Contents

1 Startup templates **2**
1.1 template . . . 2
1.2 gvimrc . . . 2

2 Graph Algorithms **3**
2.1 Kuhn Max Matching . . . 3
2.2 Dinic Max-Flow . . . 3
2.3 Hungary Algo . . . 4
2.4 Min Cut . . . 4
2.5 Bridges . . . 5
2.6 Cut Vertices . . . 5
2.7 Min-Cost Max-Flow . . . 6
2.8 Strongly Connected Components . . . 7
2.9 2-SAT . . . 7

3 Linear Algebra **8**
3.1 Gauss Elimination . . . 8
3.2 Fast-Fourier Transform . . . 8
3.3 Simplex . . . 9

4 String Algorithms **11**
4.1 Suffix Array . . . 11
4.2 Suffix Tree from Suffix Array . . . 11
4.3 Z-function . . . 12
4.4 Suffix Automata . . . 13
4.5 Palindromes . . . 13
4.6 Lyndon decomposition & Duval . . . 14

5 Modular **15**

1 Startup templates

1.1 template

```
#include <vector>
#include <list>
#include <map>
#include <set>
#include <deque>
#include <stack>
#include <bitset>
#include <algorithm>
#include <functional>
#include <numeric>
#include <utility>
#include <sstream>
#include <iostream>
#include <cstdio>
#include <cmath>
#include <cstdlib>
#include <cstring>
#include <cassert>

using namespace std;

typedef long long ll;
typedef pair<int, int> pii;

template<typename T> int size(T& a){ return (int) a.size(); }
template<typename T> T sqr(T a){ return a * a; }

#define _(a, b) memset((a), (b), sizeof(a))
#define fs first
#define sc second
#define pb push_back
#define mp make_pair
#define all(a) a.begin(), a.end()
#define REP(i, a, b) for(int i = (a); i < (b); ++i)
#define REPD(i, a, b) for(int i = (b) - 1; i >= a; --i)
#define ve vector
```

1.2 gvimrc

```
set autoread
set autoindent
set autochdir
set cindent
set number
syntax on
set shiftwidth =4
set tabstop =4
colorscheme desert
set gfn =Monospace\ 12
```

2 Graph Algorithms

2.1 Kuhn Max Matching

```
// need: graph( head, to, nxt ), used, mate
bool kuhn(int v) {
    if (used[v]) return false;
    used[v] = true;
    for(int e = head[v]; e != -1; e = nxt[e]) {
        if (mate[to[e]] == -1) {
            mate[to[e]] = v;
            return true;
        }
    }
    for(int e = head[v]; e != -1; e = nxt[e]) {
        if (kuhn(mate[to[e]])) {
            mate[to[e]] = v;
            return true;
        }
    }
    return false;
}

int max_matching() {
    int res = 0;
    REP(i, 0, n) {
        _(used, false);
        if (kuhn(i)) res++;
    }
    return res;
}
```

2.2 Dinic Max-Flow

```
// need: graph( head, nxt, to, capa, flow ), dist, q
bool bfs(int src, int dest) {
    _(dist, -1);
    dist[src] = 0;
    int H = 0;
    q[H++] = src;
    REP(i, 0, H) {
        int cur = q[i];
        for(int e = head[cur]; e != -1; e = nxt[e]) {
            if (capa[e] > flow[e] && dist[to[e]] == -1) {
                dist[to[e]] = dist[cur] + 1;
                q[H++] = to[e];
            }
        }
    }
    return dist[dest] >= 0;
}

int dfs(int cur, int curflow) {
    if (cur == dest) return curflow;
    int d ;
    for(int& e = work[cur]; e != -1; e = nxt[e]) {
        if (capa[e] > flow[e] && (dist[to[e]] == dist[cur] + 1) &&
                (d = dfs(to[e], min(curflow, capa[e] - flow[e])))) {
            flow[e] += d;
            flow[e ^ 1] -= d;
            return d;
        }
    }
    return 0;
}

int dinic() {
    int res = 0;
    while (bfs(src, des)) {
        int d;
        memcpy(work, head, sizeof(head));
```

```
        while (true) {
            d = dfs(src, INF);
            if (d == 0) break;
            res += d;
        }
    }
    return res;
}
```

2.3 Hungary Algo

```
// need: a[n][m], all indices start with 1
vector<int> u (n+1), v (m+1), p (m+1), way (m+1);
for (int i=1; i<=n; ++i) {
  p[0] = i;
  int j0 = 0;
  vector<int> minv (m+1, INF);
  vector<char> used (m+1, false);
  do {
    used[j0] = true;
    int i0 = p[j0],  delta = INF,  j1;
    for (int j=1; j<=m; ++j)
      if (!used[j]) {
        int cur = a[i0][j]-u[i0]-v[j];
        if (cur < minv[j])
          minv[j] = cur,  way[j] = j0;
        if (minv[j] < delta)
          delta = minv[j],  j1 = j;
      }
    for (int j=0; j<=m; ++j)
      if (used[j])
        u[p[j]] += delta,  v[j] -= delta;
      else
        minv[j] -= delta;
    j0 = j1;
  } while (p[j0] != 0);
  do {
    int j1 = way[j0];
    p[j0] = p[j1];
    j0 = j1;
  } while (j0);
}
// restore ans[] -- selected column for each row
for (int j=1; j<=m; ++j)
  ans[p[j]] = j;
// cost
int cost = -v[0];
```

2.4 Min Cut

```
pair<int, ve<int> > GetMinCut(ve< ve<int> > &weights) {
  int N = weights.size();
  ve<int> used(N), cut, best_cut;
  int best_weight = -1;

  REPD(phase, 0, N) {
    ve<int> w = weights[0];
    ve<int> added = used;
    int prev, last = 0;
    REP(i, 0, phase) {
      prev = last;
      last = -1;
      REP(j, 1, N)
        if (!added[j] && (last == -1 || w[j] > w[last])) last = j;
      if (i == phase-1) {
        REP(j, 0, N) weights[prev][j] += weights[last][j];
        REP(j, 0, N) weights[j][prev] = weights[prev][j];
        used[last] = true;
        cut.pb(last);
        if (best_weight == -1 || w[last] < best_weight) {
```

```
          best_cut = cut;
          best_weight = w[last];
        }
      } else {
        REP(j, 0, N)
          w[j] += weights[last][j];
        added[last] = true;
      }
    }
  }
  return mp(best_weight, best_cut);
}
```

2.5 Bridges

```
// need: graph( head, to, nxt ), used, tin, fup, timer
void dfs (int v, int par = -1) {
  used[v] = true;
  tin[v] = fup[v] = timer++;
  for(int e = head[v]; e != -1; e = nxt[e]) {
    int u = to[e];
    if (u == par)  continue;
    if (used[u])
      fup[v] = min (fup[v], tin[u]);
    else {
      dfs (u, v);
      fup[v] = min (fup[v], fup[u]);
      if (fup[u] > tin[v])
        IS_BRIDGE(v, u);
    }
  }
}

void find_bridges() {
  timer = 0;
  _(used, 0)
  REP(i, 0, N)
    if (!used[i]) dfs (i);
}
```

2.6 Cut Vertices

```
// need: graph (head, to, nxt), tin, fup, used, timer
void dfs (int v, int par = -1) {
  used[v] = true;
  tin[v] = fup[v] = timer++;
  int children = 0;
  for(int e = head[v]; e != -1; e = nxt[e]) {
    int u = to[e];
    if (u == par)  continue;
    if (used[u])
      fup[v] = min (fup[v], tin[u]);
    else {
      dfs (u, v);
      fup[v] = min (fup[v], fup[u]);
      if (fup[u] >= tin[v] && p != -1)
        IS_CUTPOINT(v);
      ++children;
    }
  }
  if (p == -1 && children > 1)
    IS_CUTPOINT(v);
}

int main() {
  timer = 0;
  _(used, 0);
  dfs (0);
}
```

2.7 Min-Cost Max-Flow

```
// need: graph (head, nxt, to, from, capa, cost, flow)
// pi, dist, prve
void updatePotentials() {
  memcpy(pi, dist, sizeof(int) * N);
}

bool fordBellman(int src, int dst) {
  REP(i, 0, N) dist[i] = INF;
  dist[src] = 0;
  bool changed;

  REP(phase, 0, N) {
    changed = false;
    REP(v, 0, N) {
      if(dist[v] == INF) continue;
      for(int e = head[v]; e != -1; e = nxt[e]) {
        int u = to[e];
        if(capa[e] > flow[e] && dist[u] > dist[v] + cost[e]) {
          dist[u] = dist[v] + cost[e];
          prve[u] = e;
          changed = true;
        }
      }
    }
    if(!changed) break;
  }
  return !changed;
}

set< pii > q;
bool dijkstra(int src, int dst) {
  REP(i, 0, N) dist[i] = INF;
  dist[src] = 0;
  q.insert( mp(0, 0) );

  while(size(q)) {
    pii tmp = (*q.begin());
    int v = tmp.sc, d = tmp.fs;
    q.erase(q.begin());
    if(d != dist[v]) continue;

    for(int e = head[v]; e != -1; e = nxt[e]) {
      int u = to[e];
      if(capa[e] > flow[e] && dist[u] > dist[v] + cost[e] - pi[v] + pi[u]) {
        dist[u] = dist[v] + cost[e] - pi[v] + pi[u];
        prve[u] = e;
        q.insert( mp(dist[u], u) );
      }
    }
  }
  return dist[dst] != INF;
}

pii minCostMaxFlow(int src, int dst) {
  if(!fordBellman(src, dst)) return mp(0, 0);
  int sumFlow = 0, sumCost = 0;

  do {
    int curFlow = INF, curCost = 0;
    int cur = dst;
    while(cur != src) {
      int e = prve[cur];
      curFlow = min(curFlow, capa[e] - flow[e]);
      curCost += cost[e];
      cur = from[e];
    }
    cur = dst;
    while(cur != src) {
      int e = prve[cur];
      flow[e] += curFlow;
      flow[e ^ 1] -= curFlow;
      cur = from[e];
```

```
    }
    sumCost += curFlow * curCost;
    updatePotentials();
  } while(dijkstra(src, dst));

  return mp(sumFlow, sumCost);
}
```

2.8 Strongly Connected Components

```
// need: graph (head, to, nxt), reverse graph (rhead, rto, rnxt)
// used, compID, order
void dfs1(int v) {
  used[v] = true;
  for(int e = head[v]; e != -1; e = nxt[e]) {
    if(!used[to[e]]) dfs1(to[e]);
  }
  order.pb(v);
}
void dfs2(int v, int id) {
  used[v] = true;
  compID[v] = id;
  for(int e = rhead[v]; e != -1; e = rnxt[e]) {
    if(!used[ rto[e] ]) dfs2(rto[e]);
  }
}
void main() {
  _(used, false);
  REP(i, 0, N)
    if(!used[i]) dfs1(i);
  _(used, false);
  int id = 0;
  REPD(i, 0, N) {
    int v = order[i];
    if(!used[v]) dfs2(v, id++);
  }
}
```

2.9 2-SAT

Problem: $(a \lor c) \& (a \lor !b) \& \ldots$

Edges: $(a \lor b)$ is equivalent to $(!a \Rightarrow b) \lor (!b \Rightarrow a)$

Solution: there is no solution iff for some x $compID[x] = compID[!x]$, else see code below

```
// need: graph, scc
int main() {
  _(used, false);
  REP(i, 0, N)
    if (!used[i]) dfs1 (i);

  _(compID, -1);
  int id = 0;
  REPD(i, 0, N) {
    int v = order[i];
    if (comp[v] == -1) dfs2(v, id++);
  }

  REP(i, 0, N)
    if (compID[i] == compID[i^1]) {
      puts ("NO SOLUTION");
      return 0;
    }
  REP(i, 0, N) {
    int ans = comp[i] > comp[i^1] ? i : i^1;
    printf ("%d ", ans);
  }
}
```

3 Linear Algebra

3.1 Gauss Elimination

```
// Ax = B. RETURN: determinant, A -> A^(-1), B -> solution
typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;

T GaussJordan(VVT &a, VT &b) {
  const int n = a.size();
  ve<int> irow(n), icol(n), ipiv(n);
  T det = 1;

  REP(i, 0, n) {
    int pj = -1, pk = -1;
    REP(j, 0, n) if (!ipiv[j])
      REP(k, 0, n) if (!ipiv[k])
        if (pj == -1 || fabs(a[j][k]) > fabs(a[pj][pk])) { pj = j; pk = k; }
    if (fabs(a[pj][pk]) < EPS) { cerr << "Matrix is singular." << endl; exit(0); }
    ipiv[pk]++;
    swap(a[pj], a[pk]);
    swap(b[pj], b[pk]);
    if (pj != pk) det *= -1;
    irow[i] = pj;
    icol[i] = pk;

    T c = 1.0 / a[pk][pk];
    det *= a[pk][pk];
    a[pk][pk] = 1.0;
    REP(p, 0, n) a[pk][p] *= c;
    b[pk] *= c;
    REP(p, 0, n) if (p != pk) {
      c = a[p][pk];
      a[p][pk] = 0;
      REP(q, 0, n) a[p][q] -= a[pk][q] * c;
      b[p] -= b[pk] * c;
    }
  }

  REPD(p, 0, n) if (irow[p] != icol[p]) {
    REP(k, 0, n) swap(a[k][irow[p]], a[k][icol[p]]);
  }

  return det;
}
```

3.2 Fast-Fourier Transform

```
typedef complex<double> base;

void fft (vector<base> & a, bool invert) {
  int n = (int) a.size();

  for (int i = 1, j = 0; i < n; i++) {
    int bit = n >> 1;
    for (; j >= bit; bit >>= 1)
      j -= bit;
    j += bit;
    if (i < j) swap (a[i], a[j]);
  }

  for (int len = 2; len <= n; len <<= 1) {
    double ang = 2 * PI/len * (invert ? -1 : 1);
    base wlen (cos(ang), sin(ang));
    for (int i = 0; i < n; i += len) {
      base w (1);
      for (int j = 0; j < len/2; j++) {
        base u = a[i+j], v = a[i+j+len/2] * w;
        a[i+j] = u + v;
        a[i+j+len/2] = u - v;
```

```
        w *= wlen;
      }
    }
  }
  if (invert)
    REP(i, 0, n) a[i] /= n;
}
```

3.3 Simplex

```
// maximize c^T x
// Ax <= b
// x >= 0

struct LPSolver {
  int m, n;
  ve<int> B, N;
  ve< ve<double> > D;

  LPSolver(const ve< ve<double> > &A, const ve<double> &b, const ve<double> &c) :
    m(b.size()), n(c.size()), N(n+1), B(m), D(m+2, ve<double>(n+2)) {
      REP(i, 0, m) REP(j, 0, n) D[i][j] = A[i][j];
      REP(i, 0, m) { B[i] = n+i; D[i][n] = -1; D[i][n+1] = b[i]; }
      REP(j, 0, n) { N[j] = j; D[m][j] = -c[j]; }
      N[n] = -1; D[m+1][n] = 1;
    }

  void Pivot(int r, int s) {
    REP(i, 0, m + 2) if (i != r)
      REP(j, 0, n + 2) if (j != s)
        D[i][j] -= D[r][j] * D[i][s] / D[r][s];
    REP(j, 0, n + 2) if (j != s) D[r][j] /= D[r][s];
    REP(i, 0, n + 2) if (i != r) D[i][s] /= -D[r][s];
    D[r][s] = 1.0 / D[r][s];
    swap(B[r], N[s]);
  }

  bool Simplex(int phase) {
    int x = phase == 1 ? m+1 : m;
    while (true) {
      int s = -1;
      REP(j, 0, n + 1) {
        if (phase == 2 && N[j] == -1) continue;
        if (s == -1 || D[x][j] < D[x][s] || D[x][j] == D[x][s] && N[j] < N[s]) s = j;
      }
      if (D[x][s] >= -EPS) return true;
      int r = -1;
      REP(i, 0, m) {
        if (D[i][s] <= 0) continue;
        if (r == -1 || D[i][n+1] / D[i][s] < D[r][n+1] / D[r][s] ||
            D[i][n+1] / D[i][s] == D[r][n+1] / D[r][s] && B[i] < B[r]) r = i;
      }
      if (r == -1) return false;
      Pivot(r, s);
    }
  }

  double Solve(ve<double> &x) {
    int r = 0;
    REP(i, 1, m) if (D[i][n+1] < D[r][n+1]) r = i;
    if (D[r][n+1] <= -EPS) {
      Pivot(r, n);
      if (!Simplex(1) || D[m+1][n+1] < -EPS) return -numeric_limits<double>::infinity();
      REP(i, 0, m) if (B[i] == -1) {
        int s = -1;
        REP(j, 0, n + 1)
          if (s == -1 || D[i][j] < D[i][s] || D[i][j] == D[i][s] && N[j] < N[s]) s = j;
        Pivot(i, s);
      }
    }
    if (!Simplex(2)) return numeric_limits<double>::infinity();
    x = ve<double>(n);
```

```
    REP(i, 0, m) if (B[i] < n) x[B[i]] = D[i][n+1];
    return D[m][n+1];
  }
};
```

4 String Algorithms

4.1 Suffix Array

```
struct entry {
  int nr[2], p;
} L[MAXN], tmp[MAXN];
#define eq(a, b) ((a).nr[0] == (b).nr[0] && (a).nr[1] == (b).nr[1])
int cnt[MAXN], p[2][MAXN];

void radixPass(entry * a, int N, int pass, int K, entry * b) {
  memset(cnt, 0, (K + 1) * sizeof(int));
  REP(i, 0, N) cnt[a[i].nr[pass]]++;
  int sum = 0;
  REP(i, 0, K + 1) {
    sum += cnt[i];
    cnt[i] = sum - cnt[i];
  }
  REP(i, 0, N) b[cnt[a[i].nr[pass]]++] = a[i];
}

void makeSA(int * s, int N, int * suftab, int * isuftab) {
  REP(i, 0, N) p[0][i] = s[i];
  int k = 200;
  for(int step = 1, len = 1; ; step ^= 1, len <<= 1) {
    for(int i = 0, j = len; i < N; i++, j++) {
      L[i].nr[0] = p[step ^ 1][i];
      L[i].nr[1] = j < N ? p[step ^ 1][j] : 0;
      L[i].p = i;
    }
    radixPass(L, N, 1, k, tmp);
    radixPass(tmp, N, 0, k, L);
    k = 1;
    REP(i, 0, N) {
      p[step][L[i].p] = i > 0 && eq(L[i], L[i - 1]) ?
        p[step][L[i - 1].p] : k++;
    }
    if(k > N) break;
  }
  REP(i, 0, N) suftab[i] = L[i].p;
  REP(i, 0, N) isuftab[suftab[i]] = i;
}

void makeLCP(int * s, int * suftab, int * isuftab, int N, int * lcptab) {
  int cur = 0;
  REP(i, 0, N) {
    if(isuftab[i] == 0) continue;
    int ii = i + cur, jj = suftab[isuftab[i] - 1] + cur;
    while(ii < N && jj < N && s[ii] == s[jj]) ii++, jj++, cur++;
    lcptab[isuftab[i]] = cur--;
    if(cur < 0) cur = 0;
  }
}
```

4.2 Suffix Tree from Suffix Array

```
struct Seg {
  int lb, rb, lcp;
  vector<Seg*> childList;
  void init(int l, int i, int j) {
    lb = i; rb = j; lcp = l;
  }
  void add(Seg * son) {
    childList.pb(son);
  }
};
typedef Seg* pSeg;

struct Stack {
  pSeg segs[MAXN << 1];
  int size;
```

```
  void push(pSeg seg) {
    segs[size++] = seg;
  }
  pSeg pop() {
    return segs[--size];
  }
  pSeg top() {
    return segs[size - 1];
  }
} stack;

pSeg top() { return stack.top(); }
void push(pSeg seg) { stack.push(seg); }
pSeg pop() { return stack.pop(); }

pSeg init(int lcp, int lb, int rb) {
  pSeg ret = new Seg;
  ret->init(lcp, lb, rb);
  return ret;
}

pSeg makeTree() {
  stack.size = 0;
  pSeg lastInterval = NULL;
  int lastSingleton = 0;
  stack.push(init(0, 0, -1));
  REP(i, 1, N) {
    int lb = i - 1;
    pSeg singleton = init(N - suftab[i - 1] - 1, i - 1, i - 1);
    //process(singleton);

    while(lcptab[i] < top()->lcp) {
      if(singleton != NULL) {
        top()->add(singleton);
        singleton = NULL;
      }
      top()->rb = i - 1;
      lastInterval = pop();
      //process(lastInterval);
      lb = lastInterval->lb;
      if(lcptab[i] <= top()->lcp) {
        top()->add(lastInterval);
        lastInterval = NULL;
      }
    }
    if(lcptab[i] > top()->lcp) {
      if(lastInterval != NULL) {
        pSeg seg = init(lcptab[i], lb, -1);
        seg->add(lastInterval);
        push(seg);
        lastInterval = NULL;
      } else push(init(lcptab[i], lb, -1));
    }
    if(singleton != NULL) {
      top()->add(singleton);
    }
  }
  assert(stack.size == 1);
  //process(top());
  return top();
}
```

4.3 Z-function

```
ve<int> z_function (string s) {
  int n = (int) s.length();
  ve<int> z (n);
  for (int i=1, l=0, r=0; i<n; ++i) {
    if (i <= r)
      z[i] = min (r-i+1, z[i-l]);
    while (i+z[i] < n && s[z[i]] == s[i+z[i]])
      ++z[i];
```

```
    if (i+z[i]-1 > r)
      l = i,  r = i+z[i]-1;
  }
  return z;
}
```

4.4 Suffix Automata

```
struct state {
  int len, link;
  map<char,int> next;
};

state st[MAXLEN*2];
int sz, last;

void sa_init() {
  sz = last = 0;
  st[0].len = 0;
  st[0].link = -1;
  ++sz;
}

void sa_extend (char c) {
  int cur = sz++;
  st[cur].len = st[last].len + 1;
  int p;
  for (p=last; p!=-1 && !st[p].next.count(c); p=st[p].link)
    st[p].next[c] = cur;
  if (p == -1)
    st[cur].link = 0;
  else {
    int q = st[p].next[c];
    if (st[p].len + 1 == st[q].len)
      st[cur].link = q;
    else {
      int clone = sz++;
      st[clone].len = st[p].len + 1;
      st[clone].next = st[q].next;
      st[clone].link = st[q].link;
      for (; p!=-1 && st[p].next[c]==q; p=st[p].link)
        st[p].next[c] = clone;
      st[q].link = st[cur].link = clone;
    }
  }
  last = cur;
}
```

4.5 Palindromes

```
for(i = 0; i < n; i++){
  if(i > r) k = 1;
  else k = min(d1[l + r - i], r - i);

  while(0 <= i-k && i+k < n && s[i - k] == s[i + k]) k++;
  d1[i] = k;
  if(i + k - 1 > r)
    r = i + k - 1, l = i - k + 1;
}

for(i = 0; i < n; i++){
  if(i > r) k = 0;
  else k = min(d2[l + r - i + 1], r - i + 1);

  while(i + k < n && i - k - 1 >= 0 && s[i+k] == s[i - k - 1]) k++;
  d2[i] = k;

  if(i + k - 1 > r)
    l = i - k, r = i + k - 1;
}
```

4.6 Lyndon decomposition & Duval

```
// Lyndon decomposition
for(int i = 0; i < n;) {
  int j=i+1, k=i;
  while (j < n && s[k] <= s[j]) {
    if (s[k] < s[j])
      k = i;
    else
      ++k;
    ++j;
  }
  while (i <= k) {
    cout << s.substr (i, j-k) << ' ';
    i += j - k;
  }
}

string min_cyclic_shift (string s) {
  s += s;
  int n = (int) s.length();
  int i=0, ans=0;
  while (i < n/2) {
    ans = i;
    int j=i+1, k=i;
    while (j < n && s[k] <= s[j]) {
      if (s[k] < s[j])
        k = i;
      else
        ++k;
      ++j;
    }
    while (i <= k)  i += j - k;
  }
  return s.substr (ans, n/2);
}
```

5 Modular

```
// All algorithms described here work on nonnegative integers.

// return a % b (positive value)
int mod(int a, int b) {
  return ((a%b)+b)%b;
}

// computes gcd(a,b)
int gcd(int a, int b) {
  int tmp;
  while(b){a%=b; tmp=a; a=b; b=tmp;}
  return a;
}

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

// returns d = gcd(a,b); finds x,y such that d = ax + by
int extended_euclid(int a, int b, int &x, int &y) {
  int xx = y = 0;
  int yy = x = 1;
  while (b) {
    int q = a/b;
    int t = b; b = a%b; a = t;
    t = xx; xx = x-q*xx; x = t;
    t = yy; yy = y-q*yy; y = t;
  }
  return a;
}

// finds all solutions to ax = b (mod n)
ve<int> modular_linear_equation_solver(int a, int b, int n) {
  int x, y;
  ve<int> solutions;
  int d = extended_euclid(a, n, x, y);
  if (!(b%d)) {
    x = mod (x*(b/d), n);
    for (int i = 0; i < d; i++)
      solutions.push_back(mod(x + i*(n/d), n));
  }
  return solutions;
}

// computes b such that ab = 1 (mod n), returns -1 on failure
int mod_inverse(int a, int n) {
  int x, y;
  int d = extended_euclid(a, n, x, y);
  if (d > 1) return -1;
  return mod(x,n);
}

// find z such that z % x = a, z % y = b.  Here, z is unique modulo M = lcm(x,y).
// Return (z,M).  On failure, M = -1.
pii chinese_remainder_theorem(int x, int a, int y, int b) {
  int s, t;
  int d = extended_euclid(x, y, s, t);
  if (a%d != b%d) return make_pair(0, -1);
  return make_pair(mod(s*b*x+t*a*y,x*y)/d, x*y/d);
}

// Chinese remainder theorem: find z such that
// z % x[i] = a[i] for all i.  Note that the solution is
// unique modulo M = lcm_i (x[i]). Return (z,M). On failure, M = -1.
pii chinese_remainder_theorem(const ve<int> &x, const ve<int> &a) {
  pii ret = make_pair(a[0], x[0]);
  for (int i = 1; i < x.size(); i++) {
    ret = chinese_remainder_theorem(ret.second, ret.first, x[i], a[i]);
    if (ret.second == -1) break;
  }
  return ret;
```

```
}

// computes x and y such that ax + by = c; on failure, x = y =-1
void linear_diophantine(int a, int b, int c, int &x, int &y) {
  int d = gcd(a,b);
  if (c%d) {
    x = y = -1;
  } else {
    x = c/d * mod_inverse(a/d, b/d);
    y = (c-a*x)/b;
  }
}
```