# Team Contest Reference
## **Team:** stoptryharding

*Moritz Hoffmann*
*Ian Pösse*
*Marcel Wienöbst*

# Contents

# 1 ds

## 1.1 Fenwick-Tree

Can be used for computing prefix sums.

```
//note that 0 can not be used
//globaly create array
int fwktree[1000001];
int read(int index) {
   int sum = 0;
   while (index > 0) {
      sum += fenwickTree[index];
      index -= (index & -index);
   }
   return sum;
}
// n is the actual size of the tree (e.g. the array is
     used from 1 to n-1)
void update(int index, int addValue, int n) {
   while (index <= n - 1) {
      fenwickTree[index] += addValue;
      index += (index & -index);
   }
}
```

**MD5:** 9f2366fa36268df7f3bf1ac4d3772f91 | $\mathcal{O}(logn)$

## 1.2 Range maximum query

finds maximum in range [i,j] in O(1) preprocessing takes O(n log n)

```
// create A globally, contains the input
int A[10000];
// M is the DP table has size N*log N
int M[10000][20];
// N is the input size
void process(int N) {
  for(int i = 0; i < N; i++)
    M[i][0] = i;
  // filling table M
  // M[i][j] = max(M[i][j-1], M[i+(1<<(j-1))][j-1]),
  // cause interval of length 2^j can be partitioned
  // into two intervals of length 2^(j-1)
  for(int j = 1; 1 << j <= N; j++) {
    for(int i = 0; i + (1 << j) - 1 < N; i++) {
      if(A[M[i][j-1]] >= A[M[i+(1 << (j-1))][j-1]])
        M[i][j] = M[i][j-1];
      else
        M[i][j] = M[i + (1 << (j-1))][j-1];
    }
  }
}
// range is [i,j], returns index of max
int query(int N, int i, int j) {
  // k = |_ log_2(j-i+1) _|
  int k = (int) (log(j - i + 1) / log(2));
  if(A[M[i][k]] >= A[M[j- (1 << k) + 1][k]])
    return M[i][k];
  else
    return M[j - (1 << k) + 1][k];
}
```

**MD5:** eae61471981a55989f42aa6631bb2f13 | $\mathcal{O}(?)$

## 1.3 Suffix array

```
vector<int> sa, pos, tmp, lcp;
string s;
int N, gap;

bool sufCmp(int i, int j) {
  if(pos[i] != pos[j])
    return pos[i] < pos[j];
  i += gap;
  j += gap;
  return (i < N && j < N) ? pos[i] < pos[j] : i > j;
}

void buildSA()
{
  N = s.size();
  for(int i = 0; i < N; ++i) {
    sa.push_back(i);
    pos.push_back(s[i]);
  }
  tmp.resize(N);
  for(gap = 1;;gap *= 2) {
    sort(sa.begin(), sa.end(), sufCmp);
    for(int i = 0; i < N - 1; ++i) {
      tmp[i+1] = tmp[i] + sufCmp(sa[i], sa[i+1]);
    }
    for(int i = 0; i < N; ++i) {
      pos[sa[i]] = tmp[i];
    }
    if(tmp[N-1] == N-1) break;
  }
}

void buildLCP()
```

```
{
  lcp.resize(N);
  for(int i = 0, k = 0; i < N; ++i) {
    if(pos[i] != N - 1) {
      for(int j = sa[pos[i] + 1]; s[i + k] == s[j + k];) {
        ++k;
      }
      lcp[pos[i]] = k;
      if (k) --k;
    }
  }
}

int main()
{
  string r, t;
  cin >> r >> t;
  s = r + "§" + t;
  buildSA();
  buildLCP();
  for(int i = 0; i < N; ++i) {
    cout << sa[i] << " " << lcp[i] << endl;
  }
  //suffix arrays can be used for various things:
  //for example: finding lcs between to strings
}
```

**MD5:** `47eb870ecfe9cb548eb96a15c077fab7` | $\mathcal{O}(?)$

## 1.4 trie

source: github -> SuprDewd

```
template <class T>
struct trie {
  struct node {
    map<T, node*> children;
    int prefixes, words;
    node() { prefixes = words = 0; } };
  node* root;
  trie() : root(new node()) {  }
  template <class I>
  void insert(I begin, I end) {
    node* cur = root;
    while (true) {
      cur->prefixes++;
      if (begin == end) { cur->words++; break; }
      else {
        T head = *begin;
        typename map<T, node*>::const_iterator it;
        it = cur->children.find(head);
        if (it == cur->children.end()) {
          pair<T, node*> nw(head, new node());
          it = cur->children.insert(nw).first;
        } begin++, cur = it->second; } } }
  template<class I>
  int countMatches(I begin, I end) {
    node* cur = root;
    while (true) {
      if (begin == end) return cur->words;
      else {
        T head = *begin;
        typename map<T, node*>::const_iterator it;
        it = cur->children.find(head);
        if (it == cur->children.end()) return 0;
        begin++, cur = it->second; } } }
  template<class I>
  int countPrefixes(I begin, I end) {
    node* cur = root;
    while (true) {
      if (begin == end) return cur->prefixes;
      else {
        T head = *begin;
        typename map<T, node*>::const_iterator it;
        it = cur->children.find(head);
        if (it == cur->children.end()) return 0;
begin++, cur = it->second; } } } };

// use as follows
trie<char> t;
string s = "aaa";
t.insert("aaa");
```

**MD5:** `4410c62ce77f58cf564ac6881096d200` | $\mathcal{O}(?)$

## 1.5 Union-Find

*union* joins the sets $x$ and $y$ are contained in. *find* returns the representative of the set $x$ is contained in.
*Input:* number of elements $n$, element $x$, element $y$
*Output:* the representative of element $x$ or a boolean indicating whether sets got merged.

```
// globally create arrays
int p[100000];
int r[100000];

int count() {
    return count;
} // number of sets

int find(int x) {
    int root = x;
    while (p[root] >= 0) { // find root
  root = p[root];
    }
    while (p[x] >= 0) { // path compression
  int tmp = p[x];
  p[x] = root;
  x = tmp;
    }
    return root;
}

// return true, if sets merged and false, if already
    from same set
bool union(int x, int y) {
    int px = find(x);
    int py = find(y);
    if (px == py)
  return false; // same set -> reject edge
    if (r[px] < r[py]) { // swap so that always h[px
        ]>=h[py]
  int tmp = px;
  px = py;
  py = tmp;
    }
    p[py] = px; // hang flatter tree as child of
        higher tree
    r[px] = max(r[px], r[py] + 1); // update (worst-
        case) height
    count--;
```

```
    return true;
}

int main() {
    // init count to number of nodes
    int count = n;

    for(int i = 0; i < n; ++i) {
  p[i] = -1;
    }
    // do something
}
```

**MD5:** e5cb75e4854c060b0e08655fecd44ae8 | $\mathcal{O}(\alpha(n))$

# 2 graph

## 2.1 2SAT

```
// create implication graph
// do SCC
// check if var and its negation are in the same
    component
```

**MD5:** a2e8b2ae500366ce942af79e0a3f4283 | $\mathcal{O}(V+E)$

## 2.2 BellmanFord

Finds shortest pathes from a single source. Negative edge weights are allowed. Can be used for finding negative cycles.

```
// globally create arrays and graph
vector<vector<pair<int, int>>> g;
int dist[n];
int MAX_VALUE = (1 << 30);

bool bellmanFord() {
    //source is 0
    dist[0] = 0;
    //calc distances
    //the path has max length |V|-1
    for(int i = 0; i < n-1; i++) {
  //each iteration relax all edges
  for(int j = 0; j < n; j++) {
      for(int k = 0; k < g[j].size(); ++k) {
    pair<int, int> e = g[j][k];
    if(dist[j] != MAX_VALUE
       && dist[e.first] > dist[j] + e.second) {
        dist[e.first] = dist[j] + e.second;
    }
      }
  }
    }
    //check for negative-weight cycle
    for(int i = 0; i < n; i++) {
  for(int j = 0; j < g[i].size(); ++j) {
      if(dist[i] != Integer.MAX_VALUE
         && dist[e.first] > dist[i] + e.second) {
    return true;
      }
  }
    }
    return false;
}
```

**MD5:** 0dfb4089a47db73dbaaad5add58fd2a0 | $\mathcal{O}(|V|\cdot|E|)$

## 2.3 bipartite graph check

```
// traverse through graph with bfs
// assign labels 0 and 1
// if child is unexplored it gets different label from
     parent and put in the queue
// if already visited check if labels are different
```

**MD5:** 0b64ac42e8b846e97c338eaeb7d73575 | $\mathcal{O}(?)$

## 2.4 Maximum Bipartite Matching

Finds the maximum bipartite matching in an unweighted graph using DFS.

*Input:* An unweighted adjacency matrix boolean[M][N] with M nodes being matched to N nodes.

*Output:* The maximum matching. (For getting the actual matching, little changes have to be made.)

```
// globally create graph array
// adjacency matrix but smaller as only edges between
    M and N exist
bool bpGraph[M][N];

// A DFS based recursive function
// that returns true if a matching
// for vertex u is possible
bool bpm(int u,  vector<bool> &seen, vector<int> &
    matchR)
{
    // Try every job one by one
    for (int v = 0; v < N; v++)
    {
        // If applicant u is interested in
        // job v and v is not visited
        if (bpGraph[u][v] && !seen[v])
        {
            // Mark v as visited
            seen[v] = true;

            // If job v is not assigned to an
            // applicant OR previously assigned
            // applicant for job v (which is matchR[v
                ])
            // has an alternate job available.
            // Since v is marked as visited in
            // the above line, matchR[v] in the
                following
            // recursive call will not get job v again
            if (matchR[v] < 0 || bpm(bpGraph, matchR[v
                ],
                                     seen, matchR))
            {
                matchR[v] = u;
                return true;
            }
        }
    }
    return false;
}
```

```
// Returns maximum number
// of matching from M to N
int maxBPM()
{
    // An array to keep track of the
    // applicants assigned to jobs.
    // The value of matchR[i] is the
    // applicant number assigned to job i,
    // the value -1 indicates nobody is
    // assigned.
    vector<int> matchR (N);

    // Initially all jobs are available
    for(int i = 0; i < N; ++i) {
  matchR[i] = -1;
    }

    // Count of jobs assigned to applicants
    int result = 0;
    for (int u = 0; u < M; u++)
    {
        // Mark all jobs as not seen
        // for next applicant.
        vector<int> seen (N);

        // Find if the applicant u can get a job
        if (bpm(bpGraph, u, seen, matchR))
            result++;
    }
    return result;
}
```

**MD5:** `035f3ecf4735d724aad793ac4c1417c3` | $\mathcal{O}(M \cdot N)$

## 2.5 shortest path for dags

can also be applied to longest path problem in dags

```
// calc topological sorting
// go through nodes in ts order
// relaxate its neighbours
```

**MD5:** `337da9f825b3decf382ab7a8278b025c` | $\mathcal{O}(?)$

## 2.6 Recursive Depth First Search

Recursive DFS with different options (storing times, connected/unconnected graph). this is very much pseudocode, needs a lot of problem adaption anyway

*Input:* A source vertex $s$, a target vertex $t$, and adjlist $G$ and the time (0 at the start)

*Output:* Indicates if there is connection between $s$ and $t$.

```
// globally create adj list etc
vector<vector<int>> g;
int dtime[n];
int ftime[n];
int vis[n];
int pre[n];
//first call with time = 0
void rec_dfs(int u, int time){
  //it might be necessary to store the time of
      discovery
  time = time + 1;
  dtime[u] = time;

  vis[u] = 1; //new vertex has been discovered
  //For cycle check vis should be int and 0 are not
      vis nodes
  //1 are vis nodes which havent been finished and 2
      are finished nodes
  //cycle exists iff edge to node with vis=1
  //when reaching the target return true
  //not necessary when calculating the DFS-tree
  for(int i = 0; i < g[u].size(); ++i) {
      int v = g[u][i];
      //exploring a new edge
      if(!vis[v]) {
      pre[v] = u;
      if(rec_dfs(v, time)) return true;
    }
  }
  //storing finishing time
  time = time + 1;
  ftime[s] = time;
  vis[s] = 2;
  return false;
}

//if we want to visit the whole graph, even if it is
    not connected we might use this
//make sure all vertices vis value is false etc
int time = 0;
for(int i = 0; i < n; i++) {
    if(vis[i]) {
  //note that we leave out t so this does not work
      with the below function
  //adaption will not be too difficult though
  //time should not always start at zero, change if
      needed
  rec_dfs(i, 0);
    }
 }
}
```

**MD5:** `c7de745b3c11151bfa0c9093b827cefc` | $\mathcal{O}(|V| + |E|)$

## 2.7 Dijkstra

Finds the shortest paths from one vertex to every other vertex in the graph (SSSP).

For negative weights, add |min|+1 to each edge, later subtract from result.

To get a different shortest path when edges are ints, add an $\varepsilon = \frac{1}{k+1}$ on each edge of the shortest path of length $k$, run again.

*Input:* A source vertex $s$ and an adjacency list $G$.

*Output:* Modified adj. list with distances from s and predcessor vertices set.

```
int mxi = (1 << 25);

bool cmp(pair<int, int> a, pair<int, int> b)
{
    return (a.second > b.second);
}

int dijkstra(vector<vector<pair<int, int>>> &g, int N)
{
    priority_queue<pair<int, int>, vector<pair<int,
        int>>, decltype(cmp) *> pq(cmp);
```

```
    vector<int> dist (N, mxi);
    dist[0] = 0;
    pq.push({0, 0});
    while(!pq.empty()) {
        int u = pq.top().first;
        int d = pq.top().second;
        pq.pop();
        if(d > dist[u]) continue;
        if(u == N-1) return d;
        for(auto it = g[u].begin(); it != g[u].end(); ++it) {
            int v = it -> first;
            int w = it -> second;
            if(w + dist[u] < dist[v]) {
                dist[v] = w + dist[u];
                pq.push({v, dist[v]});
            }
        }
    }
    return dist[N-1];
}
```

**MD5:** b4e62c815fb25574ef371d1913584c6c | $\mathcal{O}(|E|\log|V|)$

## 2.8 FloydWarshall

Finds all shortest paths. Paths in array next, distances in ans.

```
int MAX_VALUE = (1 << 30);

void floydWarshall(int[][] graph,
       int[][] next, int[][] ans, int n) {
  for(int i = 0; i < n; i++)
    for(int j = 0; j < n; j++)
      ans[i][j] = graph[i][j];

  for (int k = 0; k < n; k++)
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
        if (ans[i][k] + ans[k][j] < ans[i][j]
                 && ans[i][k] < MAX_VALUE
                 && ans[k][j] < MAX_VALUE) {
          ans[i][j] = ans[i][k] + ans[k][j];
          next[i][j] = next[i][k];
        }
}
```

**MD5:** d93432a80b6b67952eedde97a4e7df79 | $\mathcal{O}(|V|^3)$

## 2.9 kruskal algorithm

finds the minimum spanning tree

```
// sort edges by increasing weight
// init union find (the nodes are the sets)
// go through the sorted edges and check if the corresponding nodes
// are in the same set, if yes skip the edge, if no the edge is part
// of the minimum spanning tree -> unite nodes
```

**MD5:** 82c91537f2425cfed1809d2f685dafcd | $\mathcal{O}(?)$

## 2.10 EdmondsKarp

Finds the greatest flow in a graph. Capacities must be positive.

```
#include<iostream>
#include<vector>
#include<queue>
#include<unordered_map>
#include<cmath>

using namespace std;

bool bfs(vector<unordered_map<int, long long>> &g, int s, int t, vector<int> &pre)
{
    int n = g.size();
    for(int i = 0; i < n; ++i) {
        pre[i] = -1;
    }
    vector<bool> vis (n);
    queue<int> q;
    vis[s] = true;
    q.push(s);
    while(!q.empty()) {
        int u = q.front();
        q.pop();
        if(u == t) return true;
        for(auto v = g[u].begin(); v != g[u].end(); ++v) {
            if(!vis[v->first] && (v->second) > 0) {
                vis[v->first] = true;
                pre[v->first] = u;
                q.push(v->first);
            }
        }
    }
    return vis[t];
}

long long ed_karp(vector<unordered_map<int, long long>> &g, int s, int t)
{
    long long mxf = 0;
    int n = g.size();
    vector<int> pre (n);
    while(bfs(g, s, t, pre)) {
        long long pf = (1L << 58);
        for(int v = t; v != s; v = pre[v]) {
            int u = pre[v];
            pf = min(pf, g[u][v]);
        }
        for(int v = t; v != s; v = pre[v]) {
            int u = pre[v];
            g[u][v] -= pf;
            g[v][u] += pf;
        }
        mxf += pf;
    }
    return mxf;
}
```

**MD5:** 7ea28f50383117106939588171692efe | $\mathcal{O}(|V|^2 \cdot |E|)$

## 2.11 find min cut edges

```
// do a maxflow
```

```
// go through residual graph with dfs or bfs
    traversing edges with residual cap > 0 and
// back edges with flow > 0, mark all visited nodes
// then output all edges from a marked to an unmarked
    node (maybe another BFS or something)
```

**MD5:** fb27cd04a3f1ab0ea7e494c40be18fbe | $\mathcal{O}(?)$

## 2.12 strongly connected components

```
// use two DFSs
// 1. DFS: topological sort produces list l
// 2. DFS: go through sorting and for transposed graph
     (edges are flipped) do the DFS, all reached nodes
     get the same label (are in the same component),
    of course BFS could also be used
```

**MD5:** 8ba4235a4fe35b79c0c3d4a86341c525 | $\mathcal{O}(?)$

## 2.13 topological sort

```
//two options:
//1. remove nodes with in-degree 0
//2. do DFS and prepend nodes to list when they are
    done
// (so all the nodes they depend on have already been
    prepended as they already finished)
```

**MD5:** db8519c36fbafe6a952fa5c808a5932e | $\mathcal{O}(?)$

# 3 math

## 3.1 binomial coefficient

gives binomial coefficient (n choose k)

```
// note that if we have to calculate the bin coeff
    modulo some prime
// we cannot divide, but have to multiply by the
    inverse of k
// that can be easily computed as k^p-2 % p with
    modular exponentiation (use successive squaring)
// another approach would be to just calculate n! / ((
    n-k)!*k!) (again invert denominator and use mod in
     all steps)
long long bin(int n, int k) {
  if (k == 0)
    return 1;
  else if (k > n/2)
    return bin(n, n-k);
  else
    return n*bin(n-1, k-1)/k;
}
```

**MD5:** 610ff61f07eef70ca116e75e1b15cf7c | $\mathcal{O}(k)$

## 3.2 Iterative EEA

Calculates the gcd of $a$ and $b$ and their modular inverse $x = a^{-1} \mod b$ and $y = b^{-1} \mod a$.

```
// extended euclidean algorithm - iterativ
if (b > a) {
    long tmp = a;
    a = b;
    b = tmp;
 }
long x = 0, y = 1, u = 1, v = 0;
while (a != 0) {
    long q = b / a, r = b % a;
    long m = x - u * q, n = y - v * q;
    b = a; a = r; x = u; y = v; u = m; v = n;
 }
long gcd = b;
// x = a^-1 % b, y = b^-1 % a
// ax + by = gcd
```

**MD5:** 737c57d8f09d748f54c57851ea1e759d | $\mathcal{O}(\log a + \log b)$

## 3.3 Fourier transform

calculates the fourier transform for a given vector here used for polynom multiplication in O(n log n)

```
// pol is the vector that should be transformed
// fft is the resulting vector (note the complex
    numbers)
// n is the size of pol and fft which has to be of the
     form 2^k (just fill up with zeros and choose big
    enough size)
// if inv = true the inverse transform is calculated (
    here too the result can be found in fft!)
void iterativefft(const vector<long long> &pol, vector
    <complex<double>> &fft, int n, bool inv)
{
    //copy pol into fft
    if(!inv) {
        for(int i = 0; i < n; ++i) {
            complex<double> cp (pol[i], 0);
            fft[i] = cp;
        }
    }
    //swap positions accordingly
    for(int i = 0, j = 0; i < n; ++i) {
        if(i < j) swap(fft[i], fft[j]);
        int m = n >> 1;
        while(1 <= m && m <= j) j -= m, m >>= 1;
        j += m;
    }
    for(int m = 1; m <= n; m <<= 1) { //<= or <
        double theta = (inv ? -1 : 1) * 2 * M_PI / m;
        complex<double> wm(cos(theta), sin(theta));
        for(int k = 0; k < n; k += m) {
            complex<double> w = 1;
            for(int j = 0; j < m/2; ++j) {
                complex<double> t = w * fft[k + j + m
                    /2];
                complex<double> u = fft[k + j];
                fft[k + j] = u + t;
                fft[k + j + m/2] = u - t;
                w = w*wm;
            }
        }
    }
    if(inv) {
        for(int i = 0; i < n; ++i) {
            fft[i] /= complex<double> (n);
```

```
        }
    }
}
// the polynom pol gets squared, the result is put in
    res
vector<complex<double>> fft (n);
iterativefft(pol, fft, n, false);
for(int i = 0; i < n; ++i) {
    fft[i] *= fft[i];
 }
iterativefft(pol, fft, n, true);
vector<long long> res(n);
for(int i = 0; i < n; ++i) {
    res[i] = round(fft[i].real());
 }
```

**MD5:** 9dd418b1bc3d7685c5c55b287cc8555e | $\mathcal{O}(?)$

## 3.4 Greatest Common Divisor

Calculates the gcd of two numbers $a$ and $b$ or of an array of numbers $input$.
*Input:* Numbers $a$ and $b$ or array of numbers $input$
*Output:* Greatest common divisor of the input

```
long long gcd(long long a, long long b) {
    while (b > 0) {
        long long temp = b;
        b = a % b; // % is remainder
        a = temp;
    }
    return a;
}

long long gcd(vector<long long> &input) {
    long long result = input[0];
    for(int i = 1; i < input.size(); i++)
    result = gcd(result, input[i]);
    return result;
}
```

**MD5:** 27f69f32d6e1f59d16b9c8ea0028a9fb | $\mathcal{O}(\log a + \log b)$

## 3.5 geometry lib

```
// this library has been copied from https://github.
    com/SuprDewd/T-414-AFLV
#include <complex>
using namespace std;
#define P(p) const point &p
#define L(p0, p1) P(p0), P(p1)
#define C(p0, r) P(p0), double r
#define PP(pp) pair<point,point> &pp
typedef complex<double> point;
const double pi = acos(-1.0);
const double EPS = 1e-9;
double dot(P(a), P(b)) {
    return real(conj(a) * b);
}
double cross(P(a), P(b)) {
    return imag(conj(a) * b);
}
point rotate(P(p), double radians = pi / 2, P(about) =
     point(0,0)) {
    return (p - about) * exp(point(0, radians)) +
        about;
}
point proj(P(u), P(v)) {
    return dot(u, v) / dot(u, u) * u;
}
point normalize(P(p), double k = 1.0) {
    return abs(p) == 0 ? point(0,0) : p / abs(p) * k;
}
bool parallel(L(a, b), L(p, q)) {
    return abs(cross(b - a, q - p)) < EPS;
}
double ccw(P(a), P(b), P(c)) {
    return cross(b - a, c - b);
}
bool collinear(P(a), P(b), P(c)) { return abs(ccw(a, b
     , c)) < EPS; }
double angle(P(a), P(b), P(c)) {
    return acos(dot(b - a, c - b) / abs(b - a) / abs(c
         - b));
}
bool intersect(L(a, b), L(p, q), point &res, bool
     segment = false) {
    // NOTE: check for parallel/collinear lines before
         calling this function
    point r = b - a, s = q - p;
    double c = cross(r, s), t = cross(p - a, s) / c, u
         = cross(p - a, r) / c;
    if (segment && (t < 0-EPS || t > 1+EPS || u < 0-
        EPS || u > 1+EPS))
        return false;
    res = a + t * r;
    return true;
}
point closest_point(L(a, b), P(c), bool segment =
     false) {
    if (segment) {
        if (dot(b - a, c - b) > 0) return b;
        if (dot(a - b, c - a) > 0) return a;
    }
    double t = dot(c - a, b - a) / norm(b - a);
    return a + t * (b - a);
}

typedef vector<point> polygon;
#define MAXN 1000
point hull[MAXN];
bool cmp(const point &a, const point &b) {
    return abs(real(a) - real(b)) > EPS ?
        real(a) < real(b) : imag(a) < imag(b); }
int convex_hull(vector<point> p) {
    int n = p.size(), l = 0;
    sort(p.begin(), p.end(), cmp);
    for (int i = 0; i < n; i++) {
        if (i > 0 && p[i] == p[i - 1])
            continue;
        while (l >= 2 && ccw(hull[l - 2], hull[l - 1],
             p[i]) >= 0)
            l--;
        hull[l++] = p[i];
    }
    int r = l;
    for (int i = n - 2; i >= 0; i--) {
        if (p[i] == p[i + 1])
            continue;
        while (r - l >= 1 && ccw(hull[r - 2], hull[r -
             1], p[i]) >= 0)
            r--;
```

```
        hull[r++] = p[i];
    }
    return l == 1 ? 1 : r - 1;
}
```

**MD5:** `3563f20cd2010aee48a137414d73506c` | $\mathcal{O}(?)$

## 3.6 Least Common Multiple

Calculates the lcm of two numbers $a$ and $b$ or of an array of numbers $input$.
*Input:* Numbers $a$ and $b$ or array of numbers $input$
*Output:* Least common multiple of the input

```
long long lcm(long long a, long long b) {
    return a * (b / gcd(a, b));
}

long long lcm(vector<long long> &input) {
  long result = input[0];
  for(int i = 1; i < input.size(); i++)
    result = lcm(result, input[i]);
  return result;
}
```

**MD5:** `f9b4919c74ef3ca9c1e0e2964d59fd7b` | $\mathcal{O}(\log a + \log b)$

## 3.7 phi function calculator

takes sqrt(n) time

```
int phi(int n)
{
    double result = n;
    for(int p = 2; p * p <= n; ++p) {
        if(n % p == 0) {
            while(n % p == 0) n /= p;
            result *= (1.0 - (1.0 / (double) p));
        }
    }
    if(n > 1) result *= (1.0 - (1.0 / (double) n));
    return round(result);
}
```

**MD5:** `2ec930cc10935f1638700bb74e3439d9` | $\mathcal{O}(?)$

## 3.8 Sieve of Eratosthenes

Calculates Sieve of Eratosthenes.
*Input:* A integer $N$ indicating the size of the sieve.
*Output:* A boolean array, which is true at an index $i$ iff i is prime.

```
vector<boolean> is_prime (n+1);
for (int i = 2; i <= n; i++) is_prime[i] = true;
for (int i = 2; i*i <= n; i++)
    if (is_prime[i])
  for (int j = i*i; j <= n; j+=i)
      is_prime[j] = false;
```

**MD5:** `2b965443a98027ed7f531d5360e00b48` | $\mathcal{O}(n)$

## 3.9 successive squaring

calculates $g^L$ here shown for matrix mult, but can be applied in other cases

```
void mult(int a[][nos], int b[][nos], int N)
{
    int res[nos][nos] = {0};
    for(int i = 0; i < N; i++) {
        for(int j = 0; j < N; j++) {
            for(int k = 0; k < N; k++) {
                res[i][j] = (res[i][j] + a[i][k]*b[k][
                    j]) % 10000;
            }
        }
    }
    for(int i = 0; i < N; i++) {
        for(int j = 0; j < N; j++) {
            a[i][j] = res[i][j];
        }
    }
}
// res stores the result and is initialized to the
    identity matrix
int res[nos][nos] = {0};
for(int i = 0; i < N; i++) {
    for(int j = 0; j < N; j++) {
  if(i == j) res[i][j] = 1;
    }
 }
for(int i = 0; (1 << i) <= L; i++) {
    if(((1 << i) & L) == (1 << i)) {
  mult(res, g, N);
    }
    mult(g, g, N);
 }
```

**MD5:** `f86c0e996e5eec0aedce9308951f2ddc` | $\mathcal{O}(?)$

# 4 misc

## 4.1 Binary Search

Binary searchs for an element in a sorted array.
*Input:* sorted $array$ to search in, amount $N$ of elements in $array$, element to search for $a$
*Output:* returns the index of $a$ in $array$ or $-1$ if $array$ does not contain $a$

```
int lo = 0;
int hi = N-1;
// a might be in interval [lo,hi] while lo <= hi
while(lo <= hi) {
    int mid = (lo + hi) / 2;
    // if a > elem in mid of interval,
    // search the right subinterval
    if(array[mid] < a)
  lo = mid+1;
    // else if a < elem in mid of interval,
    // search the left subinterval
    else if(array[mid] > a)
  hi = mid-1;
    // else a is found
    else
  return mid;
```

```
 }
// array does not contain a
return -1;
```

**MD5:** `2049104cd8aaced6ba8de166e9bd2abe` | $\mathcal{O}(\log n)$

## 4.2 comparator in C++

```
bool myfunction (int i, int j) {return (i<j); }

int main() {
    vector<int> vec;
    sort(vec.begin(), vec.end(), myfunction);
    priority_queue<int, vector<int>, decltype(
        myfunction) *> pq(myfunction);
}
```

**MD5:** `f4beb6e197be08977fd4f74b2537ae09` | $\mathcal{O}(?)$

## 4.3 hashing pair in C++

```
struct pairhash {
public:
  template <typename T, typename U>
  std::size_t operator()(const std::pair<T, U> &x)
      const
  {
    return std::hash<T>()(x.first) ^ std::hash<U>()(x.
        second);
  }
};

int main() {
    unordered_map<pair<unsigned int, char>, double,
        pairhash> T;
}
```

**MD5:** `49bde857f5a8078349cf97308bd8144c` | $\mathcal{O}(?)$

## 4.4 knuth-morris-pratt

finds pattern in a string

```
//----------------------------
// Returns a vector containing the zero based index of
//  the start of each match of the string K in S.
//  Matches may overlap
// source: wikipedia
//----------------------------
vector<int> KMP(string S, string K)
{
    vector<int> T(K.size() + 1, -1);
    vector<int> matches;

    if (K.size() == 0) {
        matches.push_back(0);
        return matches;
    }

    for (int i = 1; i <= K.size(); i++) {
        int pos = T[i - 1];
        while (pos != -1 && K[pos] != K[i - 1])
            pos = T[pos];
        T[i] = pos + 1;
    }

    int sp = 0;
    int kp = 0;
    while (sp < S.size()) {
        while (kp != -1 && (kp == K.size() || K[kp] !=
             S[sp]))
            kp = T[kp];
        kp++;
        sp++;
        if (kp == K.size())
            matches.push_back(sp - K.size());
    }

    return matches;
}
```

**MD5:** `856843d59319d4adac8e62968cc7ccf0` | $\mathcal{O}(?)$

## 4.5 LongestIncreasingSubsequence

*Input:* array $arr$ containing a sequence and empty array $p$ of length $arr.length$ for storing indices of the LIS
*Output:* array $s$ containing the longest increasing subsequence

```
// p[k] stores index of the predecessor of arr[k]
// in the LIS ending at arr[k]
// m[j] stores index k of smallest value arr[k]
// so there is a LIS of length j ending at arr[k]
int m[n+1];
int l = 0;
for(int i = 0; i < n; i++) {
    // bin search for the largest positive j <= l
    // with arr[m[j]] < arr[i]
    int lo = 1;
    int hi = l;
    while(lo <= hi) {
  int mid = (int) (((lo + hi) / 2.0) + 0.6);
  if(arr[m[mid]] <= arr[i])
      lo = mid+1;
  else
      hi = mid-1;
    }
    // lo is 1 greater than length of the
    // longest prefix of arr[i]
    int newL = lo;
    p[i] = m[newL-1];
    m[newL] = i;
    // if LIS found is longer than the ones
    // found before, then update l
    if(newL > l)
  l = newL;
 }
// reconstruct the LIS
vector<int> s (l);
int k = m[l];
for(int i= l-1; i>= 0; i--) {
    s[i] = arr[k];
    k = p[k];
 }
//s is the resulting seq
```

**MD5:** `8eb64842ea26475286a264c3557c355d` | $\mathcal{O}(n \log n)$

## 4.6 Mo's algorithm

Works for queries on intervals. Idea: Sort queries. Add and remove on borders has to work in O(1). Thus only usable when this is possible for the task.

```
// sort the querys [L,R] as follows: if L is in the same block (blocks have size sqrt n), sort by increasing R else sort by L
bool cmp(const pair<pair<int, int>, int> &i, const pair<pair<int, int>, int> &j) {
    if(i.first.first / BLOCK_SIZE != j.first.first / BLOCK_SIZE) {
        return i.first.first < j.first.first;
    }
    return i.first.second < j.first.second;
}

int main() {
    BLOCK_SIZE = static_cast<int>(sqrt(N));
    // store original index in queries
    vector<pair<pair<int, int>, int>> queries(M);
    vector<int> answers(M);
    //sort the queries into buckets
    sort(queries.begin(), queries.end(), cmp);
    //this is the essential part
    //for each querie we shift the previous borders one by one
    //careful analysis shows that the runtime is something like n*sqrtn + m*sqrt n (n elements and m queries)
    int mo_left = 0, mo_right = -1;
    for(int i = 0; i < M; ++i) {
  int left = queries[i].first.first;
  int right = queries[i].first.second;
  while(mo_right < right) {
      ++mo_right;
      // add can be any function as long as it is O(1)
      add(lmen[mo_right], lwomen[mo_right]);
  }
  while(mo_right > right) {
      // remove can be any function as long as it is O(1)
      remove(lmen[mo_right], lwomen[mo_right]);
      --mo_right;
  }
  while(mo_left < left) {
      remove(lmen[mo_left], lwomen[mo_left]);
      ++mo_left;
  }
  while(mo_left > left) {
      --mo_left;
      add(lmen[mo_left], lwomen[mo_left]);
  }
  answers[queries[i].second] = cur_answer;
    }
}
```

**MD5:** `3819261a7ee35c7d05e57ea167e0a27a` | $\mathcal{O}(?)$

## 4.7 Next number with n bits set

From $x$ the smallest number greater than $x$ with the same amount of bits set is computed. Little changes have to be made, if the calculated number has to have length less than 32 bits.

*Input:* number $x$ with $n$ bits set $(x = (1 << n) - 1)$
*Output:* the smallest number greater than $x$ with $n$ bits set

```
int nextNumber(int x) {
  //break when larger than limit here
  if(x == 0) return 0;
  int smallest = x & -x;
  int ripple = x + smallest;
  int new_smallest = ripple & -ripple;
  int ones  = ((new_smallest/smallest) >> 1) - 1;
  return ripple | ones;
}
```

**MD5:** `a70e3ab92156018533fa25fea2297214` | $\mathcal{O}(1)$

# 5 more math

## 5.1 Tree

Diameter: BFS from any node, then BFS from last visited node. Max dist is then the diameter. Center: Middle vertex in second step from above.

## 5.2 Divisability Explanation

$D \mid M \Leftrightarrow D \mid \texttt{digit\_sum}(\texttt{M}, \texttt{k}, \texttt{alt})$, refer to table for values of $D, k, alt$.

## 5.3 Combinatorics

- Variations (ordered): $k$ out of $n$ objects (permutations for $k = n$)
  - without repetition: $M = \{(x_1, \ldots, x_k) : 1 \leq x_i \leq n,\ x_i \neq x_j \text{ if } i \neq j\}$, $|M| = \frac{n!}{(n-k)!}$
  - with repetition: $M = \{(x_1, \ldots, x_k) : 1 \leq x_i \leq n\}, |M| = n^k$
- Combinations (unordered): $k$ out of $n$ objects
  - without repetition: $M = \{(x_1, \ldots, x_n) : x_i \in \{0, 1\},\ x_1 + \ldots + x_n = k\}, |M| = \binom{n}{k}$
  - with repetition: $M = \{(x_1, \ldots, x_n) : x_i \in \{0, 1, \ldots, k\},\ x_1 + \ldots + x_n = k\}, |M| = \binom{n+k-1}{k}$
- Ordered partition of numbers: $x_1 + \ldots + x_k = n$ (i.e. 1+3 = 3+1 = 4 are counted as 2 solutions)
  - #Solutions for $x_i \in \mathbb{N}_0$: $\binom{n+k-1}{k-1}$
  - #Solutions for $x_i \in \mathbb{N}$: $\binom{n-1}{k-1}$
- Unordered partition of numbers: $x_1 + \ldots + x_k = n$ (i.e. 1+3 = 3+1 = 4 are counted as 1 solution)
  - #Solutions for $x_i \in \mathbb{N}$: $P_{n,k} = P_{n-k,k} + P_{n-1,k-1}$ where $P_{n,1} = P_{n,n} = 1$
- Derangements (permutations without fixed points): $!n = n! \sum_{k=0}^{n} \frac{(-1)^k}{k!} = \lfloor \frac{n!}{e} + \frac{1}{2} \rfloor$

## 5.4 Polynomial Interpolation

### 5.4.1 Theory

Problem: for $\{(x_0,y_0),\dots,(x_n,y_n)\}$ find $p \in \Pi_n$ with $p(x_i) = y_i$ for all $i = 0,\dots,n$.
Solution: $p(x) = \sum_{i=0}^{n} \gamma_{0,i} \prod_{j=0}^{i-1}(x - x_i)$ where $\gamma_{j,k} = y_j$ for $k = 0$ and $\gamma_{j,k} = \frac{\gamma_{j+1,k-1} - \gamma_{j,k-1}}{x_{j+k} - x_j}$ otherwise.
Efficient evaluation of $p(x)$: $b_n = \gamma_{0,n}$, $b_i = b_{i+1}(x - x_i) + \gamma_{0,i}$ for $i = n-1,\dots,0$ with $b_0 = p(x)$.

## 5.5 Fibonacci Sequence

### 5.5.1 Binet's formula

$\begin{pmatrix} f_n \\ f_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \begin{pmatrix} 0 \\ 1 \end{pmatrix} \Rightarrow f_n = \frac{1}{\sqrt{5}}(\phi^n - \tilde{\phi}^n)$ where $\phi = \frac{1+\sqrt{5}}{2}$ and $\tilde{\phi} = \frac{1-\sqrt{5}}{2}$.

### 5.5.2 Generalization

$g_n = \frac{1}{\sqrt{5}}(g_0(\phi^{n-1} - \tilde{\phi}^{n-1}) + g_1(\phi^n - \tilde{\phi}^n)) = g_0 f_{n-1} + g_1 f_n$ for all $g_0, g_1 \in \mathbb{N}_0$

### 5.5.3 Pisano Period

Both $(f_n \mod k)_{n\in\mathbb{N}_0}$ and $(g_n \mod k)_{n\in\mathbb{N}_0}$ are periodic.

## 5.6 Reihen

$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}, \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}, \sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}$
$\sum_{i=0}^{n} c^i = \frac{c^{n+1}-1}{c-1}, c \neq 1, \sum_{i=0}^{\infty} c^i = \frac{1}{1-c}, \sum_{i=1}^{\infty} c^i = \frac{c}{1-c}, |c| < 1$
$\sum_{i=0}^{n} ic^i = \frac{nc^{n+2} - (n+1)c^{n+1} + c}{(c-1)^2}, c \neq 1, \sum_{i=0}^{\infty} ic^i = \frac{c}{(1-c)^2}, |c| < 1$

## 5.7 Binomialkoeffizienten

$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$, $\binom{n}{m}\binom{m}{k} = \binom{n}{k}\binom{n-k}{m-k}$, $\binom{m+n}{r} = \sum_{k=0}^{r}\binom{m}{k}\binom{n}{r-k}$ and in general, $n_1 + \cdots + n_p = \sum_{k_1+\cdots+k_p=m}\binom{n_1}{k_1}\cdots\binom{n_p}{k_p}$

## 5.8 Catalanzahlen

$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!n!}$
$C_0 = 1, C_{n+1} = \sum_{k=0}^{n} C_k C_{n-k}, C_{n+1} = \frac{4n+2}{n+2} C_n$

## 5.9 Geometrie

**Polygonfläche:** $A = \frac{1}{2}(x_1y_2 - x_2y_1 + x_2y_3 - x_3y_2 + \cdots + x_{n-1}y_n - x_ny_{n-1} + x_ny_1 - x_1y_n)$

## 5.10 Zahlentheorie

**Chinese Remainder Theorem:** Es existiert eine Zahl $C$, sodass: $C \equiv a_1 \mod n_1, \cdots, C \equiv a_k \mod n_k, \text{ggt}(n_i,n_j) = 1, i \neq j$
Fall $k = 2$: $m_1n_1 + m_2n_2 = 1$ mit EEA finden.
Lösung ist $x = a_1m_2n_2 + a_2m_1n_1$.
Allgemeiner Fall: iterative Anwendung von $k = 2$
**Eulersche $\varphi$-Funktion:** $\varphi(n) = n\prod_{p|n}(1 - \frac{1}{p}), p$ prim
$\varphi(p) = p - 1, \varphi(pq) = \varphi(p)\varphi(q), p, q$ prim
$\varphi(p^k) = p^k - p^{k-1}, p, q$ prim, $k \geq 1$
**Eulers Theorem:** $a^{\varphi(n)} \equiv 1 \mod n$
**Fermats Theorem:** $a^p \equiv a \mod p, p$ prim

## 5.11 Faltung

$(f * g)(n) = \sum_{m=-\infty}^{\infty} f(m)g(n-m) = \sum_{m=-\infty}^{\infty} f(n-m)g(m)$