Table of Contents

Stable Marriage:

```
/* A person has an integer preference for each of the persons of the opposite
 * sex, produces a matching of each man to some woman. The matching will follow:
 *          - Each man is assigned to a different woman (n must be at least m)
 *          - No two couples M1W1 and M2W2 will be unstable.
 * Two couples are unstable if (M1 prefers W2 over W1 and W1 prefers M2 over M1)
 * INPUT: m – number of man, n – number of woman (must be at least as large as m)
 *          - L[i][]: the list of women in order of decreasing preference of man i
 *          - R[j][i]: the attractiveness of i to j.
 * OUTPUTS: - L2R[]: the mate of man i (always between 0 and n-1)
 *          - R2L[]: the mate of woman j (or -1 if single)      */
int m, n, L[MAXM][MAXW], R[MAXW][MAXM], L2R[MAXM], R2L[MAXW], p[MAXM];
void stableMarriage() {
    memset( R2L, -1, sizeof( R2L ) );
    memset( p, 0, sizeof( p ) );
    for( int i = 0; i < m; i++ ) { // Each man proposes...
        int man = i;
        while( man >= 0 ) {
            int wom;
            while( 1 ) {
                wom = L[man][p[man]++];
                if( R2L[wom] < 0 || R[wom][man] > R[wom][R2L[wom]] ) break;
            }
            int hubby = R2L[wom];
            R2L[L2R[man] = wom] = man;
            man = hubby;
        }
    }
}
```

Stoer Wagner (Minimum Min Cut between All Pairs):

```
// Maximum edge weight (MAXW * NN * NN must fit into an int), NN number of vertices
#define MAXW 1000
int g[NN][NN], v[NN], w[NN], na[NN]; // Adjacency matrix and some internal arrays
bool a[NN];
int minCut( int n ) { // 0 indexed
    int i, j;
    for( i = 0; i < n; i++ ) v[i] = i; // init the remaining vertex set
    int best = MAXW * n * n;
    while( n > 1 ) { // initialize the set A and vertex weights
        a[v[0]] = true;
        for( i = 1; i < n; i++ ) {
            a[v[i]] = false;
            na[i - 1] = i;
            w[i] = g[v[0]][v[i]];
        }
        int prev = v[0];
        for(i = 1; i < n; i++) { // find the most tightly connected non-A vertex
            int zj = -1;
            for(j=1;j<n;j++) if( !a[v[j]] && ( zj < 0 || w[j] > w[zj] ) ) zj = j;
            a[v[zj]] = true; // add it to A
            if( i == n - 1 ) { // last vertex?
                best = (best < w[zj]) ? best : w[zj]; // remember the cut weight
                for(j=0;j<n;j++) g[v[j]][prev] = g[prev][v[j]] += g[v[zj]][v[j]];
                v[zj] = v[--n];
                break;
            }
            prev = v[zj];
            for( j = 1; j < n; j++ ) if( !a[v[j]] ) w[j] += g[v[zj]][v[j]];
        }
    }
    return best;
}
```

MST (Directed Graph):

1. For each node (except the root), look for the minimum weight incoming edge.
2. Look for cycles, if there's no cycle, we already have a tree (which is an MST) goto End
3. Pick one cycle and find an edge p->q, p is in set (not part of the cycle). q is in set (s part of the cycle). Pick this p and q such that: cost of (p->q + sum of all edges in the cycle) - the minimum incoming edge to q (computed in step 1) is minimum. Return to step 2.

```
struct edge { // Caution: The vertices should be reachable from the root
    int v, w;
    bool operator < ( const edge &v ) const { return w > v.w; }
};
vector <edge > adj[MAX];     // For saving incoming edges and their costs
int DMST( int n, int root ) {   // 1 indexed
    int i, res=0, pr[MAX], cost[MAX], sub[MAX], sn[MAX], visited[MAX];
    vector <int> ::iterator v, it;
    vector <int> node[MAX];
    for(i = 0; i <= n; i++) {
        node[i].clear(); node[i].push_back( i );
        sn[i] = i, sub[i] = pr[i] = 0;
    }
    for(i = 1; i <= n; i++) if( i != root ) {
        sort( adj[i].begin(), adj[i].end() ); // sorted in descending order of w
        pr[i] = adj[i].back().v;
        cost[i] = sub[i] = adj[i].back().w;
        res += cost[i];
    }
    bool cycle = true;
    while( cycle ) {
        cycle = false;
        CLR( visited );
        for(i = 1; i <= n; i++) {
            if( visited[i] || sn[i] != i ) continue;
            int cur = i;
            do {
                visited[cur] = i;
                cur = pr[cur];
            }while( cur > 0 && !visited[ cur ] );
            if( cur > 0 && visited[ cur ] == i ) {
                cycle = true;
                int start = cur; // assert( sn[start] == start ) ;
                do{
                    if( *node[cur].begin() != cur ) break;
                    for(it=node[cur].begin();it!=node[cur].end();it++) {
                        sn[ *it ] = start;
                        if(cur!=start) node[ start ].push_back ( *it );
                    }
                    if( cur != start ) node[ cur ].clear();
                    cur = pr[ cur ];
                }while( cur != start );
                int best = INT_MAX;
                for( v = node[start].begin(); v!=node[start].end(); v++) {
                    while(!adj[*v].empty()&&sn[adj[*v].back().v]==start)
                        adj[ *v ].pop_back();
                    if( !adj[*v].empty() ) {
                        int tcost = adj[*v].back().w - sub[ *v ];
                        if( tcost < best ) best = tcost, pr[ start ] = adj[*v].back().v;
                    }
                } //assert( best >= 0 && best < INT_MAX );
                cost[ start ] = best;
                for(v=node[start].begin();v!=node[start].end();v++ )sub[*v] += best;
                res += best;
            }
        }
        for(i = 1; i <= n; i++) pr[i] = sn[ pr[i] ];
    }
    return res;
}
```

Bipartite Matching:

```
int adj[MAX][MAX], deg[MAX], Left[MAX], Right[MAX], m, n;
bool visited[MAX];
bool bpm( int u ) {
    for(int i = 0, v; i < deg[u]; i++) {
        v = adj[u][i];
        if( visited[v] ) continue;
        visited[v] = true;
        if( Right[v] == -1 || bpm( Right[v] ) ) {
            Right[v] = u, Left[u] = v;
            return true;
        }
    }
    return false;
}
int bipartiteMatching() { // Returns Maximum Matching
    memset ( Left, -1, sizeof( Left ) );
    memset ( Right, -1, sizeof( Right ) );
    int i, cnt=0;
    for(i=0; i < m; i++) {
        CLR(visited);
        if( bpm( i ) ) cnt++;
    }
    return cnt;
}
```

Erdos and Gallai Theorem:

```
// Input - the deg[] array
bool ErdosGallai() { // 1 indexed
    bool poss = true;
    int i, sum = 0, j, r;
    for(i = 1; i <= n; i++) {
        if( deg[i] >= n ) poss = false;
        sum += deg[i];
    }
    //Summation of degrees has to be ODD and all degrees has to be < n-1
    if( !poss || sum&1 || ( n == 1 && deg[1] > 0 ) ) return false;
    qsort(deg+1, n, sizeof(int), cmp); // in descending order
    degSum[0] = 0;
    j = n;
    for(i=1; i <= n; i++) {
        degSum[i] = degSum[i-1] + deg[i];      //CONSTRUCTING: degSum
        for( ; j >= 1 && deg[j] < i; j-- );  //CONSTRUCTING: ind
        ind[i] = j+1;
    }
    //CONSTRUCTING : minVal
    for(r = 1; r < n; r++) {
        j = ind[r];
        if( j == n+1 ) minVal[r]=( n – r ) * r;
        else if( j <= r ) minVal[r] = degSum[n] - degSum[r];
        else {
            minVal[r] = degSum[n] - degSum[j-1];
            minVal[r] += (j-r-1)*r;
        }
    }
    //Checking  : Erdos & Gallai Theorem
    for( r=1; r < n; r++) if( degSum[r] > ( r*(r-1) + minVal[r] ) ) return false;
    return true;
}
```

Catalan Number

$$C_n = \frac{2n!}{n!(n+1)!} \qquad C_{n+1} = \frac{2(2n+1)}{n+2} C_n$$

Maximum Flow (Dinic) with Min Cut:

```
// cap[][] and Cap[][] both contains the capacity, cap reduces after the flow
int deg[MAX], adj[MAX][MAX], cap[MAX][MAX], Cap[MAX][MAX], q[100000];
int dinic( int n,int s,int t ) {
    int prev[MAX], u, v, i, z, flow = 0, qh, qt, inc;
    while(1) {
        memset( prev, -1, sizeof( prev ) );
        qh = qt = 0;
        prev[s] = -2;
        q[qt++] = s;
        while( qt != qh && prev[t] == -1 ) {
            u = q[qh++];
            for(i = 0; i < deg[u]; i++) {
                v = adj[u][i];
                if( prev[v] == -1 && cap[u][v] ) {
                    prev[v] = u;
                    q[qt++] = v;
                }
            }
        }
        if(prev[t] == -1) break;
        for(z = 0; z < n; z++) if( prev[z] !=- 1 && cap[z][t] ) {
            inc = cap[z][t];
            for(v=z, u=prev[v]; u>=0; v=u, u=prev[v]) inc = min( inc, cap[u][v] );
            if( !inc ) continue;
            cap[z][t] -= inc;
            cap[t][z] += inc;
            for(v=z, u = prev[v]; u >= 0; v = u, u = prev[v]) {
                cap[u][v] -= inc;
                cap[v][u] += inc;
            }
            flow += inc;
        }
    }
    return flow;
}
bool visited[MAX];
void dfs( int u ) {
    visited[u] = true;
    for(int i = 0; i < deg[u]; i++) {
        int v = adj[u][i];
        if( !visited[v] && cap[u][v] && Cap[u][v] ) dfs(v);
    }
}
void printMincut( int s ) {
    CLR( visited );
    dfs( s );
    for(int u = 0; u < n; u++) if( visited[u] )
        for(int v = 0; v < n; v++) if( !visited[v] && !cap[u][v] && Cap[u][v] )
            printf("%d %d\n", u+1, v+1);
}
```

Extended Euclid & GCD:

```
struct Euclid {
    int x, y, d;
    Euclid() {}
    Euclid( int xx, int yy, int dd ) { x = xx, y = yy, d = dd; }
};
int gcd( int a, int b ) { return !b ? a : gcd ( b, a % b ); }
Euclid egcd( int a, int b ) { // Input a, b; Output x, y, d; ax + by = d, d = gcd(a,b)
    if( !b ) return Euclid ( 1, 0, a );
    Euclid r = egcd ( b, a % b );
    return Euclid( r.y, r.x - a / b * r.y, r.d );
}
```

Articulation Point:

```
// result[i] will contain true if the ith node is an articulation
int adj[MAX][MAX], deg[MAX], n, visited[MAX], assignedVal;
bool result[MAX];
int articulation( int u, int depth ) {
    if( visited[u] > 0 ) return visited[u];
    visited[u] = ++assignedVal;
    int mn = visited[u], rootCalled = 0;
    for(int i = 0; i < deg[u]; i++) {
        int v = adj[u][i];
        if( !visited[v] ) {
            if( !depth ) rootCalled++;
            int k = articulation( v, depth+1 );
            if( k >= visited[u] ) result[u] = true;
        }
        mn = min( mn, visited[v] );
    }
    if( !depth ) result[u] = ( rootCalled >= 2 );
    return visited[u] = mn;
}
void processArticulation( int root ) {
    assignedVal = 0;
    CLR( result );
    CLR( visited );
    articulation( root, 0 );
}
```

Euler Phi Function:

```
int phi( int n ) { // Uses Prime Factoring of n
    int factors[NN], factorCount[NN], factorLen;
    findPrimeFactors( n, factors, factorCount, factorLen );
    for( int i = 0; i < factorLen; i++ ) {
        n /= factors[i];
        n *= factors[i] - 1;
    }
    return n;
}
```

Euler Phi Function (Sieve Version):

```
int Phi[MAX];
void sievePHI() { // Phi[i] = phi(i), uses the idea of sieve
    CLR( Phi );
    Phi[1] = 1;
    int i, j;
    for( i = 2; i < MAX; i++ ) if( !Phi[i] ) {
        Phi[i] = i - 1;
        for( j = i + i; j < MAX; j += i ) {
            if( !Phi[j] ) Phi[j] = j;
            Phi[j] = Phi[j] / i * ( i - 1 );
        }
    }
}
```

Prime Factoring of n:

```
// factors[]-contains all factors, factorCount[]-contains frequency, factorLen-length
void findPrimeFactors( int n, int *factors, int *factorCount, int &factorLen ) {
    int i, cnt, sqrtN = ( int ) sqrt ( ( double ) n ) + 1;
    factorLen = 0;
    for( i = 0; pr[i] < sqrtN; i++ ) if( !( n % pr[i] ) ) {
        factors[factorLen] = pr[i], cnt = 0;
        while( !( n % pr[i] ) ) n /= pr[i], cnt++;
        factorCount[factorLen++] = cnt;
        sqrtN = ( int ) sqrt ( ( double ) n ) + 1;
    }
    if( n > 1 ) factors[factorLen] = n, factorCount[factorLen++] = 1;
}
```

Sieve for Finding Primes:

```
// prime upto – PrimeLIMIT, pr[] contains the primes, prlen-length of pr[]
int prime[PrimeLIMIT / 64], pr[MAX_TOTAL], prlen;
#define gP(n) (prime[n>>6]&(1<<((n>>1)&31)))
#define rP(n) (prime[n>>6]&=~(1<<((n>>1)&31)))
void sieve() {
    unsigned int i,j,sqrtN,i2;
    memset( prime, -1, sizeof( prime ) );
    sqrtN = ( int ) sqrt ( ( double ) PrimeLIMIT ) + 1;
    for( i = 3; i < sqrtN; i += 2 ) if( gP( i ) )
        for( j = i * i, i2 = i << 1; j < PrimeLIMIT; j += i2 ) rP( j );
    pr[prlen++] = 2;
    for( i = 3; i < PrimeLIMIT; i += 2 ) if( gP( i ) ) pr[prlen++] = i;
}
```

Divisors of n:

```
// *divisor-divisors list(not sorted), divisorLen-*divisor length, MM should be bigger
void findDivisors( int n, int *divisors, int &divisorLen ) {
    int factors[NN], factorCount[NN], factorLen, st[MM][2], top = 0;
    findPrimeFactors( n, factors, factorCount, factorLen ); // Prime Factoring of n
    divisorLen = 0;
    int result, i, j, k;
    st[top][0] = 1, st[top++][1] = 0;
    while( top-- ) {
        result = st[top][0];
        i = st[top][1];
        if( i == factorLen ) {
            divisors[divisorLen++] = result;
            continue;
        }
        for( j = 0, k = 1; j <= factorCount[i]; j++, k *= factors[i] )
            st[top][0] = result * k, st[top++][1] = i + 1;
    }
}
```

Modular Inverse:

```
int modularInverse( int a, int n ) { // given a and n, returns x, ax mod n = 1
    Euclid t = egcd( a, n );
    if( t.d > 1 ) return 0;
    int r = t.x % n;
    return r < 0 ? r + n : r;
}
```

Modular Linear Equation Solver:

```
// Input – a, b, n; Output – all x in a vector; ax = b (mod n)
vector <int> modularEqnSolver( int a, int b, int n ) {
    Euclid t = egcd( a, n );
    vector <int> r;
    if( b % t.d ) return r;
    int x = ( b / t.d * t.x ) % n;
    if( x < 0 ) x += n;
    for( int i = 0; i < t.d; i++ ) r.push_back( ( x + i * n / t.d ) % n );
    return r;
}
```

Prime Factoring of n!:

```
// factors[]-contains all factors, factorCount[]-contains frequency, factorLen-length
void findPrimeFactorsOfFactorial(int n,int *factors,int *factorCount,int &factorLen) {
    factorLen = 0;
    for( int i = 0; pr[i] <= n; i++ ) {
        int cnt = 0;
        for( int j = pr[i]; j <= n; j *= pr[i] ) cnt += n / j;
        factors[factorLen] = pr[i], factorCount[factorLen++] = cnt;
    }
}
```

Pick's Theorem:

```
// Only for integer points
I = area + 1 – B/2
Where   I = number of points inside
        B = number of points on the border
```

Min Cost Max Flow Optimized (Dijkstra + Johnson):

```
// Input-**cap, **cost; Output-fcost, flow in fnet, fnet[u][v]-fnet[v][u] is net flow
int cap[NN][NN],fnet[NN][NN],adj[NN][NN],deg[NN],pr[NN],d[NN],pi[NN],cost[NN][NN];
#define Pot(u,v) (d[u] + pi[u] - pi[v])
bool dijkstra( int n, int s, int t ) {
    int i;
    for( i = 0; i < n; i++ ) d[i] = inf, pr[i] = -1;
    d[s] = 0;
    pr[s] = -n - 1;
    while( 1 ) {
        int u = -1, bestD = inf;
        for( i = 0; i < n; i++ ) if( pr[i] < 0 && d[i] < bestD ) bestD = d[u = i];
        if( bestD == inf ) break;
        pr[u] = -pr[u] - 1;
        for( i = 0; i < deg[u]; i++ ) {
            int v = adj[u][i];
            if( pr[v] >= 0 ) continue;
            if( fnet[v][u] && d[v] > Pot(u,v) - cost[v][u] )
                d[v] = Pot( u, v ) - cost[v][u], pr[v] = -u - 1;
            if( fnet[u][v] < cap[u][v] && d[v] > Pot(u,v) + cost[u][v] )
                d[v] = Pot(u,v) + cost[u][v], pr[v] = -u - 1;
        }
    }
    for( i = 0; i < n; i++ ) if( pi[i] < inf ) pi[i] += d[i];
    return pr[t] >= 0;
}
int mcmf3( int n, int s, int t, int &fcost ) {
    int u, v, flow = 0;
    fcost = 0;
    CLR( deg );
        for( u = 0; u < n; u++ ) for( v = 0; v < n; v++ ) if( cap[u][v] || cap[v][u] )
        adj[u][ deg[u]++ ] = v;
    CLR( fnet );
    CLR( pi );
    while( dijkstra( n, s, t ) ) {
        int bot = INT_MAX;
        for( v = t, u = pr[v]; v != s; u = pr[v = u] )
            bot = min ( bot, fnet[v][u] ? fnet[v][u] : ( cap[u][v] - fnet[u][v] ) );
        for( v = t, u = pr[v]; v != s; u = pr[v = u] )
            if( fnet[v][u] ) { fnet[v][u] -= bot; fcost -= bot * cost[v][u]; }
            else { fnet[u][v] += bot; fcost += bot * cost[u][v]; }
        flow += bot;
    }
    return flow;
}
```

KMP Matcher(T, P)

```
n = length(T)
m = length(P)
pi = ComputePrefixFunction(P)
q = 0
for i = 1 to n
    while q > 0 and P[q+1] != T[i] do q = pi[q]
    if P[q+1] == T[i] then q = q + 1
    if q == m then print "Pattern occurs with shift i-m"
        q = pi[q]                    //Look for the next match
```

ComputePrefixFunction(P)

```
m = length(P)
pi[1] = k = 0
for q = 2 to m
    while k > 0 and P[k+1] != P[q] do k = pi[k]
    if P[k+1] == P[q] then k = k + 1
    pi[q] = k
return pi
```

Min Cost Max Flow (Bellman Ford):

```
// Input-**cap, **cost; Output-fcost, reduces cap, Works only for directed graphs
// The back edge cost will be the negative of the forward edge cost
int cap[NN][NN], pr[NN], cost[NN][NN], d[NN], adj[NN][NN], deg[NN];
bool bellmanford( int n, int s, int t ) {
    for( int i = 0; i < n; i++ ) d[i] = inf;
    d[s] = 0, pr[s] = -1;
    bool flag = true;
    for( int it = 0; it < n - 1 && flag; it++ ) {
        flag = false;
        for( int u = 0; u < n; u++ )
            for( int i = 0; i < deg[u]; i++ )s {
                int v = adj[u][i];
                if( cap[u][v] && d[v] > d[u] + cost[u][v] )
                    d[v] = d[u] + cost[u][v], pr[v] = u, flag = true;
            }
    }
    return d[t] < inf;
}
int mcmf2( int n, int s, int t, int &fcost ) {
    int netFlow = 0, u, v;
    fcost = 0;
    CLR( deg );
    for( u = 0; u < n; u++ ) for( v = 0; v < n; v++ ) if( cap[u][v] || cap[v][u] )
        adj[u][ deg[u]++ ] = v;
    while( bellmanford( n, s, t ) ) {
        int bot = inf;
        for( v = t; pr[v] != -1; v = pr[v] ) bot = min( bot, cap[ pr[v] ][v] );
        for( v = t; pr[v] != -1; v = pr[v] ) {
            cap[ pr[v] ][v] -= bot;
            cap[v][ pr[v] ] += bot;
            fcost += bot * cost[ pr[v] ][v];
        }
        netFlow += bot;
    }
    return netFlow;
}
```

Least Common Ancestor (LCA):

```
// N is the number of nodes, T[i] contains the parent of i, calculates P[][]
// First cal preprocessLCA, then run the queryLCA for each query
void preprocessLCA( int N, int T[MAXN], int P[MAXN][LOGMAXN] ) { // 0 indexed
    int i, j;
    //we initialize every element in P with -1
    for( i = 0; i < N; i++ ) for( j = 0; 1 << j < N; j++ ) P[i][j] = -1;
    for( i = 0; i < N; i++ ) P[i][0] = T[i]; //the first ancestor of i is T[i]
    for( j = 1; 1 << j < N; j++ ) for( i = 0; i < N; i++ ) //bottom up dp
        if( P[i][j - 1] != -1 ) P[i][j] = P[P[i][j - 1]][j - 1];
}
// L is the depth/level array, should be precalculated
int queryLCA( int N, int P[MAXN][LOGMAXN], int T[MAXN], int L[MAXN], int p, int q ) {
    int tmp, log, i;
    //if p is situated on a higher level than q then we swap them
    if( L[p] < L[q] ) tmp = p, p = q, q = tmp;
    for( log = 1; 1 << log <= L[p]; log++ ); //we compute the value of [log(L[p)]
    log--;
    //we find the ancestor of p situated on the same level with q using the values in P
    for( i = log; i >= 0; i-- ) if( L[p] - (1 << i) >= L[q] ) p = P[p][i];
    if( p == q ) return p;
    //we compute LCA(p, q) using the values in P
    for( i = log; i >= 0; i-- ) if(P[p][i] != -1 && P[p][i] != P[q][i] )
        p = P[p][i], q = P[q][i];
    return T[p];
}
```

Weighted Bipartite Matching O(n^3):

```
// Take input in cost[][]
#define N 55
#define INF 100000000

int cost[N][N], n, max_match;
int lx[N], ly[N];
int xy[N], yx[N];
bool S[N], T[N];
int slack[N], slackx[N], prev[N];

void init_labels() {
    memset( lx, 0, sizeof(lx) );
    memset( ly, 0, sizeof(ly) );
    for( int x = 0; x < n; x++ ) for( int y = 0; y < n; y++ ) lx[x] = max(lx[x],
cost[x][y]);
}
void update_labels() {
    int x, y, delta = INF;
    for (y = 0; y < n; y++) if (!T[y]) delta = min(delta, slack[y]);
    for (x = 0; x < n; x++) if (S[x]) lx[x] -= delta;
    for (y = 0; y < n; y++) if (T[y]) ly[y] += delta;
    for (y = 0; y < n; y++) if (!T[y]) slack[y] -= delta;
}
void add_to_tree( int x, int prevx ) {
    S[x] = true;
    prev[x] = prevx;
    for (int y = 0; y < n; y++)
        if (lx[x] + ly[y] - cost[x][y] < slack[y]) {
            slack[y] = lx[x] + ly[y] - cost[x][y];
            slackx[y] = x;
        }
}
void augment() {
    if( max_match == n ) return;
    int x, y, root;
    int q[N], wr = 0, rd = 0;
    memset(S, false, sizeof(S));
    memset(T, false, sizeof(T));
    memset(prev, -1, sizeof(prev));
    for( x = 0; x < n; x++ ) if (xy[x] == -1) {
        q[wr++] = root = x;
        prev[x] = -2;
        S[x] = true;
        break;
    }
    for( y = 0; y < n; y++ ) {
        slack[y] = lx[root] + ly[y] - cost[root][y];
        slackx[y] = root;
    }
    while( true ) {
        while( rd < wr ) {
            x = q[rd++];
            for( y = 0; y < n; y++ )
                if( cost[x][y] == lx[x] + ly[y] && !T[y] ) {
                    if( yx[y] == -1 ) break;
                    T[y] = true;
                    q[wr++] = yx[y];
                    add_to_tree( yx[y], x);
                }
            if(y < n) break;
        }
        if(y < n) break;
        update_labels();
        wr = rd = 0;
```

```
        for(y = 0; y < n; y++) if(!T[y] &&  slack[y] == 0) {
            if(yx[y] == -1) {
                x = slackx[y];
                break;
            }
            else {
                T[y] = true;
                if (!S[yx[y]]) {
                    q[wr++] = yx[y];
                    add_to_tree(yx[y], slackx[y]);
                }
            }
        }
        if(y < n) break;
    }
    if(y < n) {
        max_match++;
        for( int cx = x, cy = y, ty; cx != -2; cx = prev[cx], cy = ty ) {
            ty = xy[cx];
            yx[cy] = cx;
            xy[cx] = cy;
        }
        augment();
    }
}
int hungarian() {
    int ret = 0;
    max_match = 0;
    memset(xy, -1, sizeof(xy));
    memset(yx, -1, sizeof(yx));
    init_labels();
    augment();
    for(int x = 0; x < n; x++) ret += cost[x][xy[x]];
    return ret;
}
```

Gaussian Elimination:

```
void gauss( int N, long double mat[NN][NN] ) {
    int i, j, k;
    for (i = 0; i < N; i++) {
        k = i;
        for (j = i+1; j < N; j++) if (fabs(mat[j][i]) > fabs(mat[k][i])) k = j;
        if (k != i) for (j = 0; j <= N; j++) swap(mat[k][j], mat[i][j]);
        for (j = i+1; j <= N; j++) mat[i][j] /= mat[i][i];
        mat[i][i] = 1;
        for (k = 0; k < N; k++) if( k != i ) {
            long double t = mat[k][i];
            if (t == 0.0L) continue;
            for (j = i; j <= N; j++) mat[k][j] -= t * mat[i][j];
            mat[k][i] = 0.0L;
        }
    }
}
```

Segmented Sieve:

```
void segmented_sieve( int l , int h ) {
    int i , j , k , m , end ;
    double L = (double) l ;
    memset ( composite , 0 , sizeof ( composite ) ) ;
    end = ceil ( sqrt ( h ) ) ;
    for ( i=3 ; i<end ; i+=2 ) {
        if ( !COMPS[i] ){
            j = ceil ( L / i ) ;     k = h / i ; m = i*j-l ;
            for ( j , m ; j<=k ; j++ , m+=i )   composite[m] = 1 ;
        }
    }
}
```

Union of Rectangles in O(nlogn):

```
struct Axis {
    int value, type, id;
};
bool cmp( Axis A, Axis B ) {
    if( A.value < B.value ) return 1;
    if( A.value == B.value && A.type < B.type ) return 1;
    return 0;
}
struct Node {
    int low, high, left, right;
    int range, count;
};
struct Rectangle {
    int size, lx[MAX], hx[MAX], ly[MAX], hy[MAX], Hsize, Vsize, avail, root, INF_COUNT;
    Axis H[ MAX << 1 ], V[ MAX << 1 ];
    Node node[ 1<<16 ];
    Rectangle() { INF_COUNT = 1000000; }
    void INIT() {
        int i, j;
        for( j = i = 0; i < size; i++ ) {
            H[j].value = ly[i]; H[j].type = 0; H[j].id = i;
            V[j].value = lx[i]; V[j].type = 0; V[j++].id = i;
            H[j].value = hy[i]; H[j].type = 1; H[j].id = i;
            V[j].value = hx[i]; V[j].type = 1; V[j++].id = i;
        }
        sort(H, H+j, cmp);
        sort(V, V+j, cmp);
        Hsize = Vsize = j;
        for( j = i = 0; i < Hsize; i++ ) {
            if( H[j].value == H[i].value ) continue;
            H[++j] = H[i];
        }
        while( j & ( j - 1 ) ) H[j + 1] = H[j++];
        Hsize = j + 1;
    }
    int MYTREE( int from, int to ) {
        int here = avail++;
        node[here].low = H[ from - 1 ].value;
        node[here].high = H[to].value;
        node[here].range = 0;
        node[here].count = 0;
        if( from == to ) {
            if( node[here].low == node[here].high ) node[here].count = INF_COUNT;
            node[here].left = node[here].right=-1;
            return here;
        }
        node[here].left = MYTREE( from,( from + to - 1 ) >> 1 );
        node[here].right = MYTREE( ( ( from + to - 1 ) >> 1 ) + 1, to );
        return here;
    }
    void INSERT( int low, int high, int at ) {
        if( node[at].low == low && node[at].high == high) {
            node[at].count++;
            node[at].range = high - low;
            return;
        }
        if(node[node[at].left].high > low)
            INSERT( low, min(node[node[at].left].high, high), node[at].left );
        if( node[node[at].right].low < high)
            INSERT( max( low, node[node[at].right].low ), high, node[at].right );
        if( !node[at].count )
            node[at].range=node[node[at].left].range + node[node[at].right].range;
    }
```

```
    void REMOVE( int low, int high, int at) {
        if(node[at].low == low && node[at].high == high) {
            node[at].count--;
            if( !node[at].count ) {
                if(node[at].left==-1) node[at].range=0;
                else
                   node[at].range=node[node[at].left].range+node[node[at].right].range;
            }
            return;
        }
        if(node[node[at].left].high>low)
            REMOVE(low,min(node[node[at].left].high,high),node[at].left);
        if(node[node[at].right].low<high)
            REMOVE(max(low,node[node[at].right].low),high,node[at].right);
        if(node[at].count==0)
            node[at].range=node[node[at].left].range + node[node[at].right].range;
    }
    int area() {
        if( !size ) return 0;
        int ans = 0, now, prev, current;
        INIT();
        avail = 0;
        root = MYTREE(1, Hsize-1 );
        now = 0;
        prev = V[0].value;
        while( now < Vsize ) {
            current = V[now].value;
            ans += node[root].range*(current-prev);
            prev = current;
            for( ; V[now].value == current && now < Vsize; now++ ) {
                if( V[now].type == 0 ) INSERT( ly[V[now].id], hy[V[now].id], root);
                else REMOVE( ly[V[now].id], hy[V[ now].id ], root);
            }
        }
        return ans;
    }
}R;
int main() {
    int n, i;
    scanf("%d", &n);
    R.size = n;
    for( i = 0 ; i < n; i++ ) scanf("%d%d%d%d",&R.lx[i], &R.ly[i], &R.hx[i], &R.hy[i]);
    printf("%d\n", R.area());
    return 0;
}
```

Fitting A Rectangle Inside Another Rectangle:

```
// Checks whether ractangle with sides (a, b) fits into rectangle with sides (c, d)
bool fits( int a, int b, int c, int d ) {
    double X, Y, L, K, DMax;
    if( a < b ) swap( a, b );
    if( c < d ) swap( c, d );
    if( c <= a && d <= b ) return true;
    if( d >= b ) return false;
    X = sqrt( a*a + b*b );
    Y = sqrt( c*c + d*d );
    if( Y < b ) return true;
    if( Y > X ) return false;
    L = ( b - sqrt( Y*Y - a*a) ) /2;
    K = ( a - sqrt( Y*Y - b*b) ) /2;
    DMax = sqrt(L * L + K * K);
    if( d >= DMax ) return false;
return true;
}
```

Misc Geometric Formula:

Triangle	Circum Radius = a*b*c/(4*area) In Radius = area/s, where s = (a+b+c)/2 length of median to side c = sqrt(2*(a*a+b*b)-c*c)/2 length of bisector of angle C = sqrt(ab[(a+b)*(a+b)-c*c])/(a+b)
Ellipse	Area = PI*a*b Circumference = 4a *int(0,PI/2){sqrt(1-(k*sint)*(k*sint))}dt = 2*PI*sqrt((a*a+b*b)/2) approx where k = sqrt((a*a-b*b)/a) = PI*(3*(r1+r2)-sqrt[(r1+3*r2)*(3*r1+r2)])
Spherical cap	V = (1/3)*PI*h*h*(3*r-h) Surface Area = 2*PI*r*h
Spherical Sector	V = (2/3)*PI*r*r*h
Spherical Segment	V = (1/6)*PI*h*(3*a*a+3*b*b+h*h)
Torus	V = 2*PI*PI*R*r*r
Truncated Conic	V = (1/3)*PI*h*(a*a+a*b+b*b) Surface Area = PI*(a+b)*sqrt(h*h+(b-a)*(b-a)) = PI*(a+b)*l
Pyramidal frustum	(1/3)*h*(A1+A2+sqrt(A1*A2))

Misc Trigonometric Functions and Formulas:

```
tan A/2 = ±sqrt((1-cos A)/(1+cos A))
        = sin A / (1+cos A)
        = (1-cos A) / sin A
        = cosec A – cot A
sin 3A  = 3*sin A – 4*sincube A      cos 3A  = 4*coscube A – 3*cos A
tan 3A  = (3*tan A-tancube A)/(1-3*tansq A)
sin 4A  = 4*sin A*cos A – 8*sincube A*cos A
cos 4A  = 8*cos^4 A – 8*cossq A + 1
[r*(cost+i*sint)]^p = r^p*(cos pt+i*sin pt)
acosx + bsinx = c, x = 2nπ + α ± β, where
cosα = a / (sqrt(a^2+b^2)), cosβ = c / (sqrt(a^2+b^2));

2sinAcosB = sin(A+B) + sin(A-B)
2cosAsinB = sin(A+B) - sin(A-B)
2cosAcosB = cos(A-B) + cos(A+B)
2sinAsinB = cos(A-B) – cos(A+B)
sinC + sinD = 2sin[(C+D)/2]cos[(C-D)/2]
sinC - sinD = 2cos[(C+D)/2]sin[(C-D)/2]
cosD + cosC = 2cos[(C+D)/2]cos[(C-D)/2]
cosD - cosC = 2sin[(C+D)/2]sin[(C-D)/2]
```

Misc Integration Formula:

```
a^x => a^x/ln(a)
1/sqrt(x*x+a*a) => ln(x+sqrt(x*x+a*a))
1/sqrt(x*x-a*a) => ln(x+sqrt(x*x-a*a))
1/(x*sqrt(x*x+a*a) => -(1/a)*ln([a+sqrt(x*x+a*a)]/x)
1/(x*sqrt(a*a-x*x) => -(1/a)*ln([a+sqrt(a*a-x*x)]/x)
```

Misc Differentiation Formula:

```
asin x => 1/sqrt(1-x*x)      acos x => -1/sqrt(1-x*x)
atan x => 1/(1+x*x)          acot x => -1/(1+x*x)
asec x => 1/[x*sqrt(x*x-1)] acosec x => -1/[x*sqrt(x*x-1)]
a^x => a^x*ln(x)                 cot x => -cosecsq x
sec x => sec x * tan x           cosec x => -cosec x * cot x
```

Centroid of a 2D polygon:

As in the calculation of the area above, xN is assumed to be x0, in other words the polygon is closed.

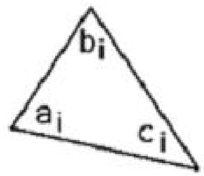
$$c_x = \frac{1}{6A} \sum_{i=0}^{N-1} (x_i + x_{i+1})(x_i y_{i+1} - x_{i+1} y_i)$$

$$c_y = \frac{1}{6A} \sum_{i=0}^{N-1} (y_i + y_{i+1})(x_i y_{i+1} - x_{i+1} y_i)$$

Centroid of a 3D shell described by 3 vertex facets:

The centroid C of a 3D object made up of a collection of N triangular faces with vertices (a_i,b_i,c_i) is given below. R_i is the average of the vertices of the i'th face and Ai is twice the area of the i'th face. Note the faces are assumed to be thin sheets of uniform mass, they need not be connected or form a solid object.

$$C = \frac{\sum_{i=0}^{N-1} A_i R_i}{\sum_{i=0}^{N-1} A_i}$$



$$R_i = (a_i + b_i + c_i) / 3$$

$$A_i = \|(b_i - a_i) \otimes (c_i - a_i)\|$$

Mirror point(mx,my) of a point(x,y) w.r.to a line(ax+by+c=0):

```
void mirrorPoint(double a,double b,double c,double x,double y,double &mx,double &my) {
    mx = - x*(a*a-b*b) - 2.0*a*b*y - 2.0*a*c;    mx /= (a*a+b*b);
    my = y*(a*a-b*b) - 2.0*a*b*x - 2.0*b*c; my /= (a*a+b*b);
}
```

Circum Circle:

```
R = abc / (4*area)
//measuring the Circum_center M(x,y):
k1 = A.x*A.x - B.x*B.x + A.y*A.y - B.y*B.y;
k2 = A.x*A.x - C.x*C.x + A.y*A.y - C.y*C.y;
k3 = (A.x*C.y + B.x*A.y + C.x*B.y) - (C.x*A.y + A.x*B.y + B.x*C.y);
M.x = (k2*(A.y-B.y) - k1*(A.y-C.y))/(2.*k3);
M.y = (k1*(A.x-C.x) - k2*(A.x-B.x))/(2.*k3);
```

In Circle:

```
// The triangle consists of points A, B and C
r   = area / s
I.x = (A.x*a + B.x*b + C.x) / (a+b+c)
I.y = (A.y*a + B.y*b + C.y) / (a+b+c)
```

Great circle Distance Between 2 points given in Longitude/latitude format [Radius = R]

```
haversine(x) = ( 1 – cos(x) )/2.0;
a = haversine(lat2 - lat1)
b = cos(lat1) * cos(lat2) * haversine(lon2 - lon1)
c = 2 * atan2(sqrt(a + b), sqrt(1 - a - b))
d = R * c
```

Determining if a point lies on the interior of a polygon:

Solution fore 2D:

```
struct Point{ int h,v; } Point;
int InsidePolygon(Point *polygon,int n,Point p) {
    int i; double angle=0;  Point p1,p2;
    for (i=0;i<n;i++){
        p1.h = polygon[i].h - p.h;
        p1.v = polygon[i].v - p.v;
        p2.h = polygon[(i+1)%n].h - p.h;
        p2.v = polygon[(i+1)%n].v - p.v;
        angle += Angle2D(p1.h,p1.v,p2.h,p2.v);
    }
    if (ABS(angle) < PI) return(FALSE);
    else return(TRUE);
}
// Returns the angle between two vectors on a plane, from vector 1 to vector 2
// positive anticlockwise, result is in[-pi, pi]
double Angle2D( double x1, double y1, double x2, double y2 ){
    double dtheta, theta1, theta2;
    theta1 = atan2(y1,x1);  theta2 = atan2(y2,x2);
    dtheta = theta2 – theta1;
    while (dtheta > PI) dtheta -= TWOPI;
    while (dtheta < -PI) dtheta += TWOPI;
    return (dtheta);
}
```

Solution for 3D:

```
// To determine whether a point is on the interior of a convex polygon in 3D, one
// might be tempted to first determine whether the point is on the plane, then
// determine its interior status. Both of these can be accomplished at once by
// computing the sum of the angles between the test point (q below) and every pair of
// edge points p[i]->p[i+1]. This sum will only be twopi if both the point is on the
// plane of the polygon AND on the interior. The angle sum will tend to 0 the further
// away from the polygon point q becomes. The following code snippet returns the angle
// sum between the test point q and all the vertex pairs. The angle sum is in radians.
typedef struct{ double x,y,z; }XYZ;
#define EPSILON 0.0000001
#define MODULUS(p) (sqrt(p.x*p.x + p.y*p.y + p.z*p.z))
const double TWOPI = 6.283185307179586476925287, RTOD = 57.2957795;
double CalcAngleSum(XYZ q,XYZ *p,int n) {
    double m1,m2,anglesum=0,costheta;
    XYZ p1,p2;
    for(int i=0;i<n;i++){
        p1.x = p[i].x - q.x;
        p1.y = p[i].y - q.y;
        p1.z = p[i].z - q.z;
        p2.x = p[(i+1)%n].x - q.x;
        p2.y = p[(i+1)%n].y - q.y;
        p2.z = p[(i+1)%n].z - q.z;
        m1 = MODULUS(p1), m2 = MODULUS(p2);
        if(m1*m2 <= EPSILON) return(TWOPI); // We are on a node, consider this inside
        else costheta = (p1.x*p2.x + p1.y*p2.y + p1.z*p2.z) / (m1*m2);
        anglesum += acos(costheta);
    }
    return(anglesum);
}
```

Rotation Matrices:

$$Q_{\mathbf{x}}(\theta)=\begin{bmatrix}1 & 0 & 0\\ 0 & \cos\theta & -\sin\theta\\ 0 & \sin\theta & \cos\theta\end{bmatrix},\ Q_{\mathbf{y}}(\theta)=\begin{bmatrix}\cos\theta & 0 & \sin\theta\\ 0 & 1 & 0\\ -\sin\theta & 0 & \cos\theta\end{bmatrix},\ Q_{\mathbf{z}}(\theta)=\begin{bmatrix}\cos\theta & -\sin\theta & 0\\ \sin\theta & \cos\theta & 0\\ 0 & 0 & 1\end{bmatrix},\ Q_{2\times 2}=\begin{bmatrix}\cos\theta & -\sin\theta\\ \sin\theta & \cos\theta\end{bmatrix},$$

Construct n from the Sum of Its Divisors:

```
// powi64(a, b) computes a^b, rememver that prime upto i-1 are used
i64 table[NN+1][NN+1]; // if there is an overflow, table[i][j] = inf;
void preprocessTable() {
    for( int i = 0; i <= NN; i++ ) table[0][i] = 1;
    for( int i = 1; i <= NN; i++ ) {
        table[i][0] = 1;
        for( int j = 1; j < NN; j++ ) table[i][j] = table[i][j-1] + powi64(pr[i-1], j);
    }
}
vector <i64> calculateXFromSumOfDivisors( int sum ) {
    vector <i64> res;
    i64 val = 1, prevD = 1;
    for( int i = NN; ; i-- ) {
        if( sum == 1 ) {
            res.push_back( val ); // Here the value is saved
            sum *= prevD, val = 1;
        }
        if( i <= 0 || sum == 1 ) break;
        for( int j = NN - 1; j >= 0; j-- ) {
            if( table[i][j] > 1 && ( sum % table[i][j] == 0 ) ) {
                val *= powi64( pr[i-1], j );
                sum /= table[i][j], prevD = table[i][j];
                break;
            }
        }
    }
    return res;
}
```

Misc Geometry:

```
#define LD double
const LD eps = 1e-11;
const LD pi = 2*acos(0.0);
#define dot(u,v) ((u).x * (v).x + (u).y * (v).y + (u).z * (v).z)
#define norm(v) sqrt(dot(v,v))  // norm = length of vector
#define dis(u,v) norm(u-v)       // distance = norm of difference
struct Point { // Creates normal 2D Point
    LD x, y;
    Point() {}
    Point( LD xx, LD yy ) { x = xx, y = yy; }
};
struct Point3D { // Creates normal 3D Point
    LD x, y, z;
};
struct polygon { // Creates a Polygon with 2D Points

    int n;
    Point P[PolygonSize];
};
struct line { // Creates a line with equation ax + by + c = 0
    LD a, b, c;
    line() {}
    line( Point p1,Point p2 ) {
        a=p1.y-p2.y;
        b=p2.x-p1.x;
        c=p1.x*p2.y-p2.x*p1.y;
    }
};
struct circle { // Creates a circle with Point 'center' as center and r as radius
    Point center;
    LD r;
    circle() {}
    circle( Point P, LD rr ) { center = P; r = rr; }
};
struct segment { // Creates a segment with two end Points -> A, B
    Point A, B;
    segment() {}
    segment( Point P1, Point P2 ) { A = P1, B = P2; }
};
inline bool eq(LD a, LD b) { return fabs( a - b ) < eps; } //if real numbers are equal
```

Distance - Point, Point:

```
inline LD distance( Point a, Point b ) {
    return sqrt( ( a.x - b.x ) * ( a.x - b.x ) + ( a.y - b.y ) * ( a.y - b.y ) );
}
```

Distance^2 - Point, Point:

```
inline LD sq_distance(Point a,Point b) {
    return ( a.x - b.x ) * ( a.x - b.x ) + ( a.y - b.y ) * ( a.y - b.y );
}
```

Distance - Point, Line:

```
inline LD distance( Point P, line L ) {
    return fabs( L.a * P.x + L.b * P.y + L.c ) / sqrt( L.a * L.a + L.b * L.b );
}
// the 3rd point is (left, colinear, right) to first 2 if return value (>0, =0, <0)
```

Is left Function:

```
inline LD isleft( Point p0, Point p1, Point p2 ) {
    return( ( p1.x - p0.x ) * ( p2.y - p0.y ) - ( p2.x - p0.x ) * ( p1.y - p0.y ) );
}
```

Intersection - Line, Line:

```
inline bool intersection( line L1, line L2, Point &p ) {
    LD det = L1.a * L2.b - L1.b * L2.a;
    if( eq ( det, 0 ) ) return false;
    p.x = ( L1.b * L2.c - L2.b * L1.c ) / det;
    p.y = ( L1.c * L2.a - L2.c * L1.a ) / det;
    return true;
}
```

Intersection - Segment, Segment:

```
inline bool intersection( segment L1, segment L2, Point &p ) {
    if( !intersection( line( L1.A, L1.B ), line( L2.A, L2.B ), p) ) {
        // 1 segment can lie on another, just check their equations, and check overlap
        return false;
    }
    return(eq(distance(L1.A,p)+distance(L1.B,p),distance(L1.A,L1.B)) &&
        eq(distance(L2.A,p)+distance(L2.B,p),distance(L2.A,L2.B)));
}
```

Distance - Point, Segment:

```
inline LD distance( Point P, segment S ) {
    line L1, L2;
    Point P1;
    L2 = findPerpendicularLine( L1, P );
    if( intersection( L1, L2, P1 ) )
        if( eq ( distance( S.A, P1 ) + distance( S.B, P1 ), distance( S.A, S.B ) ) )
            return distance(P,L1);
    return min ( distance( S.A, P), distance( S.B, P) );
}
```

Intersection - Circle, Line:

```
inline bool intersection(circle C,line L,Point &p1,Point &p2) {
    if( distance( C.center, L ) > C.r + eps ) return false;
    LD a, b, c, d, x = C.center.x, y = C.center.y;
    d = C.r*C.r - x*x - y*y;
    if( eq( L.a, 0) ) {
        p1.y = p2.y = -L.c / L.b;
        a = 1;
        b = 2 * x;
        c = p1.y * p1.y - 2 * p1.y * y - d;
        d = b * b - 4 * a * c;
        d = sqrt( fabs (d) );
        p1.x = ( b + d ) / ( 2 * a );
        p2.x = ( b - d ) / ( 2 * a );
    }
    else {
        a = L.a * L.a + L.b * L.b;
        b = 2 * ( L.a * L.a * y - L.b * L.c - L.a * L.b * x);
        c = L.c * L.c + 2 * L.a * L.c * x - L.a * L.a * d;
        d = b * b - 4 * a * c;
        d = sqrt( fabs(d) );
        p1.y = ( b + d ) / ( 2 * a );
        p2.y = ( b - d ) / ( 2 * a );
        p1.x = ( -L.b * p1.y -L.c ) / L.a;
        p2.x = ( -L.b * p2.y -L.c ) / L.a;
    }
    return true;
}
```

Perpendicular Line of a Given Line Through a Point:

```
inline line findPerpendicularLine( line L, Point P ) {
    line res;
    res.a = L.b, res.b = -L.a;
    res.c = -res.a * P.x - res.b * P.y;
    return res;
}
```

Find Points that are r1 unit away from A, and r2 unit away from B:

```
inline bool findPointAr1Br2( Point A, LD r1, Point B, LD r2, Point &p1, Point &p2 ) {
    line L;
    circle C;
    L.a = 2 * (B.x - A.x );
    L.b = 2 * (B.y - A.y );
    L.c = A.x * A.x + A.y * A.y - B.x * B.x - B.y * B.y + r2 * r2 - r1 * r1;
    C.center = A;
    C.r = r1;
    return intersection( C, L, p1, p2 );
}
```

Convex Hull (Graham Scan) O(nlogn):

```
// compare Function for qsort in convex hull
Point FirstPoint;
int cmp(const void *a,const void *b) {
    LD x,y;
    Point aa,bb;
    aa = *(Point *)a;
    bb = *(Point *)b;
    x = isleft( FirstPoint, aa, bb );
    if( x > eps ) return -1;
    else if( x < -eps ) return 1;
    x = sq_distance( FirstPoint, aa );
    y = sq_distance( FirstPoint, bb );
    if( x + eps < y ) return -1;
    return 1;
}
// 'P' contains all the Points, 'C' contains the convex hull
// 'nP' = total Points of 'P', 'nC' = total Points of 'C'
void ConvexHull( Point P[], Point C[], int &nP, int &nC ) {
    int i, j, pos = 0;
    for( i = 1; i < nP; i++ )
        if(P[i].y < P[pos].y || ( eq( P[i].y, P[pos].y ) && P[i].x > P[pos].x + eps ))
            pos = i;
    // Remove duplicate Points if necesary
    swap( P[pos], P[0] );
    FirstPoint = P[0];
    qsort( P + 1, nP - 1, sizeof( Point ), cmp );
    C[0] = P[0];
    C[1] = P[1];
    i = 2, j = 1;
    while(i<nP) {
        if( isleft( C[j-1], C[j], P[i] ) > -eps ) C[++j] = P[i++];
        else j--;
    }
    nC = j + 1;
    C[nC] = C[0];
}
```

Rotating a Point anticlockwise by 'theta' radian w.r.t Origin:

```
inline Point rotate2D( LD theta, Point P ) {
    Point Q;
    Q.x = P.x * cos( theta ) - P.y * sin( theta );
    Q.y = P.x * sin( theta ) + P.y * cos( theta );
    return Q;
}
```

Circle Through Thee Points:

```
circle CircleThrough3Points( Point A, Point B, Point C ) {
    LD den; circle c;
    den = 2.0 *((B.x-A.x)*(C.y-A.y) - (B.y-A.y)*(C.x-A.x));
    c.center.x = ( (C.y-A.y) * (B.x*B.x+B.y*B.y-A.x*A.x-A.y*A.y) -
(B.y-A.y) * (C.x*C.x+C.y*C.y-A.x*A.x-A.y*A.y) );
    c.center.x /= den;
    c.center.y = ( (B.x-A.x)*(C.x*C.x+C.y*C.y-A.x*A.x-A.y*A.y) –
(C.x-A.x) * (B.x*B.x+B.y*B.y-A.x*A.x-A.y*A.y) );
    c.center.y /= den;
    c.r = distance( c.center, A );
    return c;
}
```

Area of a 2D Polygon:

```
LD areaPolygon( Point P[], int nP ) {
    LD area=0;
    P[nP] = P[0];
    for( int i = 0; i < nP; i++ ) area += P[i].x * P[i+1].y - P[i].y * P[i+1].x;
    return fabs( area ) / 2.0;
}
```

Point Inside Polygon:

```
// Finds a Point outside the polygon for which no vertex, p and the Point are colinear
void findXPoint( Point p, polygon A, Point &XPoint ) {
    XPoint.x = 11000; XPoint.y = 11001;
    while(1) {
        for( int i=0; i < A.n; i++ ) if( eq( isleft( A.P[i], p, XPoint ), 0 ) ) break;
        if( i == A.n ) return;
        XPoint.y++;
    }
}
bool PointInsidePolygon( Point p, polygon A ) {
    int count = 0;
    Point temp, XPoint;
    findXPoint( p, A, XPoint );
    A.P[A.n] = A.P[0];
    for( int i = 0; i < A.n; i++ )
        if( intersection(segment(p, XPoint),segment(A.P[i],A.P[i+1]), temp) ) count++;
    return count&1;
}
```

Intersection Area between Two Circles:

```
inline LD intersectionArea2C( circle C1, circle C2 ) {
    C2.center.x = distance( C1.center, C2.center );
    C1.center.x = C1.center.y = C2.center.y = 0;
    if( C1.r < C2.center.x - C2.r + eps ) return 0;
    if( -C1.r + eps > C2.center.x - C2.r ) return pi * C1.r * C1.r;
    if( C1.r + eps > C2.center.x + C2.r ) return pi * C2.r * C2.r;
    LD c = C2.center.x, CAD, CBD, res;
    CAD = 2 * acos( (C1.r * C1.r + c * c - C2.r * C2.r) / (2 * C1.r * c) );
    CBD = 2 * acos( (C2.r * C2.r + c * c - C1.r * C1.r) / (2 * C2.r * c) );
    res=C1.r * C1.r * ( CAD - sin( CAD ) ) + C2.r * C2.r * ( CBD - sin ( CBD ) );
    return .5 * res;
}
```

Area of a 3D Polygon:

```
LD area3D_Polygon( int n, Point3D *V ) {
    LD area = 0, val, i, j, k, an, ax, ay, az; // abs value of normal and its coords
    int coord; // coord to ignore: 1=x, 2=y, 3=z
    Point3D u, v, N;     // Find Unit Normal Vector in next step
    u.x = V[1].x - V[0].x; u.y = V[1].y - V[0].y; u.z = V[1].z - V[0].z;
    v.x = V[2].x - V[0].x; v.y = V[2].y - V[0].y; v.z = V[2].z - V[0].z;
    N.x = u.y * v.z- u.z * v.y;
    N.y = u.z * v.x- u.x * v.z;
    N.z = u.x * v.y- u.y * v.x;
    val = sqrt( N.x * N.x + N.y * N.y + N.z * N.z );
    N.x /= val; N.y /= val; N.z /= val;
    // select largest abs coordinate to ignore for projection
    ax = (N.x > 0 ? N.x : -N.x);    // abs x-coord
    ay = (N.y > 0 ? N.y : -N.y);    // abs y-coord
    az = (N.z > 0 ? N.z : -N.z);    // abs z-coord
    coord = 3;                      // ignore z-coord
    if(ax > ay) { if (ax > az) coord = 1;   }   // ignore x-coord
    else if(ay > az) coord = 2;              // ignore y-coord
    for( i = 1, j = 2, k = 0; i <= n; i++, j++, k++) { // area 2D projection
        switch (coord) {
            case 1: area += (V[i].y * (V[j].z - V[k].z)); continue;
            case 2: area += (V[i].x * (V[j].z - V[k].z)); continue;
            case 3: area += (V[i].x * (V[j].y - V[k].y)); continue;
        }
    }// scale to get area before projection
    an = sqrt( ax*ax + ay*ay + az*az);  // length of normal vector
    switch (coord) {
        case 1: area *= (an / (2*ax)); break;
        case 2: area *= (an / (2*ay)); break;
        case 3: area *= (an / (2*az));
    }
    return fabs( area );
}
```

Angle between Vectors:

```
inline LD angleBetweenVectors( Point O, Point A, Point B) {
    Point t1, t2;
    t1.x = A.x - O.x; t1.y = A.y - O.y;
    t2.x = B.x - O.x; t2.y = B.y - O.y;
    LD theta = (atan2(t2.y, t2.x) - atan2(t1.y, t1.x));
    if( theta < 0 ) theta += 2 * pi;
    return theta;
}
```

Printing a Polynomial:

```
/* Takes a list of coefficients, a largest power and a variable name
 * And prints the polynomial to stdout in the form of, e.g 4*x^3+8*n-9 */
void printPoly( int coeffs[], int n, const char *var ) {
    bool empty = true;
    while( n-- ) {
        if( coeffs[n] || empty && !n ) {
            if(!empty || coeffs[n]<0 ) printf("%c",(coeffs[n]>0?'+' : '-' ) );
            if( abs( coeffs[n] ) > 1 || n == 0 ) printf( "%d", abs( coeffs[n] ) );
            if( n > 0 ) {
                if( abs( coeffs[n] ) > 1 ) printf( "*" );
                printf( "%s", var );
            }
            if( n > 1 ) printf( "^%d", n );
            empty = false;
        }
    }
}
```

Closest Pair Problem:

```
// p contains the point, s1, and s1 are auxiliary arrays
Point p[100], s1[100], s2[100];
bool sortX(Point &a, Point &b) { return ( a.x == b.x ) ? a.y < b.y : a.x < b.x; }
bool sortY(Point &a, Point &b) { return ( a.y == b.y ) ? a.x < b.x : a.y < b.y; }
double closestPair( int k1, int k2 ){
    double d, d2 ,d3;
    if(k2-k1+1 == 1) return 0;
    if(k2-k1+1 == 2) return distance(p[k1], p[k2]);
    if(k2-k1+1 == 3) {
        d  = distance( p[k1], p[k1+1] );
        d2 = distance( p[k1+1], p[k1+2]);
        d3 = distance( p[k1+2], p[k1]);
        return min( min(d, d2), d3 );
    }
    int k, i, j, ns1, ns2;
    k = (k1 + k2) / 2;
    d  = closestPair(k1 , k);
    d2 = closestPair(k+1 , k2);
    if(d > d2)  d = d2;
    ns1 = ns2 = 0;
    for(i = k; i>=k1 ; i--) {
        if( p[k].x - p[i].x  >  d ) break;
        s1[ ns1++ ] = p[i];
    }
    sort(s1, s1+ns1, sortY);
    for(i = k+1; i<=k2 ; i++) {
        if( p[i].x - p[k].x  >  d ) break;
        s2[ ns2++ ] = p[i];
    }
    sort(s2, s2+ns2, sortY);
    for(i=0;i<ns1;i++) {
        for(j=0;j<ns2;j++) {
            if(s2[j].y - s1[i].y > d) break;
            d = min( d, distance( s1[i], s2[j] ) );
        }
    }
    return d;
}
```

```
int main() {
    //take n, points in p
    sort( p, p+n, sortX );
    d = closestPair(0,n-1);
    return 0;
}
```

Finding Determinant:

```
/* We have found the Minimum col in 1st row.
Then subTract from All nonZero column the minimum Column as possible.
Such as: 5 8 7 - 3 2 Then in next Step in stead of 5 we start with 3 because Modulus
must less than divider(5). */
#define MAX 50
#define INF 32000
int mat[MAX][MAX], N, mul;
void xchgColumn( int i, int j ) {
    for( int k = 0; k < N; k++ ) swap( mat[k][i], mat[k][j] );
    mul *= -1;
}
void reduceMat(void){
    int i, j, minCol, min, absMin, nonZero = 0, absValue, d, r;
    for(absMin=INF,i=0;i<N;i++){
        if(mat[0][i]){
            nonZero++;
            if(mat[0][i] < 0) absValue = -mat[0][i];
            else absValue = mat[0][i];
            if(absValue < absMin){
                absMin = absValue;
                min    = mat[0][i];
                minCol = i;
            }
        }
    }
    if(!nonZero) { mul = 0; return; }
    while(nonZero > 1){
        for(i=0;i<N;i++){
            if(i != minCol && mat[0][i]){
                d = mat[0][i]/min;  r = mat[0][i]-d*min;
                for(j=0;j<N;j++) mat[j][i]=mat[j][i]-d*mat[j][minCol];
                if(r){ min = r; minCol = i; }
                else nonZero--;
            }
        }
    }
    for(i=0;!mat[0][i];i++);
    if(i!=0) xchgColumn(0,i);
    mul *= mat[0][0];
    for(i=1;i<N;i++) for(j=1;j<N;j++) mat[i-1][j-1] = mat[i][j];
    N--;
}
int main() {
    int i,j,result;
    while(scanf("%d",&N) && N){
        for(i=0;i<N;i++)
            for(j=0;j<N;j++)
                scanf("%d",&mat[i][j]);
        if(N > 1){
            mul = 1;
            while(N > 2 && mul) reduceMat();
            result = mat[0][0]*mat[1][1]-mat[0][1]*mat[1][0];
            result = result*mul;
            printf("%d\n",result);
        }
        else printf("%d\n",mat[0][0]);
    }
    return 0;
}
```

Aho Corasick:

```
#define MM 100005                        // The length of the long string
#define NN 1005                          // The length of the small string
#define MAX 200001                       // The maximum number of nodes the trie can have
#define MAXCHAR 52                       // The maximum number of characters allowed
char T[MM];                              // Long string
char a[NN][NN];                          // Small String
int val( char ch ) {
    if( ch >= 'a' ) return ch - 97;
    return ch - 39;
}
struct Trie {
    int N;               // Contains the number of nodes of the Trie
    struct node {
        int edge[MAXCHAR], f;   // The alphabets, f failure function
        bool out; // out function, gives the set of patterns recognized entering this state
        void clear() {          // Clears the node
            memset( edge, -1, sizeof( edge ) );
            f = out = false;
        }
    }P[MAX];
    void clear() {       // Clears the Trie, Initially g( 0, x ) = 0, for all x
        N = 1; P[0].clear();
        memset( P[0].edge, 0, sizeof( P[0].edge ) );
    }
    void insert( char *a ) {             // Inserts an element into the trie
        int p = 0, i, k;
        for( i = 0; a[i]; i++ ) {
            k = val( a[i] );
            if( P[p].edge[k] <= 0 ) {    // Edge is not available
                P[p].edge[k] = N;
                P[N++].clear();          // Clear the edge, and increase N
            }
            p = P[p].edge[k];            // Go to the next edge
        }
        P[p].out = true;
    }
    void computeFailure() {              // Computes the failure function
        int i, u, v, w;
        queue <int> Q;
        for( i = 0; i < MAXCHAR; i++ ) if( P[0].edge[i] ) {
            u = P[0].edge[i];
            P[u].f = 0;
            Q.push(u);
        }
        while( !Q.empty() ) {
            u = Q.front(); Q.pop();
            for( i = 0; i < MAXCHAR; i++ ) if( P[u].edge[i] != -1 ) {
                v = P[u].edge[i];
                Q.push(v);
                w = P[u].f;
                while( P[w].edge[i] == -1 ) w = P[w].f;
                w = P[v].f = P[w].edge[i];
                P[v].out |= P[w].out;
            }
        }
    }
    void print() {
        for( int i = 0; i < N; i++ ) {
            printf("%d ->\n",  i);
            for( int j = 0; j < MAXCHAR; j++ ) if( P[i].edge[j] > 0 )
                printf("%c - %d\n", j + 97, P[i].edge[j]);
            printf("Failure %d\n\n", P[i].f);
        }
    }
}A;
```

```
int n, cases;
bool mark[MAX];
void AhoCorasick( Trie &A, char *T ) {// Finds the occurences of strings in the trie, in T
    int i, q = 0, k; // q Initial State
    for( i = 0; i < A.N; i++ ) mark[i] = false;
    for( i = 0; T[i]; i++ ) {
        k = val( T[i] );
        while( A.P[q].edge[k] == -1 ) q = A.P[q].f;
        q = A.P[q].edge[k];
        int x = q;
        if( A.P[x].out && !mark[x] ) {
            mark[x] = true;
            x = A.P[x].f;
        }
    }
}
bool exists( Trie &A, char *a ) {
    int i, q = 0, k;
    for( i = 0; a[i]; i++ ) {
        k = val(a[i]);
        q = A.P[q].edge[k];
    }
    return mark[q];
}
int main() {
    scanf("%d", &cases);
    while( cases-- ) {
        scanf("%s %d", T, &n);
        A.clear();
        for( int i = 0; i < n; i++ ) {
            scanf("%s", a[i]);
            A.insert( a[i] );
        }
        A.computeFailure();
        AhoCorasick( A, T );
        for( int i = 0; i < n; i++ ) {
            if( exists( A, a[i] ) ) puts("y");
            else puts("n");
        }
    }
    return 0;
}
```

To Roman:

```
string toRoman( int n ) {
    if( n < 4 ) return fill( 'i', n );
    if( n < 6 ) return fill( 'i', 5 - n ) + "v";
    if( n < 9 ) return string( "v" ) + fill( 'i', n - 5 );
    if( n < 11 ) return fill( 'i', 10 - n ) + "x";
    if( n < 40 ) return fill( 'x', n / 10 ) + toRoman( n % 10 );
    if( n < 60 ) return fill( 'x', 5 - n / 10 ) + 'l' + toRoman( n % 10 );
    if( n < 90 ) return string( "l" )+fill('x',n/10-5) + toRoman( n % 10 );
    if( n < 110 ) return fill( 'x', 10 - n / 10 ) + "c" + toRoman( n % 10 );
    if( n < 400 ) return fill( 'c', n / 100 ) + toRoman( n % 100 );
    if( n < 600 ) return fill( 'c', 5 - n / 100 ) + 'd' + toRoman( n % 100 );
    if( n < 900 ) return string("d")+fill('c',n/100- 5 ) + toRoman( n % 100 );
    if( n < 1100 ) return fill( 'c', 10 - n / 100 ) + "m" + toRoman( n % 100 );
    if( n < 4000 ) return fill( 'm', n / 1000 ) + toRoman( n % 1000 );
    return "?";
}
```

Binary Indexed Tree:

```
int bit[M],n,m;
void update(int x, int v) {
    while( x <= n ) {
        bit[x] += v;
        x += x & -x;
    }
}
int sum( int x ) {
    int ret = 0;
    while( x > 0 ){
        ret += bit[x];
        x -= x & -x;
    }
    return ret;
}
```

Template:

```
#include<cstdio>
#include<sstream>
#include<cstdlib>
#include<cctype>
#include<cmath>
#include<algorithm>
#include<set>
#include<queue>
#include<stack>
#include<list>
#include<iostream>
#include<fstream>
#include<numeric>
#include<string>
#include<vector>
#include<cstring>
#include<map>
#include<iterator>
using namespace std;

#define FORIT(i, m) for (__typeof((m).begin()) i=(m).begin(); i!=(m).end(); ++i)
#define REP(i,n) for(int i=0; i<(n); i++)
#define PER(i,n) for(int i=(n)-1; i>=0; i--)
#define FOR(i,a,b) for(__typeof(b) i=(a); i<=(b); i++)
#define ROF(i,a,b) for(__typeof(b) i=(a); i>=(b); i--)
#define sz size()
#define pb push_back
#define MP make_pair
#define ALL(x) x.begin(), x.end()
#define VI vector<int>
#define VS vector<string>
#define I64 long long
#define SET(t,v) memset((t), (v), sizeof(t))
#define REV(x) reverse( ALL( x ) )
#define INF (1<<29)
#define eps (1e-11)
```