                                | 13        |
| 1.2      | Segmentbaum . . . . .   | 2         | 4.2.1    | Faktorisierung . . . . .   | 13        |
| 1.3      | Range Minimum Query . . . . .                                 | 2         | 4.2.2    | Mod-Exponent über $\mathbb{F}_p$ . . . . .                             | 13        |
| 1.4      | STL-Tree . . . . .  | 3         | 4.3      | LGS über $\mathbb{F}_p$ . . . . .                                      | 13        |
| <b>2</b> | <b>Graphen</b>  | <b>3</b>  | 4.4      | Binomialkoeffizienten . . . . .  | 14        |
| 2.1      | Minimale Spannbäume . . . . .                                 | 3         | 4.5      | Satz von SPRAGUE-GRUNDY . . . . .                                      | 14        |
| 2.2      | Kürzeste Wege . . . . .                                       | 3         | 4.6      | Maximales Teilfeld . . . . .   | 14        |
| 2.2.1    | Algorithmus von DIJKSTRA . . . . .                            | 3         | 4.7      | Kombinatorik . . . . .   | 15        |
| 2.2.2    | BELLMANN-FORD-Algorithmus . . . . .                           | 4         | 4.7.1    | Berühmte Zahlen . . . . .  | 15        |
| 2.2.3    | FLOYD-WARSHALL-Algorithmus . . . . .                          | 4         | 4.7.2    | Verschiedenes . . . . .  | 16        |
| 2.3      | Strongly Connected Components (TARJANS-Algorithmus) . . . . . | 4         | <b>5</b> | <b>Strings</b>   | <b>16</b> |
| 2.4      | Artikulationspunkte und Brücken . . . . .                     | 5         | 5.1      | KNUTH-MORRIS-PRATT-Algorithmus . . . . .                               | 16        |
| 2.5      | Eulertouren . . . . .   | 6         | 5.2      | LEVENSHTEIN-Distanz . . . . .  | 16        |
| 2.6      | Lowest Common Ancestor . . . . .                              | 7         | 5.3      | Trie . . . . .   | 17        |
| 2.7      | Max-Flow (EDMONDS-KARP-Algorithmus) . . . . .                 | 7         | 5.4      | Suffix-Array . . . . .   | 17        |
| 2.7.1    | Maximum Edge Disjoint Paths . . . . .                         | 8         | 5.5      | Longest Common Substring . . . . .                                     | 18        |
| 2.7.2    | Maximum Independent Paths . . . . .                           | 8         | 5.6      | Longest Common Subsequence . . . . .                                   | 18        |
| 2.8      | Maximal Cardinality Bipartite Matching . . . . .              | 8         | <b>6</b> | <b>Java</b>  | <b>18</b> |
| 2.9      | TSP . . . . .   | 8         | 6.1      | Introduction . . . . .   | 18        |
| 2.10     | Bitonic TSP . . . . .   | 9         | 6.2      | BigInteger . . . . .   | 19        |
| <b>3</b> | <b>Geometrie</b>  | <b>9</b>  | <b>7</b> | <b>Sonstiges</b>   | <b>19</b> |
| 3.1      | Closest Pair . . . . .  | 9         | 7.1      | 2-SAT . . . . .  | 19        |
| 3.2      | Geraden . . . . .   | 10        | 7.2      | Sortieren in Linearzeit . . . . .                                      | 19        |
| 3.3      | Konvexe Hülle . . . . .                                       | 10        | 7.2.1    | Bucketsort . . . . .   | 19        |
| 3.4      | Formeln - std::complex . . . . .                              | 11        | 7.2.2    | LSD-Radixsort . . . . .  | 20        |
| <b>4</b> | <b>Mathe</b>  | <b>12</b> | 7.3      | Bit Operations . . . . .   | 20        |
| 4.1      | ggT, kgV, erweiterter euklidischer Algorithmus . . . . .      | 12        | 7.4      | Roman-Literal-Converting . . . . .                                     | 20        |
|          |   |           | 7.5      | Josephus-Problem . . . . .   | 21        |
|          |   |           | 7.6      | Gemischtes . . . . .   | 21        |
|          |   |           | <b>8</b> | <b>Convenience-Methoden</b>  | <b>22</b> |
|          |   |           | 8.1      | Zeileneingabe . . . . .  | 22        |

# 1 Datenstrukturen

## 1.1 Union-Find

```

1 vector<int> parent, rank2; //manche Compiler verbieten Variable mit Namen rank
2
3 int findSet(int n) { //Pfadkompression
4     if (parent[n] != n) parent[n] = findSet(parent[n]);
5     return parent[n];
6 }
7
8 void linkSets(int a, int b) { //union by rank
9     if (rank2[a] < rank2[b]) parent[a] = b;
10    else if (rank2[b] < rank2[a]) parent[b] = a;
11    else {
12        parent[a] = b;
13        rank2[b]++;
14    }
15 }
16
17 void unionSets(int a, int b) {
18     if (findSet(a) != findSet(b)) linkSets(findSet(a), findSet(b));
19 }

```

## 1.2 Segmentbaum

```

1 int a[MAX_N], m[4 * MAX_N];
2
3 int query(int x, int y, int k = 0, int X = 0, int Y = MAX_N - 1) {
4     if (x <= X && Y <= y) return m[k];
5     if (y < X || Y < x) return -1000000000; //ein "neutrales" Element
6     int M = (X + Y) / 2;
7     return max(query(x, y, 2 * k + 1, X, M), query(x, y, 2 * k + 2, M + 1, Y));
8 }
9
10 void update(int i, int v, int k = 0, int X = 0, int Y = MAX_N - 1) {
11     if (i < X || Y < i) return;
12     if (X == Y) {
13         m[k] = v;
14         a[i] = v;
15         return;
16     }
17     int M = (X + Y) / 2;
18     update(i, v, 2 * k + 1, X, M);
19     update(i, v, 2 * k + 2, M + 1, Y);
20     m[k] = max(m[2 * k + 1], m[2 * k + 2]);
21 }
22
23 void init(int k = 0, int X = 0, int Y = MAX_N - 1) {
24     if (X == Y) {
25         m[k] = a[X];
26         return;
27     }
28     int M = (X + Y) / 2;
29     init(2 * k + 1, X, M);
30     init(2 * k + 2, M + 1, Y);
31     m[k] = max(m[2 * k + 1], m[2 * k + 2]);
32 }

```

## 1.3 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 void initRMQ() {

```

```

5   for(int i = 0, s = 1, ss = 1; s <= RMQ_SIZE; ss=s, s*=2, i++) {
6       for(int l = 0; l + s <= RMQ_SIZE; l++) {
7           if(i == 0) rmq[0][l] = 1;
8           else {
9               int r = l + ss;
10              rmq[i][l] = (data[rmq[i-1][l]] <= data[rmq[i-1][r]] ? rmq[i-1][l] : rmq[i-1][r]);
11          }
12      }
13  }
14 }
15 //returns index of minimum! [a, b)
16 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 }

```

## 1.4 STL-Tree

```

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

```

## 2 Graphen

### 2.1 Minimale Spannbäume

Benutze Algorithmus von KRUSKAL oder Algorithmus von PRIM.

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

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

### 2.2 Kürzeste Wege

#### 2.2.1 Algorithmus von DIJKSTRA

Kürzeste Pfade in Graphen ohne negative Kanten.

```

1 priority_queue<ii, vector<ii>, greater<ii> > pq;
2 vector<int> dist;
3 dist.assign(NUM_VERTICES, INF);
4 dist[0] = 0;
5 pq.push(ii(0, 0));
6
7 while (!pq.empty()) {
8     ii front = pq.top(); pq.pop();
9     int curNode = front.second, curDist = front.first;
10
11     if (curDist > dist[curNode]) continue;
12
13     for (int i = 0; i < (int)adjlist[curNode].size(); i++) {
14         int nextNode = adjlist[curNode][i].first, nextDist = curDist + adjlist[curNode][i].second;
15     }

```

```

16         if (nextDist < dist[nextNode]) {
17             dist[nextNode] = nextDist; pq.push(ii(nextDist, nextNode));
18         }
19     }
20 }

```

## 2.2.2 BELLMANN-FORD-Algorithmus

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

```

1 //n = number of vertices, edges is vector of edges
2 dist.assign(n, INF); dist[0] = 0;
3 parent.assign(n, -1);
4 for (i = 0; i < n - 1; i++) {
5     for (j = 0; j < (int)edges.size(); j++) {
6         if (dist[edges[j].from] + edges[j].cost < dist[edges[j].to]) {
7             dist[edges[j].to] = dist[edges[j].from] + edges[j].cost;
8             parent[edges[j].to] = edges[j].from;
9         }
10    }
11 }
12 //now dist and parent are correct shortest paths
13 //next lines check for negative cycles
14 for (j = 0; j < (int)edges.size(); j++) {
15     if (dist[edges[j].from] + edges[j].cost < dist[edges[j].to]) {
16         //NEGATIVE CYCLE found
17     }
18 }

```

## 2.2.3 FLOYD-WARSHALL-Algorithmus

Alle kürzesten Pfade im Graphen.

```

1 //initialize adjmat, adjmat[i][i] = 0, adjmat[i][j] = INF if no edge is between i and j, length otherwise
2 for (k = 0; k < MAX_V; k++) {
3     for (i = 0; i < MAX_V; i++) {
4         for (j = 0; j < MAX_V; j++) {
5             if (adjmat[i][k] + adjmat[k][j] < adjmat[i][j]) adjmat[i][j] = adjmat[i][k] + adjmat[k][j];
6         }
7     }
8 }

```

FLOYD-WARSHALL findet auch negative Kreise. Es existiert genau dann ein negativer Kreis, wenn  $\text{dist}[i][i] < 0$  ist.

## 2.3 Strongly Connected Components (TARJANS-Algorithmus)

```

1 int counter, sccCounter, n; //n == number of vertices
2 vector<bool> visited, inStack;
3 vector< vector<int> > adjlist;
4 vector<int> d, low, sccs;
5 stack<int> s;
6
7 void visit(int v) {
8     visited[v] = true;
9     d[v] = counter;
10    low[v] = counter;
11    counter++;
12    inStack[v] = true;
13    s.push(v);
14
15    for (int i = 0; i < (int)adjlist[v].size(); i++) {
16        int u = adjlist[v][i];
17        if (!visited[u]) {
18            visit(u);
19            low[v] = min(low[v], low[u]);
20        } else if (inStack[u]) {
21            low[v] = min(low[v], low[u]);

```

```

22     }
23 }
24
25 if (d[v] == low[v]) {
26     int u;
27     do {
28         u = s.top();
29         s.pop();
30         inStack[u] = false;
31         sccs[u] = sccCounter;
32     } while(u != v);
33     sccCounter++;
34 }
35 }
36
37 void scc() {
38     //read adjlist
39
40     visited.clear(); visited.assign(n, false);
41     d.clear(); d.resize(n);
42     low.clear(); low.resize(n);
43     inStack.clear(); inStack.assign(n, false);
44     sccs.clear(); sccs.resize(n);
45
46     counter = 0;
47     sccCounter = 0;
48     for (i = 0; i < n; i++) {
49         if (!visited[i]) {
50             visit(i);
51         }
52     }
53     //sccs has the component for each vertex
54 }

```

## 2.4 Artikulationspunkte und Brücken

```

1 vector< vector<int> > adjlist;
2 vector<int> low;
3 vector<int> d;
4 vector<bool> isArtPoint;
5 vector< vector<int> > bridges; //nur fuer Bruecken
6 int counter = 0;
7
8 void visit(int v, int parent) {
9     d[v] = low[v] = ++counter;
10    int numVisits = 0, maxlow = 0;
11
12    for (vector<int>::iterator vit = adjlist[v].begin(); vit != adjlist[v].end(); vit++) {
13        if (d[*vit] == 0) {
14            numVisits++;
15            visit(*vit, v);
16            if (low[*vit] > maxlow) {
17                maxlow = low[*vit];
18            }
19
20            if (low[*vit] > d[v]) { //nur fuer Bruecken
21                bridges[v].push_back(*vit); bridges[*vit].push_back(v);
22            }
23
24            low[v] = min(low[v], low[*vit]);
25        } else {
26            if (d[*vit] < low[v]) {
27                low[v] = d[*vit];
28            }
29        }
30    }
31
32    if (parent == -1) {
33        if (numVisits > 1) isArtPoint[v] = true;

```

```

34     } else {
35         if (maxlow >= d[v]) isArtPoint[v] = true;
36     }
37 }
38
39 void findArticulationPoints() {
40     low.clear(); low.resize(adjlist.size());
41     d.clear(); d.assign(adjlist.size(), 0);
42     isArtPoint.clear(); isArtPoint.assign(adjlist.size(), false);
43     bridges.clear(); isBridge.resize(adjlist.size()); //nur fuer Bruecken
44     for (int v = 0; v < (int)adjlist.size(); v++) {
45         if (d[v] == 0) visit(v, -1);
46     }
47 }

```

## 2.5 Eulertouren

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

```

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

```

Abbildung 1: Idee für Eulerzyklen

```

1 vector< vector<int> > adjlist;
2 vector< vector<int> > otherIdx;
3 vector<int> cycle;
4 vector<int> validIdx;
5
6 void swapEdges(int n, int a, int b) { // Vertauscht Kanten mit Indizes a und b von Knoten n.
7     int neighA = adjlist[n][a];
8     int neighB = adjlist[n][b];
9     int idxNeighA = otherIdx[n][a];
10    int idxNeighB = otherIdx[n][b];
11    swap(adjlist[n][a], adjlist[n][b]);
12    swap(otherIdx[n][a], otherIdx[n][b]);
13    otherIdx[neighA][idxNeighA] = b;
14    otherIdx[neighB][idxNeighB] = a;
15 }
16
17 void removeEdge(int n, int i) { // Entfernt Kante i von Knoten n (und die zugehoerige Rueckwaertskante).
18     int other = adjlist[n][i];
19     if (other == n) { //Schlingen
20         validIdx[n]++;
21         return;
22     }
23     int otherIndex = otherIdx[n][i];
24     validIdx[n]++;
25     if (otherIndex != validIdx[other]) {
26         swapEdges(other, otherIndex, validIdx[other]);
27     }
28     validIdx[other]++;
29 }
30
31 //findet Eulerzyklus an Knoten n startend

```

```

32 //teste vorher, dass Graph zusammenhaengend ist! (isolierte Punkte sind ok)
33 //teste vorher, ob Eulerzyklus ueberhaupt existiert!
34 void euler(int n) {
35     while (validIdx[n] < (int)adjlist[n].size()) {
36         int nn = adjlist[n][validIdx[n]];
37         removeEdge(n, validIdx[n]);
38         euler(nn);
39     }
40     cycle.push_back(n); //Zyklus am Ende in cycle
41 }

```

## 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 void initLCA(int gi, int d, int &c) {
6     visited[c] = gi, depth[c] = d, first[gi] = min(c, first[gi]), c++;
7     for(int gn : graph[gi]) {
8         initLCA(gn, d+1, c);
9         visited[c] = gi, depth[c] = d, c++;
10    }
11 }
12 //[a, b]
13 int getLCA(int a, int b) {
14     return visited[queryRMQ(min(first[a], first[b]), max(first[a], first[b]))];
15 }
16 //=> int c = 0; initLCA(0,0,c); initRMQ(); done!

```

## 2.7 Max-Flow (EDMONDS-KARP-Algorithmus)

```

1 int s, t, f; //source, target, single flow
2 int res[MAX_V][MAX_V]; //adj-matrix
3 vector< vector<int> > adjList;
4 int p[MAX_V]; //bfs spanning tree
5
6 void augment(int v, int minEdge) {
7     if (v == s) { f = minEdge; return; }
8     else if (p[v] != -1) {
9         augment(p[v], min(minEdge, res[p[v]][v]));
10        res[p[v]][v] -= f; res[v][p[v]] += f;
11    }
12 }
13 int maxFlow() { //first initialize res, adjList, s and t
14     int mf = 0;
15     while (true) {
16         f = 0;
17         bitset<MAX_V> vis; vis[s] = true;
18         queue<int> q; q.push(s);
19         memset(p, -1, sizeof(p));
20         while (!q.empty()) { //BFS
21             int u = q.front(); q.pop();
22             if (u == t) break;
23             for (int j = 0; j < (int)adjList[u].size(); j++) {
24                 int v = adjList[u][j];
25                 if (res[u][v] > 0 && !vis[v]) {
26                     vis[v] = true; q.push(v); p[v] = u;
27                 }
28             }
29             augment(t, INF); //add found path to max flow
30             if (f == 0) break;
31             mf += f;
32         }
33     }
34     return mf;
35 }

```

### 2.7.1 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.

### 2.7.2 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.

## 2.8 Maximal Cardinality Bipartite Matching

```

1 vector< vector<int> > adjlist;
2 vector<int> pairs; //for every node, stores the matching node on the other side or -1
3 vector<bool> visited;
4
5 bool dfs(int i) {
6     if (visited[i]) return false;
7     visited[i] = true;
8     for (vector<int>::iterator vit = adjlist[i].begin(); vit != adjlist[i].end(); vit++) {
9         if (pairs[*vit] < 0 || dfs(pairs[*vit])) {
10             pairs[*vit] = i; pairs[i] = *vit; return true;
11         }
12     }
13     return false;
14 }
15
16 int kuhn(int n, int m) { // n = nodes on left side (numbered 0..n-1), m = nodes on the right side
17     pairs.assign(n + m, -1);
18     int ans = 0;
19     for (int i = 0; i < n; i++) {
20         visited.assign(n + m, false);
21         ans += dfs(i);
22     }
23     return ans; //size of the MCBM
24 }
```

## 2.9 TSP

```

1 //nodes[0] has to be the start and end node.
2 vector<vector<int>> dist;
3 vector<int> TSP() {
4     int n = dist.size(), m = 1 << n;
5     vector<vector<ii>> dp(n, vector<ii>(m, ii(MAX_N, -1)));
6
7     for(int c = 0; c < n; c++) dp[c][m-1].first = dist[c][0], dp[c][m-1].second = 0;
8
9     for(int v = m - 2; v >= 0; v--) {
10         for(int c = n - 1; c >= 0; c--) {
11             for(int g = 0; g < n; g++) {
12                 if(g != c && (((1 << g) & v) == 0)) {
13                     if((dp[g][(v | (1 << g))].first + dist[c][g]) < dp[c][v].first) {
14                         dp[c][v].first = dp[g][(v | (1 << g))].first + dist[c][g];
15                         dp[c][v].second = g;
16                     }
17                 }
18             }
19         }
20     }
21
22     vector<int> res; res.push_back(0); int v = 0;
23     while(res.back() != 0 || res.size() == 1) {
24         res.push_back(dp[res.back()][(v |= (1 << res.back()))].second);
25     }
```



```

25     }
26
27     return res;
28 }

```

## 2.10 Bitonic TSP

```

1  vector< vector<double> > dp; //initialize with -1
2  vector< vector<double> > dist; //initialize with all dists between points
3  vector<int> lr, rl; //left-to-right and right-to-left paths
4  int n; //number of points
5  double get(int p1, int p2) { //call get(0, 0) to get length of shortest bitonic route
6      int v = max(p1, p2) + 1;
7      if (v == n - 1) return dist[p1][v] + dist[v][p2];
8      if (dp[p1][p2] > -0.5) return dp[p1][p2];
9      double tryLR = dist[p1][v] + get(v, p2), tryRL = dist[v][p2] + get(p1, v);
10     if (tryLR < tryRL) lr.push_back(v); //reconstructs the path, pushes v to rl if the choice does not matter
11     else rl.push_back(v); //change this if needed
12     return min(tryLR, tryRL);
13 }

```

## 3 Geometrie

### 3.1 Closest Pair

```

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

```

## 3.2 Geraden

```

1 struct pt { //complex<double> does not work here, becuae we need to set pt.x and pt.y
2     double x, y;
3     pt() {};
4     pt(double x, double y) : x(x), y(y) {};
5 };
6
7 struct line {
8     double a, b, c; //a*x+b*y+c, b=0 <=> vertical line, b=1 <=> otherwise
9 };
10
11 line pointsToLine(pt p1, pt p2) {
12     line l;
13     if (fabs(p1.x - p2.x) < EPSILON) {
14         l.a = 1; l.b = 0.0; l.c = -p1.x;
15     } else {
16         l.a = -(double)(p1.y - p2.y) / (p1.x - p2.x);
17         l.b = 1.0;
18         l.c = -(double)(l.a * p1.x) - p1.y;
19     }
20     return l;
21 }
22
23 bool areParallel(line l1, line l2) {
24     return (fabs(l1.a - l2.a) < EPSILON) && (fabs(l1.b - l2.b) < EPSILON);
25 }
26
27 bool areSame(line l1, line l2) {
28     return areParallel(l1, l2) && (fabs(l1.c - l2.c) < EPSILON);
29 }
30
31 bool areIntersect(line l1, line l2, pt &p) {
32     if (areParallel(l1, l2)) return false;
33     p.x = (l2.b * l1.c - l1.b * l2.c) / (l2.a * l1.b - l1.a * l2.b);
34     if (fabs(l1.b) > EPSILON) p.y = -(l1.a * p.x + l1.c);
35     else p.y = -(l2.a * p.x + l2.c);
36     return true;
37 }

```

## 3.3 Konvexe Hülle

```

1 struct point {
2     double x, y;
3     point(){} point(double x, double y) : x(x), y(y) {}
4     bool operator <(const point &p) const {
5         return x < p.x || (x == p.x && y < p.y);
6     }
7 };
8
9 // 2D cross product.
10 // Return a positive value, if OAB makes a counter-clockwise turn,
11 // negative for clockwise turn, and zero if the points are collinear.
12 double cross(const point &O, const point &A, const point &B){
13     double d = (A.x - O.x) * (B.y - O.y) - (A.y - O.y) * (B.x - O.x);
14     if (fabs(d) < 1e-9) return 0.0;
15     return d;
16 }
17
18 // Returns a list of points on the convex hull in counter-clockwise order.
19 // Colinear points are not in the convex hull, if you want colinear points in the hull remove "=" in the CCW-
    Test
20 // Note: the last point in the returned list is the same as the first one.
21 vector<point> convexHull(vector<point> P){
22     int n = P.size(), k = 0;
23     vector<point> H(2*n);
24
25     // Sort points lexicographically

```

```

26 sort(P.begin(), P.end());
27
28 // Build lower hull
29 for (int i = 0; i < n; i++) {
30     while (k >= 2 && cross(H[k-2], H[k-1], P[i]) <= 0.0) k--;
31     H[k++] = P[i];
32 }
33
34 // Build upper hull
35 for (int i = n-2, t = k+1; i >= 0; i--) {
36     while (k >= t && cross(H[k-2], H[k-1], P[i]) <= 0.0) k--;
37     H[k++] = P[i];
38 }
39
40 H.resize(k);
41 return H;
42 }

```

### 3.4 Formeln - std::complex

```

1 //komplexe Zahlen als Darstellung fuer Punkte
2 typedef pt complex<double>;
3 //Winkel zwischen Punkt und x-Achse in [0, 2 * PI), Winkel zwischen a und b
4 double angle = arg (a), angle_a_b = arg (a - b);
5 //Punkt rotiert um Winkel theta
6 pt a_rotated = a * exp (pt (0, theta));
7 //Mittelpunkt des Dreiecks abc
8 pt centroid = (a + b + c) / 3;
9 //Skalarprodukt
10 double dot(pt a, pt b) {
11     return real(conj(a) * b);
12 }
13 //Kreuzprodukt, 0, falls kollinear
14 double cross(pt a, pt b) {
15     return imag(conj(a) * b);
16 }
17 //wenn Eckpunkte bekannt
18 double areaOfTriangle(pt a, pt b, pt c) {
19     return abs(cross(b - a, c - a)) / 2.0;
20 }
21 //wenn Seitenlaengen bekannt
22 double areaOfTriangle(double a, double b, double c) {
23     double s = (a + b + c) / 2;
24     return sqrt(s * (s-a) * (s-b) * (s-c));
25 }
26 // Sind die Dreiecke a1, b1, c1, and a2, b2, c2 aehnlich?
27 // Erste Zeile testet Aehnlichkeit mit gleicher Orientierung,
28 // zweite Zeile testst Aehnlichkeit mit unterschiedlicher Orientierung
29 bool similar (pt a1, pt b1, pt c1, pt a2, pt b2, pt c2) {
30     return (
31         (b2 - a2) * (c1 - a1) == (b1 - a1) * (c2 - a2) ||
32         (b2 - a2) * (conj (c1) - conj (a1)) == (conj (b1) - conj (a1)) * (c2 - a2)
33     );
34 }
35 //Linksknick von a->b nach a->c
36 double ccw(pt a, pt b, pt c) {
37     return cross(b - a, c - a); //<0 => falls Rechtsknick, 0 => kollinear, >0 => Linksknick
38 }
39 //Streckenschnitt, Strecken a-b und c-d
40 bool lineSegmentIntersection(pt a, pt b, pt c, pt d) {
41     if (ccw(a, b, c) == 0 && ccw(a, b, d) == 0) { //kollinear
42         double dist = abs(a - b);
43         return (abs(a - c) <= dist && abs(b - c) <= dist) || (abs(a - d) <= dist && abs(b - d) <= dist);
44     }
45     return ccw(a, b, c) * ccw(a, b, d) <= 0 && ccw(c, d, a) * ccw(c, d, b) <= 0;
46 }
47 //Entfernung von p zu a-b
48 double distToLine(pt a, pt b, pt p) {
49     return abs(cross(p - a, b - a)) / abs(b - a);

```

```

50 }
51 //liegt p auf a-b
52 bool pointOnLine(pt a, pt b, pt p) {
53     return abs(distToLine(a, b, p)) < EPSILON;
54 }
55 //testet, ob d in der gleichen Ebene liegt wie a, b, und c
56 bool isCoplanar(pt a, pt b, pt c, pt d) {
57     return (b - a) * (c - a) * (d - a) == 0;
58 }
59 //berechnet den Flaecheninhalt eines Polygons (nicht selbstschneidend)
60 double areaOfPolygon(vector<pt> &polygon) { //jeder Eckpunkt nur einmal im Vektor
61     double res = 0; int n = polygon.size();
62     for (int i = 0; i < (int)polygon.size(); i++)
63         res += real(polygon[i]) * imag(polygon[(i + 1) % n]) - real(polygon[(i + 1) % n]) * imag(polygon[i]);
64     return 0.5 * abs(res);
65 }
66 //testet, ob sich zwei Rechtecke (p1, p2) und (p3, p4) schneiden (jeweils gegenueberliegende Ecken)
67 bool rectIntersection(pt p1, pt p2, pt p3, pt p4) {
68     double minx12 = min(real(p1), real(p2)), maxx12 = max(real(p1), real(p2));
69     double minx34 = min(real(p3), real(p4)), maxx34 = max(real(p3), real(p4));
70     double miny12 = min(imag(p1), imag(p2)), maxy12 = max(imag(p1), imag(p2));
71     double miny34 = min(imag(p3), imag(p4)), maxy34 = max(imag(p3), imag(p4));
72     return (maxx12 >= minx34) && (maxx34 >= minx12) && (maxy12 >= miny34) && (maxy34 >= miny12);
73 }
74 //testet, ob ein Punkt im Polygon liegt (beliebige Polygone)
75 bool pointInPolygon(pt p, vector<pt> &polygon) { //jeder Eckpunkt nur einmal im Vektor
76     pt rayEnd = p + pt(1, 1000000);
77     int counter = 0, n = polygon.size();
78     for (int i = 0; i < n; i++) {
79         pt start = polygon[i], end = polygon[(i + 1) % n];
80         if (lineSegmentIntersection(p, rayEnd, start, end)) counter++;
81     }
82     return counter & 1;
83 }

```

## 4 Mathe

### 4.1 ggT, kgV, erweiterter euklidischer Algorithmus

```

1 ll gcd(ll a, ll b) {
2     return b == 0 ? a : gcd(b, a % b);
3 }
4
5 ll lcm(ll a, ll b) {
6     return a * (b / gcd(a, b)); //Klammern gegen Overflow
7 }
8
9 //Accepted in Aufgabe mit Forderung: |X|+|Y| minimal (primaer) und X<=Y (sekundaer)
10 //hab aber keinen Beweis dafuer :)
11 ll x, y, d; //a * x + b * y = d = ggT(a,b)
12 void extendedEuclid(ll a, ll b) {
13     if (!b) {
14         x = 1; y = 0; d = a; return;
15     }
16     extendedEuclid(b, a % b);
17     ll x1 = y; ll y1 = x - (a / b) * y;
18     x = x1; y = y1;
19 }

```

#### 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  $\alpha x + \beta n \equiv \alpha x \equiv 1 \pmod{n}$

- $x^{-1} \equiv \alpha \pmod{n}$

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

```

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

```

## 4.2 Primzahlsieb von Eratosthenes

```

1 vector<int> primes;
2 void primeSieve(ll n) { //berechnet die Primzahlen kleiner n
3     vector<int> isPrime(n, true);
4     for(ll i = 2; i < n; i+=2) {
5         if(isPrime[i]) {
6             primes.push_back(i);
7             if(i*i <= n) {
8                 for(ll j = i; i*j < n; j+=2) isPrime[i*j] = false;
9             }
10        }
11        if(i == 2) i++;
12    }
13 }

```

### 4.2.1 Faktorisierung

```

1 const ll PRIME_SIZE = 100000000;
2 vector<int> primes; //call primeSieve(PRIME_SIZE); before
3
4 //Factorize the number n
5 vector<int> factorize(ll n) {
6     vector<int> factor;
7     ll num = n;
8     int pos = 0;
9     while(num != 1) {
10        if(num % primes[pos] == 0) {
11            num /= primes[pos];
12            factor.push_back(primes[pos]);
13        }
14        else pos++;
15        if(primes[pos]*primes[pos] > num) break;
16    }
17    if(num != 1) factor.push_back(num);
18    return factor;
19 }

```

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

```

1 ll modPow(ll b, ll e, ll p) {
2     if (e == 0) return 1;
3     if (e == 1) return b;
4     ll half = modPow(b, e / 2, p), res = (half * half) % p;
5     if (e & 1) res *= b; res %= p;
6     return res;
7 }

```

## 4.3 LGS über $\mathbb{F}_p$

```

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

```

#### 4.4 Binomialkoeffizienten

```

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

```

#### 4.5 Satz von SPRAGUE-GRUNDY

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

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

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

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

```

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

```

#### 4.6 Maximales Teilfeld

```

1 //N := length of field
2 int maxStart = 1, maxLen = 0, curStart = 1, len = 0;
3 double maxValue = 0, sum = 0;
4 for (int pos = 0; pos < N; pos++) {

```

```

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

```

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

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

## 4.7 Kombinatorik

### 4.7.1 Berühmte Zahlen

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

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

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

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

**Bemerkung 3** • Die erste und dritte angegebene Formel sind relativ sicher gegen Overflows.

- Die erste Formel kann auch zur Berechnung der CATALAN-Zahlen bezüglich eines Moduls genutzt werden.

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

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

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

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

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

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

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

**Bemerkung 8 (STIRLING-Zahlen 2. Ordnung)** Die Anzahl der Möglichkeiten  $n$  Elemente in  $k$  nichtleere Teilmengen zu zerlegen. Begründung: Es gibt  $k$  Möglichkeiten die  $n$  in eine  $n - 1$ -Partition einzuordnen. Dazu kommt der Fall, dass die  $n$  in ihrer eigenen Teilmenge (alleine) steht.

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

#### 4.7.2 Verschiedenes

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

## 5 Strings

### 5.1 KNUTH-MORRIS-PRATT-Algorithmus

```

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

### 5.2 LEVENSHTAIN-Distanz

```

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



### 5.3 Trie

```

1 //nur fuer Kleinbuchstaben!
2 struct node {
3     node *(e)[26];
4     int c = 0; //anzahl der woerter die an dem node enden.
5     node() { for(int i = 0; i < 26; i++) e[i] = NULL; }
6 };
7
8 void insert(node *root, string *txt, int s) {
9     if(s >= txt->length()) root->c++;
10    else {
11        int idx = (int)((*txt).at(s) - 'a');
12        if(root->e[idx] == NULL) {
13            root->e[idx] = new node();
14        }
15        insert(root->e[idx], txt, s+1);
16    }
17 }
18
19 int contains(node *root, string *txt, int s) {
20     if(s >= txt->length()) return root->c;
21     int idx = (int)((*txt).at(s) - 'a');
22     if(root->e[idx] != NULL) {
23         return contains(root->e[idx], txt, s+1);
24     } else return 0;
25 }

```

### 5.4 Suffix-Array

```

1 //longest common substring in one string (overlapping not excluded)
2 //contains suffix array:-----
3 int cmp(string &s, vector<vector<int>> &v, int i, int vi, int u, int l) {
4     int vi2 = (vi + 1) % 2, u2 = u + i / 2, l2 = l + i / 2;
5     if(i == 1) return s[u] - s[l];
6     else if (v[vi2][u] != v[vi2][l]) return (v[vi2][u] - v[vi2][l]);
7     else { //beide groesser 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(s, v, i, vi, a[z], a[z-1]) == 0 ?
26             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[a[i+1]+c]) c++;
33         if(c > m) r=i, m=c;
34     }
35     return m == 0 ? "" : s.substr(a[r], m);
36 }

```

## 5.5 Longest Common Substring

```

1 //longest common substring.
2 struct lcse {
3     int i = 0, s = 0;
4 };
5 string lcp(string s[2]) {
6     if(s[0].length() == 0 || s[1].length() == 0) return "";
7     vector<lcse> a(s[0].length()+s[1].length());
8     for(int k = 0; k < a.size(); k++) a[k].i=(k < s[0].length() ? k : k - s[0].length()), a[k].s = (k < s[0].length() ? 0 : 1);
9     sort(a.begin(), a.end(), [&] (const lcse &u, const lcse &l) {
10         int ui = u.i, li = l.i;
11         while(ui < s[u.s].length() && li < s[l.s].length()) {
12             if(s[u.s][ui] < s[l.s][li]) return true;
13             else if(s[u.s][ui] > s[l.s][li]) return false;
14             ui++; li++;
15         }
16         return !(ui < s[u.s].length());
17     });
18     int r = 0, m=0, c=0;
19     for(int i = 0; i < a.size() - 1; i++) {
20         if(a[i].s == a[i+1].s) continue;
21         c = 0;
22         while(a[i].i+c < s[a[i].s].length() && a[i+1].i+c < s[a[i+1].s].length() && s[a[i].s][a[i].i+c] == s[a[i+1].s][a[i+1].i+c]) c++;
23         if(c > m) r=i, m=c;
24     }
25     return m == 0 ? "" : s[a[r].s].substr(a[r].i, m);
26 }

```

## 5.6 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:

```

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

```

- Ausgabe:

```

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

```

## 6.2 BigInteger

Hier ein kleiner Überblick über die Methoden der Klasse BigInteger:

```

1 //Returns this +,*,/,- val
2 BigInteger add(BigInteger val), multiply(BigInteger val), divide(BigInteger val), subtract(BigInteger val)
3
4 //Returns this^e
5 BigInteger pow(BigInteger e)
6
7 //Bit-Operations
8 BigInteger and(BigInteger val), or(BigInteger val), xor(BigInteger val), not(), shiftLeft(int n), shiftRight(
   int n)
9
10 //Returns the greatest common divisor of abs(this) and abs(val)
11 BigInteger gcd(BigInteger val)
12
13 //Returns this mod m, this^-1 mod m, this^e mod m
14 BigInteger mod(BigInteger m), modInverse(BigInteger m), modPow(BigInteger e, BigInteger m)
15
16 //Returns the next number that is greater than this and that is probably a prime.
17 BigInteger nextProbablePrime()
18
19 //Converting BigInteger. Attention: If the BigInteger is too big the lowest bits were chosen which fits into
   the converted data-type.
20 int intValue(), long longValue(), float floatValue(), double doubleValue()

```

## 7 Sonstiges

### 7.1 2-SAT

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

### 7.2 Sortieren in Linearzeit

Wenn die Eingabe aus einem kleinen Intervall  $[0, n)$  stammt ist Bucketsort schneller.

#### 7.2.1 Bucketsort

```

1 vector<int> res;
2 void bucketSort(vector<int> &a) { //stores result in global vector res
3     int c[BUCKETS] = {0};
4     for (int i = 0; i < (int)a.size(); i++) c[a[i]]++;
5     int C = 0;
6     for (int i = 0; i < BUCKETS; i++) {
7         int tmp = C;
8         C += c[i];
9         c[i] = tmp;
10    }
11    res.resize(a.size());
12    for (int i = 0; i < (int)a.size(); i++) {
13        res[c[a[i]]] = a[i];
14        c[a[i]]++;
15    }
16 }

```

## 7.2.2 LSD-Radixsort

```

1 //Comparable with sort from <algorithms> in a range from 0 to 5000, for values greater than 5000 use sort
2 const int p[10] = {1,10,100,1000,10000,100000,1000000,10000000,100000000,1000000000};
3
4 int getLongestNumber(vector<int> &a) {
5     int res = 0;
6     for (int i = 0; i < (int)a.size(); i++) res = max(res, (int)ceil(log10(a[i]) + 1));
7     return res;
8 }
9
10 int getIthDigit(int digit, int i) {
11     return (digit / p[i]) % 10;
12 }
13
14 void radixSort(vector<int> &a) {
15     int digits = getLongestNumber(a);
16     for (int d = 0; d < digits; d++) {
17         vector<int> bucket[10];
18         for(int i = 0; i < (int)a.size(); i++)
19             bucket[getIthDigit(a[i],d)].push_back(a[i]);
20         a.clear();
21         for(int i = 0; i < 10; i++)
22             copy(bucket[i].begin(), bucket[i].end(), back_inserter(a));
23     }
24 }

```

## 7.3 Bit Operations

```

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

```

## 7.4 Roman-Literal-Converting

```

1 map<char,int> m; map<int,char> o;
2 int num[7] = {1000,500,100,50,10,5,1};
3
4 void buildMap() {
5     m['M'] = 1000; m['D'] = 500; m['C'] = 100; m['L'] = 50; m['X'] = 10; m['V'] = 5; m['I'] = 1; m[' '] = 0;
6     o[1000] = 'M'; o[500] = 'D'; o[100] = 'C'; o[50] = 'L'; o[10] = 'X'; o[5] = 'V'; o[1] = 'I';
7 }
8
9 int convertToInt(string &s) {
10     int res = m[s[0]];
11     for(int i = 1; i < s.size(); i++) {
12         if(m[s[i-1]] < m[s[i]])
13             res -= 2*m[s[i-1]];
14         res += m[s[i]];
15     }
16     return res;
17 }
18
19 string convertToRoman(int n) {

```

```

20     string roman = "";
21     for(int i = 0; i < 7; i++) {
22         while(n >= num[i]) {
23             roman += o[num[i]];
24             n -= num[i];
25         }
26     }
27     int pos = roman.find("CCCC"); if(pos != string::npos) roman.replace(pos,4,"CD");
28     pos = roman.find("XXXX"); if(pos != string::npos) roman.replace(pos,4,"XL");
29     pos = roman.find("IIII"); if(pos != string::npos) roman.replace(pos,4,"IV");
30     pos = roman.find("DCD"); if(pos != string::npos) roman.replace(pos,3,"CM");
31     pos = roman.find("LXL"); if(pos != string::npos) roman.replace(pos,3,"XC");
32     pos = roman.find("VIV"); if(pos != string::npos) roman.replace(pos,3,"IX");
33     return roman;
34 }

```

## 7.5 Josephus-Problem

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

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

```

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

```

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

```

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

```

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

## 7.6 Gemischtes

- **JOHNSONS Reweighting Algorithmus:** Füge neue Quelle  $s$  hinzu, mit Kanten mit Gewicht 0 zu allen Knoten. Nutze **BELLMANN-FORD** zum Betsimmen der Entfernungen  $d[i]$  von  $s$  zu allen anderen Knoten. Stoppe, wenn es negative Zyklen gibt. Sonst ändere die gewichte von allen Kanten  $(u, v)$  im ursprünglichen Graphen zu  $d[u] + w[u, v] - d[v]$ . Dann sind alle Kantengewichte nichtnegativ, **DIJKSTRA** kann angewendet werden.
- Für ein System von Differenzbeschränkungen: Ändere alle Bedingungen in die Form  $a - b \leq c$ . Für jede Bedingung füge eine Kante  $(b, a)$  mit Gweight  $c$  ein. Füge Quelle  $s$  hinzu, mit Kanten zu allen Knoten mit Gewicht 0. Nutze **BELLMANN-FORD**, um die kürzesten Pfade von  $s$  aus zu finden.  $d[v]$  ist mögliche Lösung für  $v$ .
- **Min-Weight-Vertex-Cover** im bipartiten Graph: Partitioniere in  $A$ ,  $B$  und füge Kanten  $s \rightarrow A$  mit Gewicht  $w(A)$  und Kanten  $B \rightarrow t$  mit Gewicht  $w(B)$  hinzu. Füge Kanten mit Kapazität  $\infty$  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 Cover (kleinste Menge Knoten, die alle Kanten 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)

## 8 Convenience-Methoden

### 8.1 Zeileneingabe

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