ACM ICPC Reference

Federal University of Campina Grande

May 14, 2015

Contents

1 Math

Cubic function roots

```
// Finds the roots of a cubic function in the complex: ax^3+bx^2+cx+d=0
// Requires a != 0 and (b != 0 or c != 0)
// Running time: O(1), huge constant

#define cprint(n) printf("%.6lf + %.6lfi\n", (n).real(), (n).imag())
const complex<double> cbrt2 = pow(2, 1.0/3);
const complex<double> isqrt3 = complex<double>(0.0,sqrt(3.0));
complex<double> a, b, c, d, alfa, beta, delta, root1, root2, root3;
void calc_dga() {
  alfa = (-27.0*a*a*d+9.0*a*b*c-2.0*b*b*b), beta = (3.0*a*c-b*b);
  delta = pow(sqrt(alfa*alfa+4.0*beta*beta*beta) + alfa, 1.0/3);
```

```
  root1 = delta/(3.0*cbrt2*a) - cbrt2*beta/(3.0*a*delta) - b/(3.0*a);
  root2 = -1.0/(6.0*cbrt2*a)*(1.0-isqrt3)*delta+(1.0+isqrt3)*beta/
    (3.0*cbrt2*cbrt2*a*delta)-b/(3.0*a);
  root3 = -1.0/(6.0*cbrt2*a)*(1.0+isqrt3)*delta+(1.0-isqrt3)*beta/
    (3.0*cbrt2*cbrt2*a*delta)-b/(3.0*a);
}
```

Fibonacci

```
// Calculates fibonacci numbers.
// Running time: O(__builtin_popcount(n-1)) = O(log n)

#define LOGMAXN 63
long long f0, f1, faux;
long long fib2[LOGMAXN+1][2]={{0,1}}; // fib2[i]={Fib(n-1),Fib(n)}, n=2^i
template<int MOD> void generate_fib2() {
  for (int i = 1; i <= LOGMAXN; i++) {
    fib2[i][1] = (fib2[i-1][1]*(((fib2[i-1][0]<<1)+fib2[i-1][1])))%MOD;
    fib2[i][0] = (fib2[i][1]-(fib2[i-1][0]*(((fib2[i-1][1]<<1)-
      fib2[i-1][0]+MOD)%MOD))%MOD+MOD)%MOD;
  }
}
template<int MOD> inline long long fib(long long n) { // {0,1,1,2,...}
  if (!fib2[1][0]) generate_fib2<MOD>();
  if (!n--) return 0;
  f0 = 0, f1 = 1;
  while (n) {
    int i = __builtin_ctzll(n);
    faux = (f1*fib2[i][1] + f0*fib2[i][0]) % MOD;
    f1 = (f1*(fib2[i][0]+fib2[i][1]) + f0*fib2[i][1]) % MOD;
    f0 = faux;
    n -= 1ULL<<i;
  }
  return f1;
}
int main() {generate_fib2<1000000007>(); printf("%lld",fib<1000000007>(100));}

// Some identities:
//   F(n+1)F(n-1) - F(n)^2 = -1^n
//   F(n+k) = F(k)F(n+1) + F(k-1)F(n)
//   F(2n-1) = F(n)^2 + F(n-1)^2
//   SUM(i=0 to n)[F(i)] = F(n+2) - 1
//   SUM(i=0 to n)[F(i)^2] = F(n)F(n+1)
//   SUM(i=0 to n)[F(i)^3] = [F(n)F(n+1)^2 - (-1^n)F(n-1) + 1] / 2
//   gcd(Fm, Fn) = F(gcd(m,n))
//   sqrt(5N^2 +- 4) is natural <-> exists natural k | F(k) = N
//   [ F(0) F(1) ] [ [0 1] [1 1] ]^n = [ F(n) F(n+1) ]
// Binet's formula:
//   g = (1 + sqrt(5)) / 2
//   Fn = g^n / sqrt(5)
//   n(F) = floor(log[g](sqrt(5)F + 1/2)), where log[g] = log base g

// First 40 fibonacci numbers
//    n F(n) |  n  F(n)  | n    F(n)  |  n      F(n)  |  n       F(n)
//    0    0 |  8    21  | 16    987  | 24     46368  | 32    2178309
//    1    1 |  9    34  | 17   1597  | 25     75025  | 33    3524578
//    2    1 | 10    55  | 18   2584  | 26    121393  | 34    5702887
//    3    2 | 11    89  | 19   4181  | 27    196418  | 35    9227465
//    4    3 | 12   144  | 20   6765  | 28    317811  | 36   14930352
//    5    5 | 13   233  | 21  10946  | 29    514229  | 37   24157817
//    6    8 | 14   377  | 22  17711  | 30    832040  | 38   39088169
//    7   13 | 15   610  | 23  28657  | 31   1346269  | 39   63245986
```

Number theoretic algorithms (modular, Chinese remainder, linear Diophantine)

```
// This is a collection of useful code for solving problems that
// involve modular linear equations.  Note that all of the
// algorithms described here work on nonnegative integers.

typedef vector<int> VI;
typedef pair<int,int> PII;

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

int gcd(int a, int b) {
  int tmp;
  while(b){a%=b; tmp=a; a=b; b=tmp;}
  return a;
}
#define lcm(a,b) 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)
VI modular_linear_equation_solver(int a, int b, int n) {
  int x, y;
  VI 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);
}

// Chinese remainder theorem (special case): 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 = a[i] (mod x[i]) for all i.
// Note that the solution is unique modulo M = lcm_i (x[i]). Return (z,M). On
// failure, M = -1. Note we do not require the a[i]'s to be relatively prime.
PII chinese_remainder_theorem(const VI &x, const VI &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;
  }
}

// computes the number of coprimes of p^k, being p prime
//int phi(int p, int k) { return pow(p, k) - pow(p, k-1); } // phi(p^k)
int phi(int p, int pk) { return pk - (pk/p); } // phi(p^k), where pk=p^k
// computes the number of coprimes of n
int phi(int n) {
  int coprimes = (n != 1); // phi(1) = 0
  for (int i = 2; i*i <= n; i++)
    if (n%i == 0) {
      int pk = 1;
      while (n%i == 0)
        n /= i, pk *= i;
      coprimes *= phi(i, pk);
    }
  if (n > 1) coprimes *= phi(n, n); // n is prime
  return coprimes;
}

int main() {
  // expected: 2 8
  cout << gcd(14, 30) << " " << mod_inverse(8, 9) << endl;

  // expected: 2 -2 1
  int x, y, d = extended_euclid(14, 30, x, y);
  cout << d << " " << x << " " << y << endl;

  // expected: 23 105
  //           11 12
```

```
  int xs[] = {3, 5, 7, 4, 6}, as[] = {2, 3, 2, 3, 5};
  PII ret = chinese_remainder_theorem(VI (xs, xs+3), VI(as, as+3));
  cout << ret.first << "␣" << ret.second << endl;
  ret = chinese_remainder_theorem (VI(xs+3, xs+5), VI(as+3, as+5));
  cout << ret.first << "␣" << ret.second << endl;

  // expected: 95 45
  VI sols = modular_linear_equation_solver(14, 30, 100);
  for (int i = 0; i < sols.size(); i++) cout << sols[i] << "␣" << endl;

  // expected: 5 -15
  linear_diophantine(7, 2, 5, x, y);
  cout << x << "␣" << y << endl;
}
```

Primes

```
// Sieve of Eratosthenes: finds all primes up to N.
// Running time: O(N log log N)

#define MAXP 100000000 // ADJUST!
bool is_prime[MAXP+1]={0,0,1};
template<int N> void sieve() {
  for (int i = 3; i <= N; i += 2)
    is_prime[i] = true;
  for (int i = 3; i*i <= N; i += 2)
    if (is_prime[i])
      for (int j = i*i; j <= N; j += 2*i)
        is_prime[j] = false;
}

// Miller-Rabin Primality Test: tests whether a number is prime or not.
// Running time: O(k log n) for primes, huge constant

ulint llrand() { ulint a = rand(); a<<= 32; a+= rand(); return a; }
ulint mul_mod(ulint a, ulint b, ulint mod) { return a*b%mod; } //(a%m+m)%m*b%m
ulint exp_mod(ulint a, ulint e, ulint mod) {
  if (e == 0) return 1;
  ulint b = exp_mod(a,e/2,mod);
  return (e % 2 == 0) ? mul_mod(b,b,mod) : mul_mod(mul_mod(b,b,mod),a,mod);
}
int is_probably_prime(ulint n, int k=64) {
  if (n <= 1) return 0;
  if (n <= 3) return 1;
  ulint s = 0, d = n - 1;
  while ((d&1) == 0)
    d /= 2, s++;
  while (k--) {
    ulint a = (llrand() % (n - 3)) + 2;
    ulint x = exp_mod(a, d, n);
    if (x != 1 && x != n-1) {
      for (int r = 1; r < s; r++) {
        x = mul_mod(x, x, n);
        if (x == 1) return 0;
        if (x == n-1) break;
      }
      if (x != n-1) return 0;
    }
  }
  return 1;
}

// Primes less than 1000:
//      2      3      5      7     11     13     17     19     23     29     31     37
//     41     43     47     53     59     61     67     71     73     79     83     89
//     97    101    103    107    109    113    127    131    137    139    149    151
//    157    163    167    173    179    181    191    193    197    199    211    223
//    227    229    233    239    241    251    257    263    269    271    277    281
//    283    293    307    311    313    317    331    337    347    349    353    359
//    367    373    379    383    389    397    401    409    419    421    431    433
//    439    443    449    457    461    463    467    479    487    491    499    503
//    509    521    523    541    547    557    563    569    571    577    587    593
//    599    601    607    613    617    619    631    641    643    647    653    659
//    661    673    677    683    691    701    709    719    727    733    739    743
//    751    757    761    769    773    787    797    809    811    821    823    827
//    829    839    853    857    859    863    877    881    883    887    907    911
//    919    929    937    941    947    953    967    971    977    983    991    997

// Other primes (primes immediatly less than 10^x):
//  x       p   x           p   x                p   x                    p
//  1       7   6      999983  11      99999999977  16      9999999999999937
//  2      97   7     9999991  12     999999999989  17     99999999999999997
//  3     997   8    99999989  13    9999999999971  18    999999999999999989
//  4    9973   9   999999937  14   99999999999973  19   9999999999999999961
//  5   99991  10  9999999967  15  999999999999989  20  99999999999999999989
```

Reduced row echelon form, matrix rank

```
// Reduced row echelon form via Gauss-Jordan elimination
// with partial pivoting. This can be used for computing
// the rank of a matrix.
// Running time: O(n^3)
//
// INPUT:    a[][] = an nxm matrix
//
// OUTPUT:   rref[][] = an nxm matrix (stored in a[][])
//           returns rank of a[][]

const double EPSILON = 1e-10;
typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;

int rref(VVT &a) {
  int n = a.size(), m = a[0].size(), r = 0;
  for (int c = 0; c < m && r < n; c++) {
    int j = r;
    for (int i = r+1; i < n; i++)
      if (fabs(a[i][c]) > fabs(a[j][c])) j = i;
    if (fabs(a[j][c]) < EPSILON) continue;
    swap(a[j], a[r]);
    T s = 1.0 / a[r][c];
    for (int j = 0; j < m; j++) a[r][j] *= s;
    for (int i = 0; i < n; i++) if (i != r) {
      T t = a[i][c];
      for (int j = 0; j < m; j++) a[i][j] -= t * a[r][j];
    }
    r++;
  }
  return r;
}

int main() {
  const int n = 5, m = 4; VVT a(n);
  double A[n][m]=
    {{16,2,3,13},{5,11,10,8},{9,7,6,12},{4,14,15,1},{13,21,21,13}};
  for (int i = 0; i < n; i++) a[i] = VT(A[i], A[i] + n);

  int rank = rref(a); cout << "Rank: " << rank << endl; // expected: 4

  // expected: 1 0 0 1/0 1 0 3/0 0 1 -3/0 0 0 2.78206e-15/0 0 0 3.22398e-15
  cout << "rref: " << endl;
  for (int i = 0; i < 5; i++) {
    for (int j = 0; j < 4; j++) cout << a[i][j] << ' ';
    cout << endl;
  }
}
```

Simplex algorithm

```
// Two-phase simplex algorithm for solving linear programs of the form
//     maximize     c^T x
//     subject to   Ax <= b, xi >= 0
//
// INPUT: A -- an m x n matrix
//        b -- an m-dimensional vector
//        c -- an n-dimensional vector
//        x -- a vector where the optimal solution will be stored
//
// OUTPUT: value of the optimal solution (infinity if unbounded
//         above, nan if infeasible)
//
// To use this code, create an LPSolver object with A, b, and c as
// arguments. Then, call Solve(x).

typedef long double DOUBLE;
typedef vector<DOUBLE> VD;
typedef vector<VD> VVD;
typedef vector<int> VI;

const DOUBLE EPS = 1e-9;

struct LPSolver {
  int m, n;
  VI B, N;
  VVD D;

  LPSolver(const VVD &A, const VD &b, const VD &c) :
    m(b.size()), n(c.size()), N(n+1), B(m), D(m+2, VD(n+2)) {
    for (int i = 0; i < m; i++)
      for (int j = 0; j < n; j++) D[i][j]=A[i][j];
    for (int i = 0; i < m; i++) B[i]=n+i, D[i][n]=-1, D[i][n+1]=b[i];
    for (int j = 0; j < n; j++) { N[j] = j; D[m][j] = -c[j]; }
    N[n] = -1; D[m+1][n] = 1;
  }
```

```
void Pivot(int r, int s) {
  for (int i = 0; i < m+2; i++) if (i != r)
  for (int j = 0; j < n+2; j++) if (j != s)
  D[i][j] -= D[r][j] * D[i][s] / D[r][s];
  for (int j = 0; j < n+2; j++) if (j != s) D[r][j] /= D[r][s];
  for (int i = 0; i < m+2; i++) 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;
    for (int j = 0; j <= n; j++) {
      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;
    for (int i = 0; i < m; i++) {
      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(VD &x) {
  int r = 0;
  for (int i = 1; i < m; i++) 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();
    for (int i = 0; i < m; i++) if (B[i] == -1) {
      int s = -1;
      for (int j = 0; j <= n; j++)
      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 = VD(n);
  for (int i = 0; i < m; i++) if (B[i] < n) x[B[i]] = D[i][n+1];
  return D[m][n+1];
 }
};

int main() {
  const int m = 4, n = 3;
  DOUBLE _A[m][n] = {{6, -1, 0}, {-1, -5, 0}, {1, 5, 1}, {-1, -5, -1}};
  DOUBLE _b[m] = {10, -4, 5, -5}, _c[n] = {1, -1, 0};

  VVD A(m); VD b(_b, _b + m), c(_c, _c + n);
  for (int i = 0; i < m; i++) A[i] = VD(_A[i], _A[i] + n);
  LPSolver solver(A, b, c); VD x; DOUBLE value = solver.Solve(x);

  // expected: VALUE: 1.29032/SOLUTION: 1.74194 0.451613 1
  cerr << "VALUE: "<< value << endl; cerr << "SOLUTION:";
  for (size_t i = 0; i < x.size(); i++) cerr << " " << x[i] << endl;
}
```

Systems of linear equations, matrix inverse, determinant

```
// Gauss-Jordan elimination with full pivoting.
//
// Uses:
//   (1) solving systems of linear equations (AX=B)
//   (2) inverting matrices (AX=I)
//   (3) computing determinants of square matrices
//
// Running time: O(n^3)
//
// INPUT:    a[][] = an nxn matrix
//           b[][] = an nxm matrix
//
// OUTPUT:   X      = an nxm matrix (stored in b[][])
//           A^{-1} = an nxn matrix (stored in a[][])
//           returns determinant of a[][]

const double EPS = 1e-10;
```

```
typedef vector<int> VI;
typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;

T GaussJordan(VVT &a, VVT &b) {
  const int n = a.size(), m = b[0].size();
  VI irow(n), icol(n), ipiv(n);
  T det = 1;

  for (int i = 0; i < n; i++) {
    int pj = -1, pk = -1;
    for (int j = 0; j < n; j++) if (!ipiv[j])
      for (int k = 0; k < n; k++) 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;
    for (int p = 0; p < n; p++) a[pk][p] *= c;
    for (int p = 0; p < m; p++) b[pk][p] *= c;
    for (int p = 0; p < n; p++) if (p != pk) {
      c = a[p][pk];
      a[p][pk] = 0;
      for (int q = 0; q < n; q++) a[p][q] -= a[pk][q] * c;
      for (int q = 0; q < m; q++) b[p][q] -= b[pk][q] * c;
    }
  }
  for (int p = n-1; p >= 0; p--) if (irow[p] != icol[p])
    for (int k = 0; k < n; k++) swap(a[k][irow[p]], a[k][icol[p]]);
  return det;
}

int main() {
  const int n = 4, m = 2;
  double A[n][n] = { {1,2,3,4},{1,0,1,0},{5,3,2,4},{6,1,4,6} };
  double B[n][m] = { {1,2},{4,3},{5,6},{8,7} };
  VVT a(n), b(n);
  for (int i = 0; i < n; i++)
    a[i] = VT(A[i], A[i] + n), b[i] = VT(B[i], B[i] + m);

  double det = GaussJordan(a, b);
  cout << "Determinant: " << det << endl; // expected: 60

  // expected: -0.233333 0.166667 0.133333 0.0666667
  //           0.166667 0.166667 0.333333 -0.333333
  //           0.233333 0.833333 -0.133333 -0.0666667
  //           0.05 -0.75 -0.1 0.2
  cout << "Inverse: " << endl;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++)
      cout << a[i][j] << ' ';
    cout << endl;
  }

  // expected: 1.63333 1.3
  //           -0.166667 0.5
  //           2.36667 1.7
  //           -1.85 -1.35
  cout << "Solution: " << endl;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++)
      cout << b[i][j] << ' ';
    cout << endl;
  }
}
```

Brent's Algorithm (Cycle detection)

Let $x_0 \in S$ be an element of the finite set S and consider a function $f : S \to S$. Define

$$f_k(x) = \begin{cases} x, & k = 0 \\ f\left(f_{k-1}(x)\right), & k > 0 \end{cases}.$$

Clearly, there exists distinct numbers $i, j \in \mathbb{N}$, $i \neq j$, such that $f_i(x_0) = f_j(x_0)$.

Let $\mu \in \mathbb{N}$ be the least value such that there exists $j \in \mathbb{N} \setminus \{\mu\}$ such that $f_\mu(x_0) = f_j(x_0)$ and let $\lambda \in \mathbb{N}$ be the least value such that $f_\mu(x_0) = f_{\mu+\lambda}(x_0)$.

Given x_0 and f, this code computes μ and λ applying the operator f $\mathcal{O}(\mu + \lambda)$ times and storing at most a constant amount of elements from S.

```
p = l = 1, t = x0, h = f(x0);
while (t != h) {
  if (p == l) t = h, p*= 2, l = 0;
  h = f(h), ++l;
```

```
}
u = 0, t = h = x0;
for (i = 1; i != 0; --i) h = f(h);
while (t != h) t = f(t), h = f(h), ++u;
```

2 Counting

Catalan Numbers

C_n is:

- The number of balanced expressions built from n pairs of parentheses.
- The number of paths in an $n \times n$ grid that stays on or below the diagonal.
- The number of words of size $2n$ over the alphabet $\Sigma = \{a, b\}$ having an equal number of a symbols and b symbols containing no prefix with more a symbols than b symbols.

It holds that:

$$C_0 = 1, C_{n+1} = \sum_{k=0}^{n} C_k C_{n-k}$$

$$C_n = \binom{2n}{n} - \binom{2n}{n-1} = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{n!(n+1)!}$$

Stirling Numbers of the First Kind

$\begin{bmatrix} n \\ k \end{bmatrix}$ is:

- The number of ways to split n elements into k ordered partitions up to a permutation of the partitions among themselves and rotations within the partitions.
- The number of digraphs with n vertices and k cycles such that each vertex has in and out degree of 1.

It holds that:

$$\begin{bmatrix} n \\ 0 \end{bmatrix} = \begin{cases} 1, & n = 0 \\ 0, & n \neq 0 \end{cases}, \quad \begin{bmatrix} 0 \\ k \end{bmatrix} = \begin{cases} 1, & k = 0 \\ 0, & k \neq 0 \end{cases}$$

$$\begin{bmatrix} n \\ k \end{bmatrix} = (n-1)\begin{bmatrix} n-1 \\ k \end{bmatrix} + \begin{bmatrix} n-1 \\ k-1 \end{bmatrix}$$

$$\begin{bmatrix} n \\ 1 \end{bmatrix} = (n-1)!$$

$$\begin{bmatrix} n \\ n-1 \end{bmatrix} = \binom{n}{2}$$

$$\begin{bmatrix} n \\ n-2 \end{bmatrix} = \frac{1}{4}(3n-1)\binom{n}{3}$$

$$\begin{bmatrix} n \\ n-3 \end{bmatrix} = \binom{n}{2}\binom{n}{4}$$

$$\begin{bmatrix} n \\ 2 \end{bmatrix} = (n-1)! H_{n-1}$$

$$\begin{bmatrix} n \\ 3 \end{bmatrix} = \frac{1}{2}(n-1)!\left(H_{n-1}^2 - H_{n-1}^{(2)}\right)$$

$$H_n = \sum_{j=1}^{n} \frac{1}{j}, \quad H_n^{(k)} = \sum_{j=1}^{n} \frac{1}{j^k}$$

$$\sum_{k=0}^{n} \begin{bmatrix} n \\ k \end{bmatrix} = n!$$

$$\sum_{j=k}^{n} \begin{bmatrix} n \\ j \end{bmatrix} \binom{j}{k} = \begin{bmatrix} n+1 \\ k+1 \end{bmatrix}$$

Stirling Numbers of the Second Kind

$\begin{Bmatrix} n \\ k \end{Bmatrix}$ is the number of ways to partition an n-set into exactly k non-empty disjoint subsets up to a permutation of the sets among themselves. It holds that:

$$\begin{Bmatrix} n \\ 0 \end{Bmatrix} = \begin{cases} 1, & n = 0 \\ 0, & n \neq 0 \end{cases}, \quad \begin{Bmatrix} n \\ 1 \end{Bmatrix} = \begin{Bmatrix} n \\ n \end{Bmatrix} = 1$$

$$\begin{Bmatrix} n \\ k \end{Bmatrix} = \begin{Bmatrix} n-1 \\ k-1 \end{Bmatrix} + k\begin{Bmatrix} n-1 \\ k \end{Bmatrix}$$

$$\begin{Bmatrix} n \\ k \end{Bmatrix} \bmod 2 = \begin{cases} 0, & (n-k) \& \left\lfloor \frac{k-1}{2} \right\rfloor \neq 0 \\ 1, & \text{otherwise} \end{cases},$$

where & is the C bitwise "and" operator.

$$\begin{Bmatrix} n \\ 2 \end{Bmatrix} = 2^{n-1} - 1$$

$$\begin{Bmatrix} n \\ n-1 \end{Bmatrix} = \binom{n}{2}$$

$$\begin{Bmatrix} n \\ k \end{Bmatrix} = \frac{1}{k!}\sum_{j=0}^{k} (-1)^{k-j}\binom{k}{j} j^n$$

Bell Numbers

$\mathcal{B}_n$ is the number of equivalence relations on an n-set or, alternatively, the number of partitions of an n-set. It holds that:

$$\mathcal{B}_n = \sum_{k=0}^{n} \begin{Bmatrix} n \\ k \end{Bmatrix}$$

$$\mathcal{B}_{n+1} = \sum_{k=0}^{n} \binom{n}{k} \mathcal{B}_k$$
$$\mathcal{B}_n = \frac{1}{e} \sum_{k=0}^{\infty} \frac{k^n}{k!}$$
$$\mathcal{B}_{n+p} \equiv \mathcal{B}_n + \mathcal{B}_{n+1} \pmod p$$

The Twelvefold Way

Let A be a set of m balls and B be a set of n boxes. The following table provides methods to compute the number of equivalent functions $f : A \to B$ satisfying specific constraints.

Balls	Boxes	Any	Injective	Surjective
$\not\equiv$	$\not\equiv$	n^m	$\frac{n!}{(n-m)!}$	$n!\left\{ m \atop n \right\}$
$\not\equiv$	$\equiv$	$\sum_{k=0}^{n} \left\{ m \atop k \right\}$	$\delta_{m \leqslant n}$	$\left\{ m \atop n \right\}$
$\equiv$	$\not\equiv$	$\binom{m+n-1}{m}$	$\binom{n}{m}$	$\binom{m-1}{n-1}$
$\equiv$	$\equiv$	**(*)** $\sum_{k=0}^{n} p(m,k)$	$\delta_{m \leqslant n}$	**(**)** $p(m,n)$

(**) is a definition and both **(*)** and **(**)** are very hard to compute. So do not try to.

Lucca's Theorem

Let $n, k, p \in \mathbb{N}$ and p be a prime number. Then

$$\binom{n}{k} \equiv \prod_{j=0}^{\infty} \binom{n_j}{k_j} \pmod p,$$

where n_j and k_j are the j-th digits of the numbers n and k in base p, respectively.

Derangement (Desarranjo)

A derangement is a permutation of the elements of a set such that none of the elements appear in their original position.

Suppose that there are n persons numbered $1, 2, \ldots, n$. Let there be n hats also numbered $1, 2, \ldots, n$. We have to find the number of ways in which no one gets the hat having same number as his/her number. Let us assume that first person takes the hat i. There are $n-1$ ways for the first person to choose the number i. Now there are 2 options:

- Person i takes the hat of 1. Now the problem reduces to $n-2$ persons and $n-2$ hats.
- Person i does not take the hat 1. This case is equivalent to solving the problem with $n-1$ persons $n-1$ hats (each of the remaining $n-1$ people has precisely 1 forbidden choice from among the remaining $n-1$ hats).

From this, the following relation is derived:

$$d_n = (n-1) * (d_{n-1} + d_{n-2})$$
$$d_1 = 0$$
$$d_2 = 1$$

Starting with $n = 0$, the numbers of derangements of n are: 1, 0, 1, 2, 9, 44, 265, 1854, 14833, 133496, 1334961, 14684570, 176214841, 2290792932.

3 Geometry

Convex–hull

```
// Calculates the convex hull of a given vector of points.
// Running time: O(n log n) or, if already sorted, O(n)

typedef pair<int,int> Point;
int cross(Point a, Point b) {return a.first*b.second-a.second*b.first;}
int cross(Point O, Point a, Point b) {
  return cross(Point(a.first-O.first,a.second-O.second),
    Point(b.first-O.first,b.second-O.second));
}
template<int M> void findPoints(vector<Point>& points, vector<Point>& res) {
  for (int i = 0; i < points.size(); i++) {
    Point& p = points[i];
    while (res.size()>=2 && M*cross(res.end()[-2],res.end()[-1],p)>=0)
      res.pop_back(); // > 0 instead of >= 0 keeps collinear points
    res.push_back(p);
  }
}
// USAGE: convexHull(inputPoints, outputPolygon)
void convexHull(vector<Point>& points, vector<Point>& result) {
  vector<Point> lowerResult;
  sort(points.begin(), points.end()); // remove if already sorted
  findPoints<1>(points, result);
  findPoints<-1>(points, lowerResult);
  for (int i = lowerResult.size()-2; i; i--)
    result.push_back(lowerResult[i]);
}
```

Miscellaneous geometry

```
// C++ routines for computational geometry.

double INF = 1e100, EPS = 1e-12;

struct PT {
  double x, y;
  PT() {}
  PT(double x, double y) : x(x), y(y) {}
  PT(const PT &p) : x(p.x), y(p.y) {}
  PT operator + (const PT &p) const { return PT(x+p.x, y+p.y); }
  PT operator - (const PT &p) const { return PT(x-p.x, y-p.y); }
  PT operator * (double c) const { return PT(x*c, y*c); }
  PT operator / (double c) const { return PT(x/c, y/c); }
};
ostream& operator<<(ostream& os, const PT& p) {
    return os << "(" << p.x << "," << p.y << ")";
}
double dot(PT p, PT q)   { return p.x*q.x+p.y*q.y; }
double dist2(PT p, PT q) { return dot(p-q,p-q); }
double cross(PT p, PT q) { return p.x*q.y-p.y*q.x; }

// rotate a point CCW or CW around the origin
PT RotateCCW90(PT p) { return PT(-p.y,p.x); }
PT RotateCW90(PT p)  { return PT(p.y,-p.x); }
PT RotateCCW(PT p, double t) {
  return PT(p.x*cos(t)-p.y*sin(t), p.x*sin(t)+p.y*cos(t));
}

// project point c onto line through a and b, assuming a != b
PT ProjectPointLine(PT a, PT b, PT c) {
  return a + (b-a)*dot(c-a, b-a)/dot(b-a, b-a);
}

// project point c onto line segment through a and b
PT ProjectPointSegment(PT a, PT b, PT c) {
  double r = dot(b-a,b-a);
  if (fabs(r) < EPS) return a;
  r = dot(c-a, b-a)/r;
  if (r < 0) return a;
  if (r > 1) return b;
  return a + (b-a)*r;
}

// compute distance from c to segment between a and b
double DistancePointSegment(PT a, PT b, PT c) {
  return sqrt(dist2(c, ProjectPointSegment(a, b, c)));
}

// compute distance between point (x,y,z) and plane ax+by+cz=d
double DistancePointPlane(double x, double y, double z,
                          double a, double b, double c, double d) {
  return fabs(a*x+b*y+c*z-d)/sqrt(a*a+b*b+c*c);
}

// determine if lines from a to b and c to d are parallel/collinear
bool LinesParallel(PT a, PT b, PT c, PT d) {
  return fabs(cross(b-a, c-d)) < EPS;
}
bool LinesCollinear(PT a, PT b, PT c, PT d) {
  return LinesParallel(a, b, c, d)
    && fabs(cross(a-b, a-c)) < EPS
    && fabs(cross(c-d, c-a)) < EPS;
}

// determine if line segment from a to b intersects with
// line segment from c to d
bool SegmentsIntersect(PT a, PT b, PT c, PT d) {
  if (LinesCollinear(a, b, c, d)) {
    if (dist2(a, c) < EPS || dist2(a, d) < EPS ||
      dist2(b, c) < EPS || dist2(b, d) < EPS) return true;
    if (dot(c-a, c-b) > 0 && dot(d-a, d-b) > 0 && dot(c-b, d-b) > 0)
      return false;
    return true;
  }
  if (cross(d-a, b-a) * cross(c-a, b-a) > 0) return false;
  if (cross(a-c, d-c) * cross(b-c, d-c) > 0) return false;
  return true;
}

// compute intersection of line passing through a and b
// with line passing through c and d, assuming that unique
// intersection exists; for segment intersection, check if
// segments intersect first
PT ComputeLineIntersection(PT a, PT b, PT c, PT d) {
  b=b-a; d=c-d; c=c-a;
  assert(dot(b, b) > EPS && dot(d, d) > EPS);
  return a + b*cross(c, d)/cross(b, d);
```

```
}

// compute center of circle given three points
PT ComputeCircleCenter(PT a, PT b, PT c) {
  b=(a+b)/2, c=(a+c)/2;
  return ComputeLineIntersection(b, b+RotateCW90(a-b), c, c+RotateCW90(a-c));
}

// compute center of circle given two points and radius
PT ComputeCircleCenter(PT a, PT b, double r) {
  double det = r * r / dist2(a, b) - 0.25;
  if (det < 0) return PT(INF,INF); // does not exist
  return (a + b) * 0.5 + PT(a.y - b.y, b.x - a.x) * sqrt(det);
}

// determine if point is in a possibly non-convex polygon (by William
// Randolph Franklin); returns 1 for strictly interior points, 0 for
// strictly exterior points, and 0 or 1 for the remaining points.
// Note that it is possible to convert this into an *exact* test using
// integer arithmetic by taking care of the division appropriately
// (making sure to deal with signs properly) and then by writing exact
// tests for checking point on polygon boundary
bool PointInPolygon(const vector<PT> &p, PT q) {
  bool c = 0;
  for (int i = 0; i < p.size(); i++){
    int j = (i+1)%p.size();
    if ((p[i].y <= q.y && q.y < p[j].y ||
      p[j].y <= q.y && q.y < p[i].y) && q.x < p[i].x +
      (p[j].x - p[i].x) * (q.y - p[i].y) / (p[j].y - p[i].y))
      c = !c;
  }
  return c;
}

// determine if point is on the boundary of a polygon
bool PointOnPolygon(const vector<PT> &p, PT q) {
  for (int i = 0; i < p.size(); i++)
    if (dist2(ProjectPointSegment(p[i], p[(i+1)%p.size()], q), q) < EPS)
      return true;
  return false;
}

// compute intersection of line through points a and b with
// circle centered at c with radius r > 0
vector<PT> CircleLineIntersection(PT a, PT b, PT c, double r) {
  vector<PT> ret;
  b = b-a;
  a = a-c;
  double A = dot(b, b);
  double B = dot(a, b);
  double C = dot(a, a) - r*r;
  double D = B*B - A*C;
  if (D < -EPS) return ret;
  ret.push_back(c+a+b*(-B+sqrt(D+EPS))/A);
  if (D > EPS) ret.push_back(c+a+b*(-B-sqrt(D))/A);
  return ret;
}

// compute intersection of circle centered at a with radius r
// with circle centered at b with radius R
vector<PT> CircleCircleIntersection(PT a, PT b, double r, double R) {
  vector<PT> ret;
  double d = sqrt(dist2(a, b));
  if (d > r+R || d+min(r, R) < max(r, R)) return ret;
  double x = (d*d-R*R+r*r)/(2*d);
  double y = sqrt(r*r-x*x);
  PT v = (b-a)/d;
  ret.push_back(a+v*x + RotateCCW90(v)*y);
  if (y > 0) ret.push_back(a+v*x - RotateCCW90(v)*y);
  return ret;
}

// This code computes the area or centroid of a (possibly nonconvex)
// polygon, assuming that the coordinates are listed in a clockwise or
// counterclockwise fashion.  Note that the centroid is often known as
// the "center of gravity" or "center of mass".
double ComputeSignedArea(const vector<PT> &p) {
  double area = 0;
  for (int i = 0; i < p.size(); i++) {
    int j = (i+1) % p.size();
    area += p[i].x*p[j].y - p[j].x*p[i].y;
  }
  return area / 2.0;
}
double ComputeArea(const vector<PT> &p) {
  return fabs(ComputeSignedArea(p));
}
PT ComputeCentroid(const vector<PT> &p) {
  PT c(0,0);
  double scale = 6.0 * ComputeSignedArea(p);
  for (int i = 0; i < p.size(); i++){
    int j = (i+1) % p.size();
```

```
    c = c + (p[i]+p[j])*(p[i].x*p[j].y - p[j].x*p[i].y);
  }
  return c / scale;
}

// tests whether or not a given polygon (in CW or CCW order) is simple
// a polygon is considered simple if its sides do not intersect.
bool IsSimple(const vector<PT> &p) {
  for (int i = 0; i < p.size(); i++) {
    for (int k = i+1; k < p.size(); k++) {
      int j = (i+1) % p.size();
      int l = (k+1) % p.size();
      if (i == l || j == k) continue;
      if (SegmentsIntersect(p[i], p[j], p[k], p[l]))
        return false;
    }
  }
  return true;
}

int main() {
  // expected (-5,2)/(5,-2)/(-5,2)/(5,2)/(5,2) (7.5,3) (2.5,1)
  cerr << RotateCCW90(PT(2,5)) << endl;
  cerr << RotateCW90(PT(2,5)) << endl;
  cerr << RotateCCW(PT(2,5),M_PI/2) << endl;
  cerr << ProjectPointLine(PT(-5,-2), PT(10,4), PT(3,7)) << endl;
  cerr << ProjectPointSegment(PT(-5,-2), PT(10,4), PT(3,7)) << " "
     << ProjectPointSegment(PT(7.5,3), PT(10,4), PT(3,7)) << " "
     << ProjectPointSegment(PT(-5,-2), PT(2.5,1), PT(3,7)) << endl;

  // expected 6.78903/1 0 1/0 0 1/1 1 1 0/(1,2)/(1,1)
  cerr << DistancePointPlane(4,-4,3,2,-2,5,-8) << endl;
  cerr << LinesParallel(PT(1,1), PT(3,5), PT(2,1), PT(4,5)) << " "
     << LinesParallel(PT(1,1), PT(3,5), PT(2,0), PT(4,5)) << " "
     << LinesParallel(PT(1,1), PT(3,5), PT(5,9), PT(7,13)) << endl;
  cerr << LinesCollinear(PT(1,1), PT(3,5), PT(2,1), PT(4,5)) << " "
     << LinesCollinear(PT(1,1), PT(3,5), PT(2,0), PT(4,5)) << " "
     << LinesCollinear(PT(1,1), PT(3,5), PT(5,9), PT(7,13)) << endl;
  cerr << SegmentsIntersect(PT(0,0), PT(2,4), PT(3,1), PT(-1,3)) << " "
     << SegmentsIntersect(PT(0,0), PT(2,4), PT(4,3), PT(0,5)) << " "
     << SegmentsIntersect(PT(0,0), PT(2,4), PT(2,-1), PT(-2,1)) << " "
     << SegmentsIntersect(PT(0,0), PT(2,4), PT(5,5), PT(1,7)) << endl;
  cerr << ComputeLineIntersection(PT(0,0),PT(2,4),PT(3,1),PT(-1,3)) << endl;
  cerr << ComputeCircleCenter(PT(-3,4), PT(6,1), PT(4,5)) << endl;

  vector<PT> v; v.push_back(PT(0,0)); v.push_back(PT(5,0));
  v.push_back(PT(5,5)); v.push_back(PT(0,5));

  // expected: 1 1 1 0 0
  cerr << PointInPolygon(v, PT(2,2)) << " "
     << PointInPolygon(v, PT(2,0)) << " "
     << PointInPolygon(v, PT(0,2)) << " "
     << PointInPolygon(v, PT(5,2)) << " "
     << PointInPolygon(v, PT(2,5)) << endl;

  // expected: 0 1 1 1 1
  cerr << PointOnPolygon(v, PT(2,2)) << " "
     << PointOnPolygon(v, PT(2,0)) << " "
     << PointOnPolygon(v, PT(0,2)) << " "
     << PointOnPolygon(v, PT(5,2)) << " "
     << PointOnPolygon(v, PT(2,5)) << endl;

  // expected: (1,6)/(5,4) (4,5)//(4,5) (5,4)//(4,5) (5,4)
  vector<PT> u = CircleLineIntersection(PT(0,6), PT(2,6), PT(1,1), 5);
  for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
  u = CircleLineIntersection(PT(0,9), PT(9,0), PT(1,1), 5);
  for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
  u = CircleCircleIntersection(PT(1,1), PT(10,10), 5, 5);
  for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
  u = CircleCircleIntersection(PT(1,1), PT(8,8), 5, 5);
  for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
  u = CircleCircleIntersection(PT(1,1), PT(4.5,4.5), 10, sqrt(2.0)/2.0);
  for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
  u = CircleCircleIntersection(PT(1,1), PT(4.5,4.5), 5, sqrt(2.0)/2.0);
  for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;

  // area should be 5.0; centroid should be (1.1666666, 1.166666)
  PT pa[] = {PT(0,0), PT(5,0), PT(1,1), PT(0,5)};
  vector<PT> p(pa, pa+4); PT c = ComputeCentroid(p);
  cerr << "Area: " << ComputeArea(p) << endl;
  cerr << "Centroid: " << c << endl;
}
```

4 Strings

Knuth–Morris–Pratt Algorithm (KMP)

```
// Searchs for a pattern P in a string T.
// Running time: O(n)
```

```
string P, T; // Pattern, Text
int F[MAXN]; // Failure Function
void kmpPreprocess() { // Builds F[]
  int i = 0, j = -1; F[0] = -1; // starting values
  while (i < (int)P.size()) { // pre-process the pattern string P
    while (j>=0 && P[i] != P[j]) j = F[j]; // if different, reset j
    i++; j++; // if same, advance both pointers
    F[i] = j; // observe i = 8, 9, 10, 11, 12 with j = 0, 1, 2, 3, 4
  }
} // in the example of P = "SEVENTY SEVEN" above

int kmpSearch() { // this is similar as kmpPreprocess(), but on string T
  int ret = 0, i = 0, j = 0; // starting values
  while (i < (int)T.size()) { // search through string T
    while (j>=0 && T[i] != P[j]) j = F[j]; // if different, reset j
    i++; j++; // if same, advance both pointers
    if (j == (int)P.size()) { // a match found when j == m
      ret++; // printf("P is found at index %d in T\n", i - j);
      j = F[j]; // prepare j for the next possible match
    }
  }
  return ret;
}
```

Lex–rot

```
// Finds # of rotations in str to find the lexicographically smaller string
// Running time: O(n)

int lexRot(string str) {
  int n = str.size(), ini=0, fim=1, rot=0;
  str += str;
  while(fim < n && rot+ini+1 < n)
    if (str[ini+rot] == str[ini+fim]) ini++;
    else if (str[ini+rot] < str[ini+fim]) fim += ini+1, ini = 0;
    else rot = max(rot+ini+1, fim), fim = rot+1, ini = 0;
  return rot;
}
```

Longest palindrome (Manacher)

```
// Finds the longest palindrome in a string s. Notice that array P[i] will
// store the radius of the longest palindrome centered at T[i].
// Running time: O(n)

// Transform S into T. Example: S = "abba", T = "^#a#b#b#a#$"
// ^ and $ signs are sentinels to avoid bounds checking
string preProcess(string& s) {
  int n = s.length();
  if (n == 0) return "^$";
  string ret = "^";
  for (int i = 0; i < n; i++)
    ret += "#" + s.substr(i, 1);
  ret += "#$";
  return ret;
}
string longestPalindrome(string& s) {
  string T = preProcess(s);
  int n = T.length(), C = 0, R = 0;
  int *P = new int[n]; // may be useful OUTSIDE this function
  for (int i = 1; i < n-1; i++) {
    int i_mirror = (C<<1)-i; // i' = C-(i-C)
    P[i] = (R > i) ? min(R-i, P[i_mirror]) : 0;
    // Attempt to expand palindrome centered at i
    while (T[i + 1 + P[i]] == T[i - 1 - P[i]])
      P[i]++;
    // If palindrome centered at i expand past R,
    // adjust center based on expanded palindrome.
    if (i + P[i] > R) {
      C = i;
      R = i + P[i];
    }
  }
  // Find the maximum element in P.
  int maxLen = 0, centerIndex = 0;
  for (int i = 1; i < n-1; i++)
    if (P[i] > maxLen) {
      maxLen = P[i];
      centerIndex = i;
    }
  delete[] P;
  return s.substr((centerIndex - 1 - maxLen)/2, maxLen);
}
```

Suffix Array and Longest Common Prefix

```
// Calculates the suffix array (and LCP) of a string.
// Running time: O(n log n)

int sa[MAXN], invsa[MAXN], n, sz;
inline bool cmp(int a, int b) { return invsa[a+sz]<invsa[b+sz]; }
void sort_sa(int a, int b) {
  if (a == b) return;
  int pivot = sa[a + rand()%(b-a)], c = a, d = b;
  for (int i = c; i < b; i++)
    if (cmp(sa[i], pivot)) swap(sa[i], sa[c++]);
  for (int i = d-1; i >= a; i--)
    if (cmp(pivot, sa[i])) swap(sa[i], sa[--d]);
  sort_sa(a, c);
  for (int i = c; i < d; i++) invsa[sa[i]] = d-1;
  if (d-c == 1) sa[c] = -1;
  sort_sa(d, b);
}
void suffix_array(char* s) { // could be int*s; but del strlen(n)
  n = strlen(s), invsa[n] = -1;
  for (int i = 0; i < n; i++) sa[i] = i, invsa[i] = s[i];
  sz = 0; sort_sa(0, n);
  for (sz = 1; sz < n; sz *= 2)
    for (int i = 0, j = i; i < n; i = j)
      if (sa[i] < 0) {
        while (sa[j] < 0) j += (-sa[j]);
        sa[i] = -(j-i);
      } else sort_sa(i, j=invsa[sa[i]]+1);
  for (int i = 0; i < n; i++) sa[invsa[i]] = i;
}
int lcp[MAXN];
void calc_lcp(char* s) { // could be int*s
  for (int i = 0, l = 0; i < n; i++, l = max(0, l-1)) {
    if (invsa[i] == 0) continue;
    int j = sa[invsa[i]-1];
    while (max(i+l, j+l) < n && s[i+l] == s[j+l]) l++;
    lcp[invsa[i]] = l;
  }//for(int i=0;i+1<n;i++)lcp[i]=lcp[i+1];lcp[n-1]=0;
}
```

Z-Algorithm

```
// Builds array z[], such that z[i] is the length of the longest substring
// starting at s[i] that is also a prefix of s.
// Running time: O(n)

// note: MAXN > maxLen(T)+maxLen(S)
int z[MAXN]; // s[:z[i]] == s[i:i+z[i]]
void z_algorithm(string& s) {
  int n = s.length(), L = 0, R = 0;
  for (int i = 1; i < n; i++) {
    if (i > R) {
      L = R = i;
      while (R < n && s[R-L] == s[R]) R++;
      z[i] = (R--)-L;
    } else {
      int k = i-L;
      if (z[k] < R-i+1)
        z[i] = z[k];
      else {
        L = i;
        while (R < n && s[R-L] == s[R]) R++;
        z[i] = (R--)-L;
      }
    }
  }
}
// finds the indexes of all occurences of T in S
void indexesOf(string T, string& S, vector<int>& v) {
  int m = T.length();
  T += "$" + S;
  z_algorithm(T);
  for (int i = m+1; i < T.length(); i++)
    if (z[i] == m)
      v.push_back(i-m-1);
}
```

5 Graphs

Articulation points

```
// Unsorted output; doesn't include duplicated points.
// Running time: O(V+E)

vector<int> G[MAXN];
```

```
int idx[MAXN], times; bool vis[MAXN], isAns[MAXN]; vector<int> ans;
#define ROOT 0 // any vertice in the graph to call getArticulationPoints with
int getArticulationPoints(int v=ROOT) {
  vis[v] = true, idx[v] = times++;
  int children = 0, lowest = idx[v];
  for (int i = 0; i < (int)G[v].size(); i++) {
    if (!vis[G[v][i]]) {
      int m = getArticulationPoints(G[v][i]);
      lowest = min(lowest, m), children++;
      if (idx[v] <= m && (v != ROOT || children >= 2) && !isAns[v])
        isAns[v] = true, ans.push_back(v);
    } else lowest = min(lowest, idx[G[v][i]]);
  }
  return lowest;
}
```

Bellman–Ford

```
// Finds the minimum distance from vertex start to all the other vertices.
// In case a -oo cycle exists, returns true.
// Running time: O(VE)

#define MAXN 1000
#define $w first
#define $u second.first
#define $v second.second
vector< pair< int, pair<int, int> > > edges; // (w, (u,v))
int dist[MAXN], n;
int bellman_ford(int start) {
  for (int i = 0; i < n; i++)
    dist[i] = INF;
  dist[start] = 0;
  for (int i = 0; i < n; i++) {
    bool edit = false;
    for (int j = 0; j < m; j++)
      if (dist[edges[j].$u] + edges[j].$w < dist[edges[j].$v]) {
        dist[edges[j].$v] = dist[edges[j].$u] + edges[j].$w;
        edit = true;
      }
    if (!edit) break;
    if (i+1 == n) return true;
  }
  return false;
}
```

Eulerian Path

```
// Finds a path in the graph that visits each edge exactly once.

struct Edge;
typedef list<Edge>::iterator iter;
struct Edge {
  int next_vertex; iter reverse_edge;
  Edge(int next_vertex):next_vertex(next_vertex) { }
};

const int max_vertices = 10000;
int num_vertices;
list<Edge> adj[max_vertices]; // adjacency list
vector<int> path;

void find_path(int v) {
  while(adj[v].size() > 0) {
    int vn = adj[v].front().next_vertex;
    adj[vn].erase(adj[v].front().reverse_edge);
    adj[v].pop_front();
    find_path(vn);
  }
  path.push_back(v);
}
void add_edge(int a, int b) {
  adj[a].push_front(Edge(b));
  iter ita = adj[a].begin();
  adj[b].push_front(Edge(a));
  iter itb = adj[b].begin();
  ita->reverse_edge = itb;
  itb->reverse_edge = ita;
}
```

Max bipartite matching

```
// This code performs maximum bipartite matching.
//
```

```
// Running time: O(|E| |V|) -- often much faster in practice
//
//   INPUT: w[i][j] = edge between row node i and column node j
//   OUTPUT: mr[i] = assignment for row node i, -1 if unassigned
//           mc[j] = assignment for column node j, -1 if unassigned
//           function returns number of matches made

typedef vector<int> VI;
typedef vector<VI> VVI;
bool FindMatch(int i, const VVI &w, VI &mr, VI &mc, VI &seen) {
  for (int j = 0; j < w[i].size(); j++) {
    if (w[i][j] && !seen[j]) {
      seen[j] = true;
      if (mc[j] < 0 || FindMatch(mc[j], w, mr, mc, seen)) {
        mr[i] = j, mc[j] = i;
        return true;
      }
    }
  }
  return false;
}
int BipartiteMatching(const VVI &w, VI &mr, VI &mc) {
  mr = VI(w.size(), -1);
  mc = VI(w[0].size(), -1);
  int ct = 0;
  for (int i = 0; i < w.size(); i++) {
    VI seen(w[0].size());
    if (FindMatch(i, w, mr, mc, seen)) ct++;
  }
  return ct;
}
```

Max–flow (Dinic)

```
// Calculates the max flow of a graph.
// Running time: O(E V^2)

const int MAXN = 5005, MAXE = 30005;
typedef long long lint;
struct Graph {
  int n, m; // << set n (number of vertices), vertices are 0-indexed
  vector<int> adj[MAXN];
  pair<int, int> edges[2*MAXE];
  inline void add_edge(int v, int u, int vu, int uv=0) {
    edges[m] = make_pair(u, vu); adj[v].push_back(m++);
    edges[m] = make_pair(v, uv); adj[u].push_back(m++);
  }
  int dis[MAXN], pos[MAXN];
  int fluxo[2*MAXE];
  int src, dst; // << set these
} G;
bool dinic_bfs(Graph& g) {
  queue<int> qu;
  qu.push(g.src);

  for (int i = 0; i < g.n; i++) g.dis[i] = -1;
  g.dis[g.src] = 0;

  while (!qu.empty()) {
    int v = qu.front(); qu.pop();
    for (int i = 0; i < g.adj[v].size(); i++) {
      int e = g.adj[v][i];
      int u = g.edges[e].first;
      int c = g.edges[e].second;
      if (c > 0 && g.dis[u] == -1) {
        g.dis[u] = g.dis[v] + 1;
        qu.push(u);
      }
    }
  }
  return g.dis[g.dst] != -1;
}
int dinic_dfs(int v, int flow, Graph& g) {
  if (v == g.dst) return flow;

  for (int& i = g.pos[v]; i < g.adj[v].size(); i++) {
    int e = g.adj[v][i];
    int u = g.edges[e].first;
    int c = g.edges[e].second;
    if (c > 0 && g.dis[u] == g.dis[v] + 1) {
      int flow_ = dinic_dfs(u, min(flow, c), g);
      if (flow_ > 0) {
        g.edges[e].second -= flow_;
        g.edges[e^1].second += flow_;
        return flow_;
      }
    }
  }
  return 0;
```

```
}
lint dinic(Graph& g) {
  lint max_flow = 0;
  while (dinic_bfs(g)) {
    for (int i = 0; i < g.n; i++) g.pos[i] = 0;
    while (int flow = dinic_dfs(g.src, INT_MAX, g))
      max_flow += flow;
  }
  return max_flow;
}
int main() {
  G.n = 6; // number of vertices: 6 = 0..5
  G.src = 1; G.dst = 3; // Vertices: source (1) and sink (3)
  G.add_edge(1, 3, 5, 9); // adds edge 1->3 with cap. 5 and 3->1 with cap. 9
  printf("Max␣flow:␣%d\n", dinic(G));
  for (int i = 0; i < G.n; i++) printf("flow[%d]␣=␣%d\n", i, G.fluxo[i]);
}
```

Min–cost matching

```
///////////////////////////////////////////////////////////////////////////
// Min cost bipartite matching via shortest augmenting paths
//
// This is an O(n^3) implementation of a shortest augmenting path
// algorithm for finding min cost perfect matchings in dense
// graphs.  In practice, it solves 1000x1000 problems in around 1
// second.
//
//   cost[i][j] = cost for pairing left node i with right node j
//   Lmate[i] = index of right node that left node i pairs with
//   Rmate[j] = index of left node that right node j pairs with
//
// The values in cost[i][j] may be positive or negative.  To perform
// maximization, simply negate the cost[][] matrix.
///////////////////////////////////////////////////////////////////////////

typedef vector<double> VD;
typedef vector<VD> VVD;
typedef vector<int> VI;

double MinCostMatching(const VVD &cost, VI &Lmate, VI &Rmate) {
  int n = int(cost.size());

  // construct dual feasible solution
  VD u(n), v(n);
  for (int i = 0; i < n; i++) {
    u[i] = cost[i][0];
    for (int j = 1; j < n; j++) u[i] = min(u[i], cost[i][j]);
  }
  for (int j = 0; j < n; j++) {
    v[j] = cost[0][j] - u[0];
    for (int i = 1; i < n; i++) v[j] = min(v[j], cost[i][j] - u[i]);
  }

  // construct primal solution satisfying complementary slackness
  Lmate = VI(n, -1);
  Rmate = VI(n, -1);
  int mated = 0;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      if (Rmate[j] != -1) continue;
      if (fabs(cost[i][j] - u[i] - v[j]) < 1e-10) {
        Lmate[i] = j;
        Rmate[j] = i;
        mated++;
        break;
      }
    }
  }

  VD dist(n);
  VI dad(n), seen(n);

  // repeat until primal solution is feasible
  while (mated < n) {
    // find an unmatched left node
    int s = 0;
    while (Lmate[s] != -1) s++;

    // initialize Dijkstra
    fill(dad.begin(), dad.end(), -1);
    fill(seen.begin(), seen.end(), 0);
    for (int k = 0; k < n; k++)
      dist[k] = cost[s][k] - u[s] - v[k];

    int j = 0;
    while (true) {
      // find closest
      j = -1;
```

```
      for (int k = 0; k < n; k++) {
        if (seen[k]) continue;
        if (j == -1 || dist[k] < dist[j]) j = k;
      }
      seen[j] = 1;

      // termination condition
      if (Rmate[j] == -1) break;

      // relax neighbors
      const int i = Rmate[j];
      for (int k = 0; k < n; k++) {
        if (seen[k]) continue;
        const double new_dist = dist[j] + cost[i][k] - u[i] - v[k];
        if (dist[k] > new_dist) {
          dist[k] = new_dist;
          dad[k] = j;
        }
      }
    }

    // update dual variables
    for (int k = 0; k < n; k++) {
      if (k == j || !seen[k]) continue;
      const int i = Rmate[k];
      v[k] += dist[k] - dist[j];
      u[i] -= dist[k] - dist[j];
    }
    u[s] += dist[j];

    // augment along path
    while (dad[j] >= 0) {
      const int d = dad[j];
      Rmate[j] = Rmate[d];
      Lmate[Rmate[j]] = j;
      j = d;
    }
    Rmate[j] = s;
    Lmate[s] = j;
    mated++;
  }
  double value = 0;
  for (int i = 0; i < n; i++)
    value += cost[i][Lmate[i]];
  return value;
}
```

Min-cost max-flow

```
// Implementation of min cost max flow algorithm using adjacency
// matrix (Edmonds and Karp 1972).  This implementation keeps track of
// forward and reverse edges separately (so you can set cap[i][j] !=
// cap[j][i]).  For a regular max flow, set all edge costs to 0.
//
// Running time, O(|V|^2) cost per augmentation
//     max flow:           O(|V|^3) augmentations
//     min cost max flow:  O(|V|^4 * MAX_EDGE_COST) augmentations
//
// INPUT:
//     - graph, constructed using AddEdge()
//     - source
//     - sink
//
// OUTPUT:
//     - (maximum flow value, minimum cost value)
//     - To obtain the actual flow, look at positive values only.

typedef vector<int> VI;
typedef vector<VI> VVI;
typedef long long L;
typedef vector<L> VL;
typedef vector<VL> VVL;
typedef pair<int, int> PII;
typedef vector<PII> VPII;

const L INF = numeric_limits<L>::max() / 4;

struct MinCostMaxFlow {
  int N;
  VVL cap, flow, cost;
  VI found;
  VL dist, pi, width;
  VPII dad;

  MinCostMaxFlow(int N) :
    N(N), cap(N, VL(N)), flow(N, VL(N)), cost(N, VL(N)),
    found(N), dist(N), pi(N), width(N), dad(N) {}

  void AddEdge(int from, int to, L cap, L cost) {
    this->cap[from][to] = cap;
```

```
  this->cost[from][to] = cost;
}

void Relax(int s, int k, L cap, L cost, int dir) {
  L val = dist[s] + pi[s] - pi[k] + cost;
  if (cap && val < dist[k]) {
    dist[k] = val;
    dad[k] = make_pair(s, dir);
    width[k] = min(cap, width[s]);
  }
}

L Dijkstra(int s, int t) {
  fill(found.begin(), found.end(), false);
  fill(dist.begin(), dist.end(), INF);
  fill(width.begin(), width.end(), 0);
  dist[s] = 0;
  width[s] = INF;

  while (s != -1) {
    int best = -1;
    found[s] = true;
    for (int k = 0; k < N; k++) {
      if (found[k]) continue;
      Relax(s, k, cap[s][k] - flow[s][k], cost[s][k], 1);
      Relax(s, k, flow[k][s], -cost[k][s], -1);
      if (best == -1 || dist[k] < dist[best]) best = k;
    }
    s = best;
  }

  for (int k = 0; k < N; k++)
    pi[k] = min(pi[k] + dist[k], INF);
  return width[t];
}

pair<L, L> GetMaxFlow(int s, int t) {
  L totflow = 0, totcost = 0;
  while (L amt = Dijkstra(s, t)) {
    totflow += amt;
    for (int x = t; x != s; x = dad[x].first) {
      if (dad[x].second == 1) {
        flow[dad[x].first][x] += amt;
        totcost += amt * cost[dad[x].first][x];
      } else {
        flow[x][dad[x].first] -= amt;
        totcost -= amt * cost[x][dad[x].first];
      }
    }
  }
  return make_pair(totflow, totcost);
}
};
```

Min–cut

```
// Adjacency matrix implementation of Stoer-Wagner min cut algorithm.
// Running time: O(|V|^3)
//
// INPUT: - graph, constructed using AddEdge()
// OUTPUT: - (min cut value, nodes in half of min cut)

typedef vector<int> VI;
typedef vector<VI> VVI;
const int INF = 1000000000;

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

  for (int phase = N-1; phase >= 0; phase--) {
    VI w = weights[0], added = used;
    int prev, last = 0;
    for (int i = 0; i < phase; i++) {
      prev = last, last = -1;
      for (int j = 1; j < N; j++)
        if (!added[j] && (last == -1 || w[j] > w[last])) last = j;
      if (i == phase-1) {
        for (int j = 0; j < N; j++) weights[prev][j] += weights[last][j];
        for (int j = 0; j < N; j++) weights[j][prev] = weights[prev][j];
        used[last] = true;
        cut.push_back(last);
        if (best_weight == -1 || w[last] < best_weight)
          best_cut = cut, best_weight = w[last];
      } else {
        for (int j = 0; j < N; j++) w[j] += weights[last][j];
        added[last] = true;
      }
    }
```

```
  }
  return make_pair(best_weight, best_cut);
}
```

Strongly connected components (Tarjan)

```
// Tarjan algorithm: finds the strongly connected components on the graph.
// Stores the scc number for vertex v in scc[v].
// Running time: O(n)

vector<int> G[MAXN];
int idx[MAXN], idx_count, scc[MAXN], scc_count, sk[MAXN], sk_size;
bool stacked[MAXN], vis[MAXN];
void tarjan(int v) {
  int idxv;
  idx[v] = idxv = ++idx_count;
  sk[sk_size++] = v, stacked[v] = true;
  for (int i = 0; i < G[v].size(); i++) {
    int u = G[v][i];
    if (!vis[u]) {
      vis[u] = true;
      tarjan(u);
    }
    if (stacked[u])
      idx[v] = min(idx[v], idx[u]);
  }
  if (idx[v] == idxv) {
    int u;
    scc_count++;
    do {
      u = sk[--sk_size];
      stacked[u] = false;
      scc[u] = scc_count;
    }
    while (u != v);
  }
}
void find_scc(int N, int st=0) {
  for (int i = st; i < N; i++)
    stacked[i] = vis[i] = scc[i] = 0;
  idx_count = scc_count = sk_size = 0;
  for (int i = st; i < N; i++)
    if (!vis[i])
      tarjan(i);
}
```

2-Sat

```
#define NOT(v) ((v)^1)
//_2sat_edge(v_not ? NOT(v) : v, u_not ? NOT(u) : u);
inline bool _2sat_edge(int v, int u) {
  G[NOT(v)].push_back(u);
  G[NOT(u)].push_back(v);
}
bool _2sat(int N, int st=0) {
  find_scc(N, st);
  for (int i = st; i < N; i += 2)
    if (scc[i] == scc[NOT(i)])
      return false;
  return true;
}
```

Tree distance sum

```
// Calculates the sum of dist(v,u) for all pairs of vertices v, u.
// Running time: O(n)

int distsum, n;
int dfs(int v, int p=-1, int w=0) {
  int k = 1;
  for (int i = 0; i < G[v].size(); i++) {
    int u = G[v][i].first, w = G[v][i].second;
    if (u != p) k += dfs(u, v, w);
  }
  distsum += w*(n-k)*k;
  return k;
}
```

6 Data Structures

Bigint

```
#include <sstream>
const int DIG = 4;
const int BASE = 10000; // BASE**3 < 2**51
const int TAM = 2048;
const double EPS = 1e-10;
inline int cmp (double x, double y = 0, double tol = EPS) {
  return (x <= y + tol) ? (x + tol < y) ? -1 : 0 : 1;
}
struct  bigint {
  int v[TAM], n;
  bigint(int x = 0): n(1) { memset(v, 0, sizeof(v)); v[n++] = x; fix(); }
  bigint(char  *s):  n(1) {
    memset(v, 0, sizeof(v));
    int sign = 1;
    while (*s && !isdigit(*s))
      if (*s++ == '-')
        sign *= -1;
    char *t = strdup(s), *p = t + strlen(t);
    while (p > t) {
      *p = 0;
      p = max(t, p - DIG);
      sscanf (p, "%d", &v[n]);
      v[n++] *= sign;
    }
    free(t), fix();
  }
  bigint& fix(int m=0) {
    n = max(m, n);
    int sign = 0;
    for (int i=1, e=0; i <= n || e && (n=i); i++) {
      v[i]  +=  e;
      e = v[i] / BASE;
      v[i] %= BASE;
      if (v[i])
        sign = (v[i] > 0) ? 1 : -1;
    }
    for (int i = n-1; i > 0; i--)
      if (v[i] * sign < 0)
        v[i] += sign * BASE, v[i+1] -= sign;
    while (n && !v[n]) n--;
    return *this;
  }
int cmp(const bigint& x=0) const {
  int i = max(n, x.n), t=0;
  while (true)
    if ((t = ::cmp(v[i], x.v[i])) || !i--)
      return t;
}
bool operator <(const bigint& x) const { return cmp(x) < 0; }
bool operator ==(const bigint& x) const { return cmp(x) == 0; }
bool operator !=(const bigint& x) const { return cmp(x) != 0; }

operator string() const {
  ostringstream s;
  s << v[n];
  for (int i = n-1; i>0; i--) {
    s.width(DIG);
    s.fill('0');
    s << abs(v[i]);
  }
  return s.str();
}
friend ostream& operator <<(ostream& o, const bigint& x) {
  return o << (string) x;
}

bigint& operator +=(const bigint& x) {
  for (int i = 1; i <= x.n; i++)
    v[i] += x.v[i];
  return fix(x.n);
}
bigint operator +(const bigint& x) { return bigint(*this) += x; }
bigint& operator -=(const bigint& x) {
  for (int i = 1; i <= x.n; i++)
    v[i] -= x.v[i];
  return fix(x.n);
}
bigint operator -(const bigint& x) { return bigint(*this) -= x; }
bigint operator -() { bigint r = 0; return r -= *this; }

void ams(const bigint& x, int m, int b) { //  *this  +=  (x  *  m)  <<  b;
  for (int i=1, e=0; (i <= x.n || e) && (n = i + b); i++) {
    v[i+b] += x.v[i] * m + e;
    e = v[i+b] / BASE;
    v[i+b]  %=  BASE;
  }
}
```

```
bigint operator *(const bigint& x) const {
  bigint r;
  for (int i = 1; i <= n; i++)
    r.ams(x, v[i], i-1);
  return r;
}
bigint& operator *=(const bigint& x) { return *this = *this * x; }
// cmp(x / y) == cmp(x) * cmp(y); cmp(x % y) == cmp(x);
bigint div(const bigint& x) {
  if (x == 0) return 0;
  bigint q;
  q.n = max(n - x.n + 1, 0);
  int d = x.v[x.n] * BASE + x.v[x.n-1];
  for (int i = q.n; i > 0; i--) {
    int j = x.n + i - 1;
    q.v[i] = int((v[j] * double(BASE) + v[j-1]) / d);
    ams(x, -q.v[i], i-1);
    if (i == 1 || j == 1)
      break;
    v[j-1] += BASE * v[j];
    v[j] = 0;
  }
  fix(x.n);
  return q.fix();
}
bigint& operator /=(const bigint& x) { return *this = div(x); }
bigint& operator %=(const bigint& x) { div(x); return *this; }
bigint operator /(const bigint& x) { return bigint(*this).div(x); }
bigint operator %(const bigint& x) { return bigint(*this) %= x; }

bigint pow(int x) {
  if (x < 0)
    return (*this == 1 || *this == -1) ? pow(-x) : 0;
  bigint r = 1;
  for (int i = 0; i < x; i++)
    r *= *this;
  return r;
}
bigint root(int x) {
  if (cmp() == 0 || cmp() < 0 && x % 2 == 0)
    return 0;
  if (*this == 1 || x == 1)
    return *this;
  if (cmp() < 0)
    return -(-*this).root(x);
  bigint a = 1, d = *this;
    while (d != 1) {
      bigint b = a + (d /= 2);
      if (cmp(b.pow(x)) >= 0) {
        d += 1;
        a = b;
      }
    }
    return a;
  }
};
```

Hashstring

```
Gerando bases B:
  B[i] = BASE^i % m, 0 <= i
  B^[-i] = BASE^(m-1-i) % m, 1 <= i
Gerando hash H para uma string S de tamanho n+1:
  H = (S[0] + S[1]*B[1] + ... + S[n]*B[n]) % m
  H[n] = (H[n-1] + S[n]*B[n]) % m
Calculando hash h no intervalo [a,b]:
  h = (H[b] - H[a-1] + m) * B[-a] % m
```

Lowest Common Ancestor (LCA)

```
// Calculates lca(a,b) in a tree in O(log n).
// Running time: O(log n) + pre-computing O(n log n)

const int MAXN=1000005, LOGMAXN=2+log2(MAXN);
vector<int> G[MAXN];
int parent[LOGMAXN][MAXN], depth[MAXN];

// Generates parent[][] and depth[]; call dfs(root)
void dfs(int v, int p=-1) {
  depth[v] = (p >= 0 ? depth[p] + 1 : 0);
  parent[0][v] = p;
  for (int i = 0, l = 31-__builtin_clz(depth[v]); i <= l; i++)
    parent[i+1][v] = parent[i][parent[i][v]];
  for (int i = 0; i < G[v].size(); i++)
    if (G[v][i] != p)
      dfs(G[v][i], v);
```

```
}

// Gets lca(a,b)
int lca(int a, int b) {
  // puts both on same depth
  if (depth[a] > depth[b]) swap(a, b);
  for (int d = depth[b] - depth[a]; d; d -= d&-d)
    b = parent[__builtin_ctz(d)][b];
  if (a == b) return a;
  // goes up as much as possible keeping a != b
  for (int up = 31-__builtin_clz(depth[a]); up >= 0; up--)
    if (parent[up][a] != parent[up][b])
      a = parent[up][a], b = parent[up][b];
  return parent[0][a]; // a != b, but parent(a) = parent(b) = lca
}
```

Segment Tree

```
#define st_left(idx) (2*(idx)+1)
#define st_right(idx) (2*(idx)+2)
#define st_middle(left,right) (((left)+(right))/2)
template<class T, int MAXSIZE>
class segtree {
  void from_array (T* v, int idx, int left, int right) {
    if (left != right) {
      from_array(v, st_left(idx), left, st_middle(left,right));
      from_array(v, st_right(idx), st_middle(left,right)+1, right);
      tree[idx] = tree[st_left(idx)] + tree[st_right(idx)];
    } else
      tree[idx] = v[left]; // to clear(), change v[left] to 0
  }
  T read (int i, int j, int idx, int left, int right) {
    if (i <= left && right <= j) return tree[idx];
    if (j < left || right < i) return 0;
    return read(i, j, st_left(idx), left, st_middle(left,right)) +
      read(i, j, st_right(idx), st_middle(left,right)+1, right);
  }
  void set (int x, T& v, int idx, int left, int right) {
    if (x < left || right < x) return;
    if (left != right) {
      set(x, v, st_left(idx), left, st_middle(left,right));
      set(x, v, st_right(idx), st_middle(left,right)+1, right);
      tree[idx] = tree[st_left(idx)] + tree[st_right(idx)];
    } else
      tree[idx] = v;
  }
public:
  T* tree; int size; segtree() { tree = new T[4*MAXSIZE]; }
  inline void from_array(T array[]) { from_array(array, 0, 0, size-1); }
  inline T read(int i, int j) { return read(i, j, 0, 0, size-1); }
  inline void set(int x, T v) { set(x, v, 0, 0, size-1); }
}; // int main () { segtree<int, MAXN> tree; tree.size = N; }
// note: it is required to clear the segtree before using!!
```

Segment Tree (with Lazy Propagation)

```
// Must receive type T of each element in the tree, type R of each element
// in the input and max size of the segtree on the template. Implement the
// update and the lines with //##//. DO NOT FORGET TO CLEAR BEFORE USING!!

#define nil 0 // value that doesn't interfere
#define st_left(idx) (2*(idx)+1)
#define st_right(idx) (2*(idx)+2)
#define st_middle(left,right) (((left)+(right))/2)
template<class T, class R, int MAXSIZE>
class lsegtree {
  void from_array(T* v, int idx, int left, int right) {
    refreshr[idx] = false;
    if (left != right) {
      from_array(v, st_left(idx), left, st_middle(left,right));
      from_array(v, st_right(idx), st_middle(left,right)+1, right);
      tree[idx] = tree[st_left(idx)] + tree[st_right(idx)]; //##//
    } else
      tree[idx] = v[left];
  }
  T read(int i, int j, int idx, int left, int right) {
    update(idx, left, right);
    if (i <= left && right <= j) return tree[idx];
    if (j < left || right < i) return nil;
    return read(i, j, st_left(idx), left, st_middle(left,right)) + //##//
      read(i, j, st_right(idx), st_middle(left,right)+1, right);
  }
  void set(int i, int j, R v, int idx, int left, int right) {
    update(idx, left, right);
    if (j < left || right < i) return;
    if (i <= left && right <= j) {
```

```
      refresh[idx] = v;
      refreshr[idx] = true;
      update(idx, left, right);
    } else {
      set(i, j, v, st_left(idx), left, st_middle(left,right));
      set(i, j, v, st_right(idx), st_middle(left,right)+1, right);
      tree[idx] = tree[st_left(idx)] + tree[st_right(idx)]; //##//
    }
  }
  void update(int idx, int left, int right) {
    if (refreshr[idx]) {
      if (left != right) {
        if (!refreshr[st_left(idx)]) refresh[st_left(idx)] = 0;
        if (!refreshr[st_right(idx)]) refresh[st_right(idx)] = 0;
        refresh[st_left(idx)] += refresh[idx]; //##//
        refresh[st_right(idx)] += refresh[idx]; //##//
        refreshr[st_left(idx)] = refreshr[st_right(idx)] = true;
      }
      tree[idx] += (right-left+1)*refresh[idx]; //##//
      refreshr[idx] = false;
    }
  }
public:
  T *tree; R *refresh; bool *refreshr; int size;
  lsegtree() {
    tree = new T[4*MAXSIZE];
    refresh = new R[4*MAXSIZE];
    refreshr = new bool[4*MAXSIZE];
  }
  inline void from_array(T array[]) { from_array(array, 0, 0, size-1); }
  inline T read(int i, int j) { return read(i, j, 0, 0, size-1); }
  inline void set(int i, int j, R v) { set(i, j, v, 0, 0, size-1); }
}; // int main() { lsegtree<int,int,MAXN> l; l.size = N; l.clear(); }
```

Union–Find

```
struct UnionFind {
  int *rank, *parent, size;
  UnionFind(int msize) { rank = new int[msize]; parent = new int[msize]; }
  void init(int msize) {
    size = msize;
    for (int i = 0; i < size; i++)
      parent[i] = i, rank[i] = 1;
  }
  int find(int node) {
    if (node == parent[node]) return node;
    return parent[node] = find(parent[node]);
  }
  void union_(int a, int b) {
    a = find(a), b = find(b);
    if (rank[a] <= rank[b]) parent[a] = b, rank[b] += rank[a];
    else parent[b] = a, rank[a] += rank[b];
  }
}; // int main() { UnionFind uf(MAXN); uf.init(n); }
```

7 Miscellaneous and C++ STL reference

Dates library

```
// Routines for performing computations on dates.  In these routines,
// months are expressed as integers from 1 to 12, days are expressed
// as integers from 1 to 31, and years are expressed as 4-digit ints.

string dayOfWeek[] = {"Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"};

// converts Gregorian date to integer (Julian day number)
int dateToInt (int m, int d, int y){
  return 1461 * (y + 4800 + (m - 14) / 12) / 4 +
    367 * (m - 2 - (m - 14) / 12 * 12) / 12 -
    3 * ((y + 4900 + (m - 14) / 12) / 100) / 4 + d - 32075;
}

// converts integer (Julian day number) to Gregorian date: month/day/year
void intToDate (int jd, int &m, int &d, int &y){
  int x, n, i, j;
  x = jd + 68569;
  n = 4 * x / 146097;
  x -= (146097 * n + 3) / 4;
  i = (4000 * (x + 1)) / 1461001;
  x -= 1461 * i / 4 - 31;
  j = 80 * x / 2447;
  d = x - 2447 * j / 80;
  x = j / 11;
  m = j + 2 - 12 * x;
  y = 100 * (n - 49) + i + x;
}
```

```
// converts integer (Julian day number) to day of week
string intToDay (int jd){
  return dayOfWeek[jd % 7];
}
```

Josephus problem

```
def josephus(n, k): # 1..n
  r, i = 0, 2
  while i <= n:
    r, i = (r + k) % i, i + 1
  return r + 1

def josephus2(n): # 1..n, k=2
  from math import log
  return 2*(n - 2**(int(log(n,2))))+1
```

STL reference

```
// QUEUE: empty, size, front, back, push, pop
// PRIORITY QUEUE: empty, size, top, push, pop
// STACK: empty, size, top, push, pop
// GENERIC (all below have at least these)
//   Iterators: begin, end, rbegin, rend
//   Capacity: empty, size, max_size
//   Modifiers: swap, clear, insert, erase

// SET
//   Operations: find, count, lower_bound, upper_bound, equal_range(=find)
// Returns <inserted element, created new(true) or already there(false)>
pair<iterator,bool> insert (const value_type& val);
// To properly take advantage of the hint:
// as begin(): val should be smaller than ALL elements in set
// as end(): val should be bigger than ALL elements in the set
// as h: h' <= val <= h, where h' is the element immediatly smaller than h
iterator insert (iterator position_hint, const value_type& val);
// Ex: s.insert(array,array+n); if [f,l) is sorted, O(n), not O(n lg n)!
template<class InputIterator> void insert(InputIterator f, InputIterator l);
void erase(iterator position);
size_type erase(const value_type& val); // returns if erased or not (0/1)
void erase(iterator first, iterator last); // removes [first,last)

iterator find(const value_type& val); // returns set::end if can't find
size_type count(const value_type& val); // basically, count(val) = has(val)
iterator lower_bound(const value_type& val); // iter to 1st element >= val
iterator upper_bound(const value_type& val); // iter to 1st element > val

// MAP
//   Operations: find, count, lower_bound, upper_bound, equal_range(=find)
//     All operations are identical to set's, but act on the key, not val!

// VECTOR
//   Access: [], at, front, back (at(n) == [n], but checks if n in [0,sz))
//   Capacity: resize, capacity, reserve
//   Modifiers: assign, push_back, pop_back

// Erases all elements; then, inserts all in [f,l), similar to set's insert
template<class InputIterator> void assign(InputIterator f, InputIterator l);
// Resizes vector to size n, then fills all elements with val
void assign (size_type n, const value_type& val);

// Inserts the element(s) immediatly BEFORE element in position
iterator insert(iterator position, const value_type& val);
void insert(iterator position, size_type n, const value_type& val);
template<class InputIterator>
  void insert(iterator position, InputIterator first, InputIterator last);

iterator erase(iterator position);
iterator erase(iterator first, iterator last);

// If v has less than n elements, expand to n and fill these with val
// If v has over n elements, just resize to n
void resize (size_type n, value_type val = value_type());

// DEQUE
//   Access: [], at, front, back (all O(1))
//   Capacity: resize
//   Modifiers: assign, push_back, push_front, pop_back, pop_front
//  Extremely similar to vector, but doesn't store ALL elements in sequence
//  (so careful about offseting a pointer), and inserts in begin in O(1).
```