# Bamboo Team Notes

## Contents



# 1 Number theory

## 1.1 Extended Euclide

```cpp
int bezout(int a, int b) {
    // return x such that ax + by == gcd(a, b)
    int xa = 1, xb = 0;
    while (b) {
        int q = a / b;
        int r = a - q * b, xr = xa - q * xb;
        a = b; xa = xb;
        b = r; xb = xr;
    }
    return xa;
}

pair<int, int> solve(int a, int b, int c) {
    // solve ax + by == c
    int d = __gcd(a, b);
    int x = bezout(a, b);
    int y = (d - a * x) / b;
    c /= d;
    return make_pair(x * c, y * c);
}

int main() {
    int a = 100, b = 128;
    int c = __gcd(a, b);
    int x = bezout(a, b);
    int y = (c - a * x) / b;
    cout << x << ' ' << y << endl;
    pair<int, int> xy = solve(100, 128, 40);
    cout << xy.first << ' ' << xy.second << endl;
    return 0;
}
```

## 1.2 System of linear equations

```cpp
// extended version, uses diophantine equation solver to solve system of congruent equations
pair<int, int> solve(int a, int b, int c) {
    // solve ax + by == c
    int d = __gcd(a, b);
    int x = bezout(a / d, b / d);
    int y = (d - a * x) / b;
    c /= d;
    return make_pair(x * c, y * c);
}

int lcm(int a, int b) {
    return a / __gcd(a, b) * b;
}

int solveSystem(vector<int> a, vector<int> b) {
    // xi mod bi = ai
    int A = a[0], B = b[0];
    // x mod B = A
    for (int i = 1; i < a.size(); ++i) {
        int curB = b[i], curA = a[i];
        // x = Bi + A = curB * j + curA
        pair<int, int> ij = solve(B, -curB, curA - A);
        assert(B * ij.first + A == curB * ij.second + curA);
        int newA = (B * ij.first + A);
        B = lcm(B, curB);
        A = newA % B;
        if (i + 1 == a.size()) return A;
    }
}

int main() {
    vector<int> a = {0, 3, 3};
    vector<int> b = {3, 6, 9};
    cout << solveSystem(a, b) << endl;
    return 0;
}
```

# 2 String

## 2.1 Suffix Array

```cpp
#include <bits/stdc++.h>

using namespace std;

struct SuffixArray {
    static const int N = 100010;

    int n;
    char *s;
    int sa[N], tmp[N], pos[N];
    int len, cnt[N], lcp[N];

    SuffixArray(char *t) {
        s = t;
        n = strlen(s + 1);
        buildSA();
    }

    bool cmp(int u, int v) {
        if (pos[u] != pos[v]) {
            return pos[u] < pos[v];
        }
        return (u + len <= n && v + len <= n) ? pos[u + len] < pos[v + len] : u > v;
    }

    void radix(int delta) {
        memset(cnt, 0, sizeof cnt);
        for (int i = 1; i <= n; i++) {
            cnt[i + delta <= n ? pos[i + delta] : 0]++;
        }
        for (int i = 1; i < N; i++) {
            cnt[i] += cnt[i - 1];
        }
        for (int i = n; i > 0; i--) {
            int id = sa[i];
            tmp[cnt[id + delta <= n ? pos[id + delta] : 0]--] = id;
        }
        for (int i = 1; i <= n; i++) {
```

```
            sa[i] = tmp[i];
        }
    }

    void buildSA() {
        for (int i = 1; i <= n; i++) {
            sa[i] = i;
            pos[i] = s[i];
        }
        len = 1;
        while (1) {
            radix(len);
            radix(0);
            tmp[1] = 1;
            for (int i = 2; i <= n; i++) {
                tmp[i] = tmp[i - 1] + cmp(sa[i - 1], sa[i]);
            }
            for (int i = 1; i <= n; i++) {
                pos[sa[i]] = tmp[i];
            }
            if (tmp[n] == n) {
                break;
            }
            len <<= 1;
        }

        len = 0;
        for (int i = 1; i <= n; i++) {
            if (pos[i] == n) {
                continue;
            }
            int j = sa[pos[i] + 1];
            while (s[i + len] == s[j + len]) {
                len++;
            }
            lcp[pos[i]] = len;
            if (len) {
                len--;
            }
        }
    }
};
```

## 2.2 Aho Corasick

```
struct AhoCorasick {
    static const int ALPHABET_SIZE = 26;

    struct Node {
        Node* to[ALPHABET_SIZE];
        Node* fail;
        int ending_length; // 0 if is not ending

        Node() {
            for (int i = 0; i < ALPHABET_SIZE; ++i) to[i] = nullptr;
            fail = nullptr;
            ending_length = false;
        }
    };

    Node* root;

    void add(const string &s) {
        Node* cur_node = root;
        for (char c : s) {
            c -= 'a';
            if (!cur_node->to[c]) {
                cur_node->to[c] = new Node();
            }
            cur_node = cur_node->to[c];
        }
        cur_node->ending_length = s.size();
    }

    AhoCorasick(const vector<string> &a) {
        root = new Node();
        for (const string &s : a) add(s);

        queue<Node*> Q;
        root->fail = root;
        Q.push(root);

        while (!Q.empty()) {
            Node *par = Q.front(); Q.pop();
            for (int c = 0; c < ALPHABET_SIZE; ++c) {
                if (par->to[c]) {
                    par->to[c]->fail = par == root ? root : par->fail->to[c];
                    Q.push(par->to[c]);
                } else {
                    par->to[c] = par == root ? root : par->fail->to[c];
                }
            }
        }
    }
};
```

## 2.3 Z algorithm

```
vector<int> calcZ(const string &s) {
    int L = 0, R = 0;
    int n = s.size();
    vector<int> Z(n);
    Z[0] = n;
    for (int i = 1; i < n; i++) {
        if (i > R)
        {
            L = R = i;
            while (R < n && s[R] == s[R - L]) R++;
            Z[i] = R - L; R--;
        }
        else
        {
            int k = i - L;
            if (Z[k] < R - i + 1) Z[i] = Z[k];
            else
            {
                L = i;
                while (R < n && s[R] == s[R - L]) R++;
                Z[i] = R - L; R--;
            }
        }
    }
    return Z;
}
```

## 2.4 Suffix Automaton

```
//set last = 0 everytime we add new string
struct SuffixAutomaton {
    static const int N = 100000;
    static const int CHARACTER = 26;
    int suf[N * 2], nxt[N * 2][CHARACTER], cnt, last, len[N * 2];

    SuffixAutomaton() {
        memset(suf, -1, sizeof suf);
        memset(nxt, -1, sizeof nxt);
        memset(len, 0, sizeof len);
        last = cnt = 0;
    }

    int getNode(int last, int u) {
        int q = nxt[last][u];
        if (len[last] + 1 == len[q]) {
            return q;
        }
        int clone = ++cnt;
        len[clone] = len[last] + 1;
        for (int i = 0; i < CHARACTER; i++) {
            nxt[clone][i] = nxt[q][i];
        }
        while (last != -1 && nxt[last][u] == q) {
            nxt[last][u] = clone;
            last = suf[last];
        }
        suf[clone] = suf[q];
        return suf[q] = clone;
    }

    void add(int u) {
        if (nxt[last][u] == -1) {
            int newNode = ++cnt;
            len[newNode] = len[last] + 1;
            while (last != -1 && nxt[last][u] == -1) {
                nxt[last][u] = newNode;
                last = suf[last];
            }
            if (last == -1) {
                suf[newNode] = 0;
                last = newNode;
                return;
            }
```

```
            suf[newNode] = getNode(last, u);
            last = newNode;
        } else {
            last = getNode(last, u);
        }
    }
};
```

# 3 Combinatorial optimization

# 4 Geometry

# 5 Numerical algorithms

## 5.1 Simplex Algorithm

```
/**
 * minimize c^T * x
 * subject to Ax <= b
 * and x >= 0
 * The input matrix a will have the following form
 * 0 c c c c c
 * b A A A A A
 * b A A A A A
 * b A A A A A
 * Result vector will be: val x x x x x
 **/

typedef long double ld;
const ld EPS = 1e-8;
struct LPSolver {
    static vector<ld> simplex(vector<vector<ld>> a) {
        int n = (int) a.size() - 1;
        int m = (int) a[0].size() - 1;
        vector<int> left(n + 1);
        vector<int> up(m + 1);
        iota(left.begin(), left.end(), m);
        iota(up.begin(), up.end(), 0);
        auto pivot = [&](int x, int y) {
            swap(left[x], up[y]);
            ld k = a[x][y];
            a[x][y] = 1;
            vector<int> pos;
            for (int j = 0; j <= m; j++) {
                a[x][j] /= k;
                if (fabs(a[x][j]) > EPS) pos.push_back(j);
            }
            for (int i = 0; i <= n; i++) {
                if (fabs(a[i][y]) < EPS || i == x) continue;
                k = a[i][y];
                a[i][y] = 0;
                for (int j : pos) a[i][j] -= k * a[x][j];
            }
        };
        while (1) {
            int x = -1;
            for (int i = 1; i <= n; i++) {
                if (a[i][0] < -EPS && (x == -1 || a[i][0] < a[x][0])) {
                    x = i;
                }
            }
            if (x == -1) break;
            int y = -1;
                for (int j = 1; j <= m; j++) {
                    if (a[x][j] < -EPS && (y == -1 || a[x][j] < a[x][y])) {
                    y = j;
                }
            }
            if (y == -1) return vector<ld>(); // infeasible
            pivot(x, y);
        }
        while (1) {
            int y = -1;
            for (int j = 1; j <= m; j++) {
                if (a[0][j] > EPS && (y == -1 || a[0][j] > a[0][y])) {
                    y = j;
                }
            }
            if (y == -1) break;
            int x = -1;
            for (int i = 1; i <= n; i++) {
                if (a[i][y] > EPS && (x == -1 || a[i][0] / a[i][y] < a[x][0] / a[x][y])) {
                    x = i;
                }
            }
            if (x == -1) return vector<ld>(); // unbounded
            pivot(x, y);
        }
        vector<ld> ans(m + 1);
        for (int i = 1; i <= n; i++) {
            if (left[i] <= m) ans[left[i]] = a[i][0];
        }
        ans[0] = -a[0][0];
        return ans;
    }
};
```

## 5.2 NTT

```
//Poly Invert: R(2n) = 2R(n) - R(n) ^ 2 * F where R(z) = invert F(z)
//Poly Sqrt: 2 * S(2n) = S(n) + F * S(n) ^ -1
const int MOD = 998244353;
struct NTT {
    int base = 1;
    int maxBase = 0;
    int root = 2;
    vector<int> w = {0, 1};
    vector<int> rev = {0, 1};
    NTT () {
        int u = MOD - 1;
        while (u % 2 == 0) {
            u >>= 1;
            maxBase++;
        }
        while (1) {
            if (power(root, 1 << maxBase) == 1 && power(root, 1 << (maxBase - 1)) != 1) {
                break;
            }
            root++;
        }
    }
    void ensure(int curBase) {
        assert(curBase <= maxBase);
        if (curBase <= base) return;
        rev.resize(1 << curBase);
        for (int i = 0; i < (1 << curBase); i++) {
            rev[i] = (rev[i >> 1] >> 1) + ((i & 1) << (curBase - 1));
        }
        w.resize(1 << curBase);
        for (; base < curBase; base++) {
            int wc = power(root, 1 << (maxBase - base - 1));
            for (int i = 1 << (base - 1); i < (1 << base); i++) {
                w[i << 1] = w[i];
                w[i << 1 | 1] = mul(w[i], wc);
            }
        }
    }
    void fft(vector<int> &a) {
        int n = a.size();
        int curBase = 0;
        while ((1 << curBase) < n) curBase++;
        int shift = base - curBase;
        for (int i = 0; i < n; i++) {
            if (i < (rev[i] >> shift)) swap(a[i], a[rev[i] >> shift]);
        }
        for (int k = 1; k < n; k <<= 1) {
            for (int i = 0; i < k; i++) {
                for (int j = i; j < n; j += k * 2) {
                    int foo = a[j];
                    int bar = mul(a[j + k], w[i + k]);
                    a[j] = add(foo, bar);
                    a[j + k] = sub(foo, bar);
                }
            }
        }
    }
    vector<int> mult(vector<int> a, vector<int> b) {
        int nResult = a.size() + b.size() - 1;
        int curBase = 0;
        while ((1 << curBase) < nResult) curBase++;
        ensure(curBase);
        a.resize(1 << curBase), b.resize(1 << curBase);
        fft(a);
        fft(b);
        for (int i = 0; i < (1 << curBase); i++) {
            a[i] = mul(mul(a[i], b[i]), inv(1 << curBase));
        }
```

```
        reverse(a.begin() + 1, a.end());
        fft(a);
        a.resize(nResult);
        return a;
    }
    vector<int> polyInv(vector<int> r, vector<int> f) {
        vector<int> foo = mult(r, f);
        foo.resize(f.size());
        foo[0] = sub(2, foo[0]);
        for (int i = 1; i < foo.size(); i++) {
            foo[i] = sub(0, foo[i]);
        }
        vector<int> res = mult(r, foo);
        res.resize(f.size());
        return res;
    }
    vector<int> polySqrt(vector<int> s, vector<int> invS, vector<int> f) {
        vector<int> res = mult(f, invS);
        res.resize(f.size());
        for (int i = 0; i < s.size(); i++) {
            res[i] = add(res[i], s[i]);
        }
        for (int i = 0; i < res.size(); i++) {
            res[i] = mul(res[i], INV_2);
        }
        return res;
    }
    vector<int> getSqrt(vector<int> c, int sz) {
        vector<int> sqrtC = {1}, invSqrtC = {1}; //change this if c[0] != 1
        for (int k = 1; k < (1 << sz); k <<= 1) {
            vector<int> foo(c.begin(), c.begin() + (k * 2));
            vector<int> bar = sqrtC;
            bar.resize(bar.size() * 2, 0);
            vector<int> tempInv = polyInv(invSqrtC, bar);
            sqrtC = polySqrt(sqrtC, tempInv, foo);
            invSqrtC = polyInv(invSqrtC, sqrtC);
        }
        return sqrtC;
    }
    vector<int> getInv(vector<int> c, int sz) {
        vector<int> res = {INV_2}; // change this if c[0] != 2
        for (int k = 1; k < (1 << sz); k <<= 1) {
            vector<int> foo(c.begin(), c.begin() + (k * 2));
            res = polyInv(res, foo);
        }
        return res;
    }
} ntt;
```

## 5.3 Partition Formula

```
/**
 * generating function : PI: (1 / (1 - x ^ k))
 * p(n)=p(n-1)+p(n-2)-p(n-5)-p(n-7)+p(n-12)+p(n-15)-p(n-22)- ...
 * p_k = k * (3k - 1) / 2 with k = 1, -1, 2, -2, 3, -3, ...
 */
```

# 6 Graph algorithms

## 6.1 Dinic Flow

```
const int V = 1e5;
const int INF = 1e9;
struct Flow {
    vector<int> adj[V];
    int to[V], c[V], f[V];
    int n, s, t, cnt;
    int d[V];
    int cur[V];
    Flow(int n, int s, int t) {
        this->n = n;
        this->s = s;
        this->t = t;
        cnt = 0;
    }
    int addEdge(int u, int v, int _c) {
        to[cnt] = v, c[cnt] = _c, f[cnt] = 0;
        adj[u].push_back(cnt++);
        to[cnt] = u, c[cnt] = 0, f[cnt] = 0;
        adj[v].push_back(cnt++);
    }
    bool bfs() {
        for (int i = 0; i < n; i++) d[i] = -1;
        d[s] = 0;
        queue<int> q;
        q.push(s);
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            for (int id : adj[u]) {
                int v = to[id];
                if (d[v] == -1 && f[id] < c[id]) {
                    d[v] = d[u] + 1;
                    q.push(v);
                }
            }
        }
        return d[t] != -1;
    }
    int dfs(int u, int res) {
        if (u == t) return res;
        for (int &it = cur[u]; it < adj[u].size(); it++) {
            int id = adj[u][it];
            int v = to[id];
            if (d[v] == d[u] + 1 && f[id] < c[id]) {
                int foo = dfs(v, min(c[id] - f[id], res));
                if (foo) {
                    f[id] += foo;
                    f[id ^ 1] -= foo;
                    return foo;
                }
            }
        }
        return 0;
    }
    int maxFlow() {
        int res = 0;
        while (bfs()) {
            for (int i = 0; i < n; i++) cur[i] = 0;
            while (1) {
                int foo = dfs(s, INF);
                if (!foo) break;
                res += foo;
            }
        }
        return res;
    }
};
```

## 6.2 Min Cost-Max Flow

```
struct Flow {
    static const int V = 100000;
    int head[V], to[V], c[V], cost[V], f[V], nxt[V], h[V], par[V], inQueue[V];
    int s, t, n, cnt;
    queue <int> q;
    Flow (int n, int s, int t) {
        this->n = n;
        this->s = s;
        this->t = t;
        cnt = 0;
        for (int i=  0; i < n; i++) {
            head[i] = -1;
            inQueue[i] = 0;
        }
    }
    int addEdge(int u, int v, int _c, int _cost) {
        to[cnt] = v, c[cnt] = _c, cost[cnt] = _cost, f[cnt] = 0, nxt[cnt] = head[u], head[u] = cnt++;
        to[cnt] = u, c[cnt] = 0, cost[cnt] = -_cost, f[cnt] = 0, nxt[cnt] = head[v], head[v] = cnt++;
        return cnt - 2;
    }
    pair<int, int> maxFlow () {
        int res = 0, minCost = 0;
        while (1) {
            for (int i = 0; i < n; i++) {
                par[i] = -1;
                h[i] = 2e9;
            }
            h[s] = 0;
            q.push(s);
            inQueue[s] = 1;
            while (!q.empty()) {
                int u = q.front();
                q.pop();
                inQueue[u] = 0;
                for (int id = head[u]; id != -1; id = nxt[id]) {
                    int v = to[id];
                    if (h[v] > h[u] + cost[id] && f[id] < c[id]) {
```

```
                    h[v] = h[u] + cost[id];
                    par[v] = id;
                    if (!inQueue[v]) {
                        inQueue[v] = 1;
                        q.push(v);
                    }
                }
            }
        }
        if (par[t] == -1) {
            break;
        }
        int x = t;
        int now = 2e9;
        while (x != s) {
            int id = par[x];
            now = min(now, c[id] - f[id]);
            x = to[id ^ 1];
        }
        x = t;
        while (x != s) {
            int id = par[x];
            minCost += cost[id] * now;
            f[id] += now;
            f[id ^ 1] -= now;
            x = to[id ^ 1];
        }
        res += now;
    }
    return make_pair(res, minCost);
  }
};
```

## 6.3 Bounded Feasible Flow

```
struct BoundedFlow {
    int low[N][N], high[N][N];
    int c[N][N];
    int f[N][N];
    int n, s, t;

    void reset() {
        memset(low, 0, sizeof low);
        memset(high, 0, sizeof high);
        memset(c, 0, sizeof c);
        memset(f, 0, sizeof f);
        n = s = t = 0;
    }
    void addEdge(int u, int v, int d, int c) {
        low[u][v] = d; high[u][v] = c;
    }

    int flow;
    int trace[N];

    bool findPath() {
        memset(trace, 0, sizeof trace);
        queue<int> Q;
        Q.push(s);
        while (!Q.empty()) {
            int u = Q.front(); Q.pop();
            for (int v = 1; v <= n; ++v) if (c[u][v] > f[u][v] && !trace[v]) {
                trace[v] = u;
                if (v == t) return true;
                Q.push(v);
            }
        }
        return false;
    }

    void incFlow() {
        int delta = INF;
        for (int v = t; v != s; v = trace[v])
            delta = min(delta, c[trace[v]][v] - f[trace[v]][v]);
        for (int v = t; v != s; v = trace[v])
            f[trace[v]][v] += delta, f[v][trace[v]] -= delta;
        flow += delta;
    }

    int maxFlow() {
        flow = 0;
        while (findPath()) incFlow();
        return flow;
    }
    bool feasible() {
        c[t][s] = INF;
        s = n + 1; t = n + 2;
        int sum = 0;
        for (int u = 1; u <= n; ++u) for (int v = 1; v <= n; ++v) {
            c[s][v] += low[u][v];
            c[u][t] += low[u][v];
            c[u][v] += high[u][v] - low[u][v];
            sum += low[u][v];
        }
        n += 2;
        return maxFlow() == sum;
    }
};
```

## 6.4 Hungarian Algorithm

```
struct BipartiteGraph {
    const int INF = 1e9;

    vector<vector<int> > c; // cost matrix
    vector<int> fx, fy; // potentials
    vector<int> matchX, matchY; // corresponding vertex
    vector<int> trace; // last vertex from the left side
    vector<int> d, arg; // distance from the tree && the corresponding node
    queue<int> Q; // queue used for BFS

    int n; // assume that |L| = |R| = n
    int start; // current root of the tree
    int finish; // leaf node of the augmenting path

    BipartiteGraph(int n) {
        this->n = n;
        c = vector<vector<int> >(n + 1, vector<int>(n + 1, INF));
        fx = fy = matchX = matchY = trace = d = arg = vector<int>(n + 1);
    }

    void addEdge(int u, int v, int cost) { c[u][v] = min(c[u][v], cost); }
    int cost(int u, int v) { return c[u][v] - fx[u] - fy[v]; }

    void initBFS(int root) {
        start = root;
        Q = queue<int>(); Q.push(start);
        for (int i = 1; i <= n; ++i) {
            trace[i] = 0;
            d[i] = cost(start, i);
            arg[i] = start;
        }
    }

    int findPath() {
        while (!Q.empty()) {
            int u = Q.front(); Q.pop();
            for (int v = 1; v <= n; ++v) if (trace[v] == 0) {
                int w = cost(u, v);
                if (w == 0) {
                    trace[v] = u;
                    if (matchY[v] == 0) return v;
                    Q.push(matchY[v]);
                }
                if (d[v] > w) d[v] = w, arg[v] = u;
            }
        }
        return 0;
    }

    void enlarge() {
        for (int y = finish, next; y; y = next) {
            int x = trace[y];
            next = matchX[x];
            matchX[x] = y;
            matchY[y] = x;
        }
    }

    void update() {
        int delta = INF;
        for (int i = 1; i <= n; ++i) if (trace[i] == 0) delta = min(delta, d[i]);
        fx[start] += delta;
        for (int i = 1; i <= n; ++i) {
            if (trace[i] != 0) {
                fx[matchY[i]] += delta;
                fy[i] -= delta;
            } else {
                d[i] -= delta;
                if (d[i] == 0) {
                    trace[i] = arg[i];
                    if (matchY[i] == 0)
                        finish = i;
                    else
                        Q.push(matchY[i]);
```

```
            }
        }
    }
}

void hungarian() {
    for (int i = 1; i <= n; ++i) {
        initBFS(i);
        do {
            finish = findPath();
            if (finish == 0) update();
        } while (finish == 0);
        enlarge();
    }
}

void show() {
        int ans = 0;
        for (int i = 1; i <= n; ++i) if (matchX[i]) ans += c[i][matchX[i]];
        cout << ans << endl;
        for (int i = 1; i <= n; ++i) cout << i << ' ' << matchX[i] << endl;
    }
};
```

# 7 Data structures

# 8 Miscellaneous