Algorithm library

From BBSWiki

This is a collection of algorithms to be used as a library for ACM-ICPC contests... Please feel free to add any new algorithm or an improved implementation in the language of your choice... Also, please limit the line of code to be of length 100 characters maximum (Better if 80), in order to be able to print it easily, as the code plugin here does not work well for long lines and the printable version.

Contents

TODO's

- Finish Polishing the page
- Test algorithms
- Add more

"TO WRITE"S

Dynamic Programming

- Longest increasing subsequence
- Longest common subsequence

Graph Theory

- Bellman-Ford
- Maximal Bipartite Matching
- Maximal Bipartite Matching (Hopcroft)
- Prim's MST

Combinatorics

- Number of tree topological sortings
- Number of derrangements

Number Theory

- Modular arithmetic
- Prime factorization
- Camirchael Function
- Sum of squares check
- Chinese Remainder Solver

Geometry

- Polygon area
- Grid points inside polygon
- Line-line intersection/distance, point-line distance...
- Circle intersections and distances

Computational Geometry

The Point Struct

- Requires: none
- Represents: 2D Point/Vector
- Operations: +, -, *(dot product), cross product, comparison (least: left-most, bottom), norm (and squared)
- Time/Space Complexity: $O(1)$
- Status: MODERATELY TESTED

C++:

```
typedef int dim; //Type of coordinates

struct pnt
{
        dim x, y;
        pnt () {} pnt (dim xx, dim yy): x (xx), y (yy) {}
};

inline pnt operator+ (const pnt & a, const pnt & b)
{
        return pnt (a.x+b.x, a.y+b.y);
}

inline pnt operator- (const pnt & a, const pnt & b)
{
        return pnt (a.x-b.x, a.y-b.y);
}

inline dim operator* (const pnt & a, const pnt & b)
{
        return a.x*b.x + a.y*b.y;
}

inline dim cross (const pnt & a, const pnt & b)
{
        return a.x*b.y - a.y*b.x;
}

inline bool operator< (const pnt & a, const pnt & b)
{
        return (a.y == b.y) ? (a.x < b.x) : (a.y < b.y);
}
inline dim norm2 (const pnt & a) //norm squared
{
        return a.x*a.x + a.y*a.y;
}

inline double norm (const pnt & a)
{
        return sqrt (a.x*a.x + a.y*a.y);
}
```

Direction of Rotation

- Requires: The Point Struct
- Inputs: 3 points a, b and c

- Outputs: 1 if c is reached with a counter-clockwise rotation from a through b, -1 if the rotation is clockwise and 0 if all three are collinear
- Time/Space Complexity: $O(1)$
- Status: NOT TESTED

C++:

```
int ccw (const pnt & a, const pnt & b, const pnt & c)
{
        // It just assumes a direction
        dim r = cross (b-a, c-b);
        return (r == 0)? 0: (r > 0)? 1:-1;
}
```

Graham Scan Convex Hull

- Requires: The Point Struct, Direction of Rotation
- Inputs: list of points, NOTE: **input is destroyed**!
- Outputs: list of convex hull points in counter-clockwise order starting from left-most, bottom-most vertex
- Time Complexity: $O(N \log N)$ (or $O(N)$ if input points are sorted in counter-clockwise order from left-most, bottom-most vertex), where N is the number of input points
- Space Complexity: $O(N)$, where N is the number of input points
- Status: MODERATELY TESTED

C++:

```
bool ccw2 (const pnt & a, const pnt & b)
{
        dim d = cross (a, b);
        return (d == 0)? norm2(a) < norm2(b) : (d > 0);
}

vector <pnt> grahamScan (vector <pnt> p)
{
        if (p.size() < 2) return p;
        #define N ret.size()

        // Making the list unique and sorted//
        set <pnt> s (p.begin(), p.end());
        p = vector <pnt> (s.begin(), s.end());
       //////////////////////////////////////

        pnt pivot = p[0];
        for (int i = 0 ; i < p.size() ; ++i)
                { p[i].x -= pivot.x; p[i].y -= pivot.y; }
        sort (p.begin()+1, p.end(), ccw2);

        vector <pnt> ret;
        ret.push_back (p[0]); ret.push_back (p[1]);
        for (int i = 2 ; i < p.size() ; ++i)
        {
                while (N > 1 && ccw(ret[N-2], ret[N-1], p[i]) != 1)
                        ret.pop_back();
                ret.push_back (p[i]);
        }

        for (int i = 0 ; i < ret.size() ; ++i)
                { ret[i].x += pivot.x; ret[i].y += pivot.y; }

        return ret;
        #undef N
}
```

Graph Theory

Floyd-Warshall all-pair shortest path

- Requires: none.
- Inputs: Two dimensional (square) array dist, with dist[i][j] := length of edge connecting vertex i with j.
- Outputs: none, **replaces inputs**
- Time complexity: $O(n^3)$
- Space complexity: $O(1)$
- Status: NOT TESTED

Java:

```
public static void floyd_warshall (int [][] dist) {
           for(int k = 0 ; k < dist.length ; k++)
                for(int i = 0 ; i < dist.length ; i++)
                     for(int j = 0 ; j < dist.length ; j++)
                          dist[i][j] = Math.min(dist[i][j], dist[i][k] + dist[k][j]);
      }
```

C:

```
void floyd_warshall (int ** dist, int n) {
           for(int k = 0 ; k < n ; k++)
                for(int i = 0 ; i < n ; i++)
                     for(int j = 0 ; j < n ; j++)
                          dist[i][j] <?= dist[i][k] + dist[k][j];
      }
```

C++:

```
void floyd_warshall (vector< vector<int> > dist, int n) {
           for(int k = 0 ; k < n ; k++)
                for(int i = 0 ; i < n ; i++)
                     for(int j = 0 ; j < n ; j++)
                          dist[i][j] <?= dist[i][k] + dist[k][j];
      }
```

Bellman-Ford Single-Source Shortest Path

- Requires: none.
- Inputs: Two dimensional (square) array dist, with dist[i][j] := length of edge connecting vertex i with j.
- Outputs: none, **replaces inputs**
- Time complexity: $O(n^3)$
- Space complexity: $O(1)$
- Status: NOT TESTED

C++:

Kuhn-Munkres Maximum-Weight Bipartite Matching[1]

- Requires: none.
- Inputs: Two dimensional array with the cost[i][j] := the cost of matching item i with item j, N - the maximum of the sizes of the 2 graph parts.
- Outputs: the cost of the maximum-weight bipartite matching, **cost matrix is altered**, X[i] holds the item matched with i (or -1 if unmatched).

- Time complexity: $O(N^3)$
- Space complexity: $O(N)$
- Status: MODERATELY TESTED

C++:

```
//Inputs//////////////
int N;
int cost[MAX_N][MAX_N];
/////////////////////

int X[MAX_N], Y[MAX_N], Lx[MAX_N], Ly[MAX_N], Q[MAX_N], prev[MAX_N];
int maxw_bipartite() {
        int match() {
        int tail, s, k;
        memset(Ly, 0, sizeof(int)*N);
        for(int i = 0 ; i < N ; i++) Lx[i] = *max_element(cost[i], cost[i] + N);
        memset(X, -1, sizeof(int)*N);
        memset(Y, -1, sizeof(int)*N);
        for(int i = 0; i < N; i++) {
                int head;
                memset(prev, -1, sizeof(int)*N);
                for (Q[0] = i, head = 0, tail = 1; head < tail && X[i] < 0; head++) {
                        s = Q[head];
                        for(int j = 0; j < N ; j++) {
                                if(X[i] >= 0) break;
                                if (Lx[s] + Ly[j] > cost[s][j] || prev[j] >= 0) continue;
                                Q[tail++] = Y[j];
                                prev[j] = s;
                                if (Y[j] < 0) while (j >= 0) {
                                        s = prev[j]; Y[j] = s; k = X[s]; X[s] = j; j = k;
                                }
                        }
                }
                if(X[i] < 0 && i-- + (k = INF)) {
                        for(int head = 0 ; head < tail ; head++)
                    for(int j = 0 ; j < N ; j++)
                         if(prev[j] == -1) k = min(k, Lx[Q[head]] + Ly[j] - cost[Q[head]][j]);
                        for(int j = 0; j < tail ; j++) Lx[Q[j]] -= k;
                        for(int j = 0; j < N ; j++) if (prev[j] >= 0) Ly[j] += k;
                }
        }
        int sum = 0;
        for(int i = 0 ; i < N ; i++) if(X[i] >= 0) sum += cost[i][X[i]];
        return sum;
}
```

Java:

```
static final int INF = 2000000000;
        public static int maxw_bipartite(int N, int cost[][], int X[]) {
                int Y[] = new int[N], Lx[] = new int[N], Ly[] = new int[N],
                        Q[] = new int[N+1], prev[] = new int[N], tail, k, s;

                for(int i = 0 ; i < N ; i++)
                        for(int j = 1, Lx[i] = cost[i][0] ; j < N ; j++)
                                Lx[i] = Math.max(Lx[i], cost[i][j]);
                Arrays.fill(X, -1);
                Arrays.fill(Y, -1);
                Arrays.fill(Ly, 0);

                for(int i = 0; i < N; i++) {
                        int head;
                        Arrays.fill(prev, -1);
                        for (Q[0] = i, head = 0, tail = 1; head < tail && X[i] < 0; head++) {
                                s = Q[head];
                                for(int j = 0; j < N ; j++) {
                                        if(X[i] >= 0) break;
                                        if (Lx[s] + Ly[j] > cost[s][j] || prev[j] >= 0) continue;
                                        Q[tail++] = Y[j];
                                        prev[j] = s;
                                        if (Y[j] < 0) while (j >= 0) {
                                                s = prev[j]; Y[j] = s; k = X[s]; X[s] = j; j = k;
                                        }
                                }
                        }
                        if(X[i] < 0 && (i-- + (k = INF)) > 0) {
                                for(head = 0 ; head < tail ; head++)
                                        for(int j = 0 ; j < N ; j++)
                                                if(prev[j] == -1) k = Math.min(
                                                        k, Lx[Q[head]] + Ly[j] - cost[Q[head]][j]);
                                for(int j = 0; j < tail ; j++) Lx[Q[j]] -= k;
                                for(int j = 0; j < N ; j++) if (prev[j] >= 0) Ly[j] += k;
                        }
                }

                int sum = 0;
                for(int i = 0 ; i < N ; i++) if(X[i] >= 0) sum += cost[i][X[i]];
                return sum;
        }
```

Mincost flow

Java:

```
public class MinCostFlowBellmanFord {
        static int inf=1000000000;
        public static int[] minCostFlow(int[][] U,int[][] C,int s,int d){
                int n=C.length;
                int[][] u =new int[n][n];
                int[][] c =new int[n][n];
                for(int i=0;i<n;i++)
                        for(int j=0;j<n;j++){
                                u[i][j]=U[i][j]; c[i][j]=C[i][j];
                        }
                int[] ans=new int[2];
                while(true){
                        int[] path=bellmanFord(u,c,s);
                        if(path[d]==-1) break;
                        ans[1]+=augment(u,c,s,d,path);
                }
                for(int i=0;i<n;i++)
                        for(int j=0;j<n;j++)
                                if(U[i][j]>u[i][j])
                                        ans[0]+=(U[i][j]-u[i][j])*C[i][j];
                return ans;
        }
        public static int augment(int[][] u,int[][] c,int s,int d,int[] p){
                int min=inf;
                for(int i=d;p[i]!=-1;i=p[i])
                        min=Math.min(min,u[p[i]][i]);
                for(int i=d;p[i]!=-1;i=p[i]){
                        u[p[i]][i]-=min; u[i][p[i]]+=min;
                        c[i][p[i]]-=c[p[i]][i];
                }
                return min;
        }
```

```
public static int[] bellmanFord(int[][] u,int[][] c,int s){
                int n=c.length;
                int[] d=new int[n];
                int[] p=new int[n];
                Arrays.fill(d,inf);
                Arrays.fill(p,-1);
                d[s]=0;
                for(int i=0;i<n;i++)
                        for(int j=0;j<n;j++)
                                for(int k=0;k<n;k++)
                                        if(u[j][k]>0&&d[k]>d[j]+c[j][k]){
                                                d[k]=d[j]+c[j][k]; p[k]=j;
                                        }
                return p;
        }
        public static int[] solveMin(int[][] U,int[][] C,int[] b){
                int n=U.length;
                int[][] u=new int[n+2][n+2];
                int[][] c=new int[n+2][n+2];
                int flow=0;
                for(int i=1;i<=n;i++)
                        if(b[i-1]>0){
                                u[0][i]=b[i-1]; c[0][i]=0;
                        }
                        else{
                                u[i][n+1]=-b[i-1]; c[i][n+1]=0;
                        }
                for(int i=0;i<n;i++)
                        for(int j=0;j<n;j++){
                                c[i+1][j+1]=C[i][j]; u[i+1][j+1]=U[i][j];
                        }
                int[] a=minCostFlow(u,c,0,n+1);
                return a;
        }
        public static int[] solveMax(int[][] U,int[][] C,int[] b){
                int n=U.length;
                int[][] u=new int[n+2][n+2];
                int[][] c=new int[n+2][n+2];
                int flow=0;
                for(int i=1;i<=n;i++)
                        if(b[i-1]>0){
                                u[0][i]=b[i-1]; c[0][i]=0;
                        }
                        else{
                                u[i][n+1]=-b[i-1]; c[i][n+1]=0;
                        }
                for(int i=0;i<n;i++)
                        for(int j=0;j<n;j++){
                                c[i+1][j+1]=-C[i][j]; u[i+1][j+1]=U[i][j];
                        }
                int[] a=minCostFlow(u,c,0,n+1);
                a[0]=-a[0];
                return a;
        }
}
```

Dijkstra's Algorithm with Binary Heap

C++:

```
int n;
map<string, int> conv;

int nei[10000];
int * adj[10000];
int * cost[10000];

char temp[10];

int heap[10000], dist[10000], pos[10000];

inline void swape(const int & i, const int & j)
{
        pos[heap[i]] = j;
        pos[heap[j]] = i;
        int tmp = heap[i];
        heap[i] = heap[j];
        heap[j] = tmp;
}

inline void heapify(const int & i, const int & qsize)
{
        for(int curr = i, nxt = 2*i + 1; nxt < qsize ; curr = nxt, nxt = 2*curr + 1)
        {
                if(nxt + 1 < qsize && dist[heap[nxt+1]] < dist[heap[nxt]]) nxt++;
                if(dist[heap[curr]] <= dist[heap[nxt]]) break;
                swape(curr, nxt);
        }
}

inline int extract(int & qsize)
{
        int res = heap[0];
        swape(0, qsize-1);
        qsize--;
        heapify(0, qsize);
        return res;
}

inline void decrkey(int pos, const int & key)
{
        for(dist[heap[pos]] = key; pos && dist[heap[(pos-1)/2]] > dist[heap[pos]]; pos = (pos-1)/2)
```

```
                swape(pos, (pos-1)/2);
}

int dijkstra(const int & from, const int & to)
{
        FOR(i, n)
        {
                heap[i] = pos[i] = i;
                dist[i] = INF;
        }
        decrkey(from, 0);

        for(int qsize = n, curr; qsize;)
        {
                if(heap[0] == to) return dist[to];
                curr = extract(qsize);
                FOR(i, nei[curr])
                        if(dist[curr] + cost[curr][i] < dist[adj[curr][i]])
                        {
                                dist[adj[curr][i]] = dist[curr] + cost[curr][i];
                                decrkey(pos[adj[curr][i]], dist[adj[curr][i]]);
                        }
        }
        return -1;
}
```

Kruskal's algorithm with disjoint-set data structure

```
#define MAX_V 10000
#define MAX_E 1000000

int n, m, pi[MAX_V], rank[MAX_V];

struct edge {
        int from, to, cost;
};

edge E[MAX_E];

inline bool operator<(const edge& a, const edge& b) {
        return a.cost < b.cost; // return a.cost > b.cost;
}

inline int find(int v) {
        return (pi[v] == v)? v : pi[v] = find(pi[v]);
}

void unite(int p1, int p2) {
        if(rank[p1] > rank[p2]) pi[p2] = p1;
        else if(rank[p1] < rank[p2]) pi[p1] = p2;
        else {
                pi[p1] = p2;
                rank[p2]++;
        }
}

int kruskal() {
        sort(E, E + m); //make_heap(E, E + m);
        FOR(i, n) pi[i] = i;
        FOR(i, n) rank[i] = 0;
        int es = 0, c = 0;
        FOR(e, m) {
                if(es >= n-1) break;
                int f = find(E[e].from), t = find(E[e].to); // E[0]
                if(f != t) {
                        unite(f, t);
                        c += E[e].cost; // E[0]
                        es++;
                }
                // pop_heap(E, E + m - e);
        }
        if(es < n-1) return -1;
        return c;
}
```

Number of spanning trees (Kirchoff's Theorem)

```
import java.math.BigInteger;
import java.io.*;

class Main
{
        static int n;
        static int m;
        static BigInteger mat[][] = new BigInteger[15][15];
        public static void main(String args[]) throws IOException
        {
                BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
                int t = Integer.parseInt(br.readLine());
                for(int u= 0; u < t; u++)
                {
                        String temp = br.readLine();
                        String [] tmp = temp.split(" ");
                        n = Integer.parseInt(tmp[0]);
                        m = Integer.parseInt(tmp[1]);
                        int a, b;
                        BigInteger NONE = BigInteger.valueOf(-1);
                        for(int i = 0 ; i < n ; i++)
                                for(int j = 0 ; j < n ; j++)
                                        mat[i][j] = BigInteger.ZERO;
                        for(int i = 0 ; i < m ; i++)
                        {
                                temp = br.readLine();
                                tmp = temp.split(" ");
                                a = Integer.parseInt(tmp[0]);
                                b = Integer.parseInt(tmp[1]);
                                a--;
                                b--;
                                mat[a][b] = NONE;
                                mat[b][a] = NONE;
                                mat[a][a] = mat[a][a].add(BigInteger.ONE);
                                mat[b][b] = mat[b][b].add(BigInteger.ONE);
                        }
                        br.readLine();
                        n--;
                        System.out.println(det());
                }
        }
}
```

Strongly connected components

Input: Graph G = (V, E), Start node v0

index = 0 // DFS node number counter S = empty // An empty stack of nodes tarjan(v0) // Start a DFS at the start node

procedure tarjan(v)

```
v.index = index               // Set the depth index for v
v.lowlink = index
index = index + 1
S.push(v)                     // Push v on the stack
forall (v, v') in E do        // Consider successors of v
  if (v'.index is undefined)  // Was successor v' visited?
    tarjan(v')                // Recurse
    v.lowlink = min(v.lowlink, v'.lowlink)
  elseif (v' in S)            // Is v' on the stack?
    v.lowlink = min(v.lowlink, v'.lowlink)
if (v.lowlink == v.index)     // Is v the root of an SCC?
  print "SCC:"
  repeat
    v' = S.pop
    print v'
  until (v' == v)
```

Stable marriage problem

```
function stableMatching {
    Initialize all m ∈ M and w ∈ W to free
    while ∃ free man m who still has a woman w to propose to {
       w = m's highest ranked such woman
       if w is free
         (m, w) become engaged
       else some pair (m', w) already exists
         if w prefers m to m'
           (m, w) become engaged
           m' becomes free
         else
           (m', w) remain engaged
    }
}
```

All pair max-flow/min-cut

```
int parent[n]; //initialized to 0
int answer[n][n]; //initialize this one to infinity
for(int i=1;i<n;++i){
        //Compute the minimum cut between i and parent[i].
        //Let the i-side of the min cut be S, and the value of the min-cut be F
        for (int j=i+1;j<n;++j)
                if ((j is in S) && parent[j]==parent[i])
                        parent[j]=i;
        answer[i][parent[i]]=answer[parent[i]][i]=F;
        for (int j=0;j<i;++j)
                answer[i][j]=answer[j][i]=min(F,answer[t][j]);
}
```

Number Theory

Sieve of Eratosthenes

- Not Tested after the last modification
- Requires: none.
- Inputs: N - upper bound
- Outputs: boolean array prime, prime[i] is true iff i is prime.
- Time complexity: $O(N log(N) loglog(N))$
- Space complexity: $O(N)$

Java:

```
public static boolean[] sieve__of_eratosthenes(int N) {
            boolean[] prime = new boolean[N];
            Arrays.fill(prime, true);
            prime[0] = prime[1] = false;
            for(int i = 2 ; i*i <= N ; i++)
                if(prime[i])
                    for(int j = i*i ; j < N ; j += i)
                          prime[i] = false;
            return prime;
       }
```

C:

```
bool* sieve__of_eratosthenes(int N) {
            bool *prime = new bool[N];
            memset(prime, false, N*sizeof(bool));
            prime[0] = prime[1] = false;
            for(int i = 2 ; i*i <= N ; i++)
                if(prime[i])
```

```
                    for(int j = i*i ; j < N ; j += i)
                          prime[i] = false;
            return prime; //'''Don't Forget "delete[] prime;"'''
       }
```

C++:

```
vector<bool> sieve__of_eratosthenes(int N) {
            vector<bool> prime(N, true);
            prime[0] = prime[1] = false;
            for(int i = 2 ; i*i <= N ; i++)
                if(prime[i])
                    for(int j = i*i ; j < N ; j += i)
                          prime[i] = false;
            return prime;
       }
```

Impressively Fast Sieve

Thanks to Hassan for digging out this code out of Yarin :).

```
#include <cstdio>
#include <algorithm>
using namespace std;

#define MAXSIEVE 100000000 // All prime numbers up to this
#define MAXSIEVEHALF (MAXSIEVE/2)
#define MAXSQRT 5000 // sqrt(MAXSIEVE)/2
char a[MAXSIEVE/16+2];
#define isprime(n) (a[(n)>>4]&(1<<(((n)>>1)&7))) // Works when n is odd

void doit()
{
        int i,j;
        memset(a,255,sizeof(a));
        a[0]=0xFE;
        for(i=1;i<MAXSQRT;i++)
                if (a[i>>3]&(1<<(i&7)))
                for(j=i+i+i+1;j<MAXSIEVEHALF;j+=i+i+1)
                        a[j>>3]&=~(1<<(j&7));
}

int main ()
{
doit();

for (int i = 3 ;  i <= 100 ; i += 2)
        if (isprime(i)) printf("%d\n", i);

return 0;
}
```

Impressively Fast Sieve 2

```
#include <stdio.h>
int BIG = 32000;
int MAXI = 1000000000;

// count primes from 1 to MAXI
int main() {
  bool *isprime = new bool[BIG];
  int *primes = new int[BIG];
  int *offset = new int[BIG];
  int nbprimes = 0;
  // Eratosthene's slieve for 1..BIG
  for(int i = 0; i < BIG; i++) isprime[i] = true;
  for(int i = 2; i < BIG; i++) if(isprime[i]) {
    primes[nbprimes] = i;
    int j;
    for(j = i + i; j < BIG; j += i) isprime[j] = false;
    offset[nbprimes] = j - BIG;
    nbprimes++;
  }
  // Count primes in ranges of length BIG, starting from BIG until MAXI
  int count = nbprimes;
  for(int START = BIG; START < MAXI; START += BIG) {
    for(int i=0; i<BIG; i++) isprime[i] = true;
    for(int j=0; j<nbprimes; j++) {
      int p = primes[j];
      int o;
      for(o = offset[j]; o < BIG; o += p) isprime[o] = false;
      offset[j] = o - BIG;
    }
    for(int i = 0; i + START < MAXI && i < BIG; i++) if(isprime[i]) {
      count++;
      // Here you got your number prime, START+i
      //fprintf(stderr,"%d\n", START+i);
    }
  }

  printf("There are %d primes in [2 .. %d]\n", count, MAXI);
  return 0;
}
```

Extended Euclidean Algorithm

- Requires: none.
- Inputs: a, b - the two numbers.
- Outputs: a pair (s, t) such that $as + bt = \gcd(a, b)$.
- Time complexity: $O(log(\max(a, b)))$
- Space complexity: $O(log(\max(a, b)))$

Java:

```
public static int[] extended_euclid(int a, int b) {
            if(a % b == 0) return new int[]{0, 1};
            int [] r = extended_euclid(b, a % b);
            return new int[]{r[1], r[0]-r[1]*a/b};
       }
```

C:

```
void extended_euclid(int a, int b, int * s, int * t) {
            if(a % b == 0) {*s = 0; *t = 1;}
            int x, y;
            extended_euclid(b, a % b, &x, &y);
            *s = y;
            *t = x - y*a/b;
       }
```

C++:

```
pair<int, int> extended_euclid(int a, int b) {
            if(a % b == 0) return make_pair(0, 1);
            pair<int, int> r = extended_euclid(b, a % b);
            return make_pair(r.second, r.first - r.second*a/b);
       }
```

Modular Binary Exponentiation

- Requires: none.
- Inputs: a, b, mod
- Outputs: a^b under mod

- Time complexity: $O(log(b))$
- Space complexity: $O(1)$

C++:

```
long long po(long long a, long long b, long long mod) {
    long long r = 1;
    a %= mod;
    do {if (b&1) r=(r*a)%mod; a=(a*a)%mod;} while (b>>=1);
    return r;
}
```

Multiplicative Inverse

- Requires: none.
- Inputs: n - the number to calculate its inverse, p - the prime number.
- Outputs: the multiplicative inverse of n (mod p)
- Time complexity: $O(log(p))$
- Space complexity: $O(1)$

C++:

```
long long mult_inverse (long long n, long long p)
{
    long long r = 1, e = p-2;
    do {if (e&1) r=(r*n)%p; n=(n*n)%p;} while (e>>=1);
    return r;
}
```

Euler's totient function

represents the number of integers less than n which are relatively prime t it Java:

```
public static int totient(int n){
                int result=n;
                for(int i=2;i*i<=n;i++){
                        if(n%i==0)      result-=(result/i);
                        while(n%i==0)   n/=i;
                }
                if(n>1) result-=(result/n);
                return result;
        }
```

Combinatorics

The Unknown Formula

The number of combinations of multisets to be chosen from a set. If the set is of size N (the number of items to choose from), the multiset size to be chosen is K, then the number of combinations is: $\binom{N+K-1}{K}$

Example: number of expected results when throwing K dice, each of them has N-sides.

Combinations and Permutations

TESTED

C++:

```
long long Cdp[1024][1024];

// Add this code before execution:
for (int i = 0 ; i < 1024 ; ++i)
        for (int j = 0 ; j < 1024 ; ++j)
                Cdp[i][j] = -1;

long long C (int n, int k)
{
        if (k == 0 || n == k) return 1;
        if (Cdp[n][k] != -1) return Cdp[n][k];
        return Cdp[n][k] = C (n-1, k) + C (n-1, k-1);
}


long long P (int n, int k)
{
        long long r = 1;
        while (k--)
                r *= n--;
        return r;
}
```

Next permutation

Java:

```
public class NextPermutation {
        public static void reverse(int[] a,int beg,int end){
                int x=beg,y=end-1,temp;
                while(x<y){
                        temp=a[x];
                        a[x]=a[y];
                        a[y]=temp;
                        x++; y--;
                }
        }
        public static boolean nextPerm(int[] a){
                if(a.length==0||a.length==1)
                        return false;
                int i=a.length-1;
                for(;;){
                        int ii=i--;
                        if(a[i]<a[ii]){
                                int j=a.length;
                                while(a[i]>=a[--j]);
                                int temp1=a[i];
                                a[i]=a[j];
                                a[j]=temp1;
                                reverse(a,ii,a.length);
                                return true;
                        }
                        if(i==0){
                                reverse(a,0,a.length);
                                return false;
                        }
                }
        }
}
```

Looping over combinations of size k from the set of size N

This loops the bitmask s over all subsets of size k of the set of size N

TESTED

C++:

```
// Loop over all combinations of size k from set of size N
int s = (1 << k) - 1;
while (!(s & 1 << N))
{
        // Here you have the bitmask (s)

        int lo = s & ~(s - 1);       // lowest one bit
        int lz = (s + lo) & ~s;      // lowest zero bit above lo
        s |= lz;                     // add lz to the set
        s &= ~(lz - 1);              // reset bits below lz
        s |= (lz >> ffs(lo)) - 1;    // put back right number of bits at end
}
```

Java:

```
// Loop over all combinations of size k from set of size N
int s = (1 << k) - 1;
while ((s & (1 << N)) == 0)
{
        // Here you have the bitmask (s)

        int lo = s & ~(s - 1);       // lowest one bit
        int lz = (s + lo) & ~s;      // lowest zero bit above lo
        s |= lz;                     // add lz to the set
        s &= ~(lz - 1);              // reset bits below lz
        if (lo == 0) break;
        s |= (lz / lo / 2) - 1;
}
```

Partition

Java:

```
public class Partition {
        static long[][] dp=new long[101][101];
        //represents the no. of ways to partition n where the
        //smallest no. is k
        public static long p(int n,int k){
                if(k>n) return 0;
                if(k==n)        return 1;
                if(dp[n][k]!=-1)        return dp[n][k];
                return dp[n][k]=p(n,k+1)+p(n-k,k);
        }
        //represents the no. of ways to write n as a sum of
        //positive integers
        public static long f(int n){
                return p(n,1);
        }
}
```

Factoring permutation into cycles

```
FOR(i, 26) vis[i] = false;
        FOR(i, 26)
        {
                if(vis[i]) continue;
                int curr = i;
                vis[i] = true;
                set<int> cyc;
                cyc.insert(i);
                do
                {
                        curr = S[curr] - 'A';
                        cyc.insert(curr);
                        vis[curr] = true;
                } while(curr != i);
                cycles.push_back(cyc);
        }
```

Miscellaneous

Problem Virtual Friends

Hashing and MST with path collapse and rank by size and size query.

```
#include <stdio.h>
#include <memory.h>

int pre [600000], cnt [600000], temp [600000], tp = 0;

inline void insert (const int vertex) {
        pre[vertex] = vertex;
        cnt[vertex] = 1;
}

inline int rep (int vertex) {
        int p1 = pre[vertex], p2 = pre[pre[vertex]];
        if (p1 == p2) return p1;
        while (p2 != p1) {
                temp[tp++] = vertex;
                vertex = p1;
                p1 = p2;
                p2 = pre[p2];
        }
        while (tp) {
                int n = temp[--tp];
                cnt[pre[n]] -= cnt[n];
                pre[n] = p1;
        }
        return p1;
}

inline void unify (const int vertexA, const int vertexB) {
        int pA = rep(vertexA);
        int pB = rep(vertexB);
        if (pA == pB) return;
        if (cnt[pA] > cnt[pB]) {int temp = pA; pA = pB; pB = temp;}
        cnt[pB] += cnt[pA];
        pre[pA] = pB;
}

inline int partitionSize (const int v) {
        return cnt[rep(v)];
}


long long p [600000];
int size;

inline void hashtable (const int s) {
        size = s;
        memset (p, -1, (s+100)*sizeof(long long));
}
```

```
inline int val (const long long n) {
        int m = (n < 0) ? (-n) % size : n % size;
        while (p[m] != -1)
                if (p[m] == n) return m;
                else m++;
        p[m] = n;
        insert (m);
        return m;
}
```

```
int main() {
        int k;
        scanf ("%d", &k);
        while (k--) {
                int n;
                scanf ("%d", &n);

                char c1 [21], c2[21];
                hashtable (n*4);
                while (n--) {
                        scanf ("%s%s", c1, c2);
                        long long s1 = 0, s2 = 0;
                        char *p;
                        for (p = c1 ; *p ; ++p)
                                s1 = (((s1^*p) << 7) ^ (s1 >> 57));
                        for (p = c2 ; *p ; ++p)
                                s2 = (((s2^*p) << 7) ^ (s2 >> 57));
                        int n1 = val(s1), n2 = val(s2);
                        unify (n1, n2);
                        printf ("%d\n", partitionSize(n2));
                }
        }
        return 0;
}
```

C/C++ Counting the One Bits

__builtin_popcount ()

C/C++ Hash Table

P.S.: Some functions aren't well tested

```
long long p [600000];
int size;
long long inf = 1000000000000000000LL;

inline void hashtable (const int s)
{
        size = s;
        memset (p, -1, (s+100)*sizeof(long long));
        // reserve more in order not to wrap around the table
}

inline bool find (const long long n)
{
        int m = (n < 0) ? (-n) % size : n % size;
        while (p[m] != -1)
                if (p[m] == n) return true;
                else m++;
        return false;
}

inline void insert (const long long n)
{
        int m = (n < 0) ? (-n) % size : n % size;
        while (p[m] != -1)
                if (p[m] == n) return;
                else m++;
        p[m] = n;
}

inline void remove (const long long n)
// Efficient if not used too much
{
        int m = (n < 0) ? (-n) % size : n % size;
        while (p[m] != -1)
                if (p[m] == n)
                {
                        p[m] = inf;
                        return;
                }
                else m++;
}

inline int insert_and_return_hash (const long long n)
// Useful if needed to replace maps that map strings to integers
// to simplify the input. It just extends the domain of the problem
// to larger numbers by factor of 2 or 3, yet really useful if the
// memory isn't the first objective rather than speed
{
        int m = (n < 0) ? (-n) % size : n % size;
        while (p[m] != -1)
                if (p[m] == n) return m;
                else m++;
        p[m] = n;
        return m;
}
```

C/C++ Hash Table 2

P.S.: The remove and insert and return key aren't well tested

```
#include <cstring>
using namespace std;

typedef int typ;
const int HN = 250000;
const int HS = HN+HN/10;
typ hh[HS];

int hinsert (typ n)
{
        typ m = n % HN;
        while (hh[m] != -1)
                m++;
        hh[m] = n;
        return m;
}

bool hfind (typ n)
{
        typ m = n % HN;
        while (hh[m] != -1 && hh[m] != n)
                m++;
        return hh[m] == n;
}

bool hrem (typ n)
{
        typ m = n % HN;
        while (hh[m] != -1 && hh[m] != n)
                m++;
        if (hh[m] == -1) return false;
        hh[m] = -2;
        return true;
}

void hclear ()
{
        memset (hh, -1, sizeof(int)* HS);
}
```

Hashing Strings

P.S.: Doesn't guaranty absolute correctness, but works best on normal cases

```
long long hash = 0;
char *p;                // The string
for (p = c1 ; *p ; ++p)
        hash = (((hash^*p) << 7) ^ (hash >> 57));
```

Logarithmic Matrix Exponentiation

```
#define FOR(i, n) for(int i = 0; i < int(n) ; i++)
int n;
int mat[50][50], T[50][50], T2[50][50];

// R = R*M
void mul (int R[][50], int M[][50])
{
        memset(T2, 0, sizeof(T2));

        FOR(i, n) FOR(j, n) FOR(k, n)
                T2[i][j] = (T2[i][j] + R[i][k] * M[k][j]) % MOD;

        FOR(i, n) FOR(j, n) R[i][j] = T2[i][j];
}

// R = M ^ rr
void mat_pow (int R[][50],  int M[][50], int rr) {
        FOR(i, n) FOR(j, n) T[i][j] = (i == j);
        do {
                if (rr&1) mul(T, M);
                mul (M, M);
        } while (rr>>=1);
        FOR(i, n) FOR(j, n) R[i][j] = T[i][j];
}
```

Determinant using Gaussian Elimination

Java:

```
static int n;
        static int m;
        static BigInteger mat[][] = new BigInteger[15][15];
        static boolean swap(int row)
        {
                for(int i = row+1 ; i < n ; i++)
                        if(!mat[i][row].equals(BigInteger.ZERO))
                        {
                                for(int j = 0 ; j < n ; j++)
                                {
                                        //swap(mat[row][j], mat[i][j])
                                        BigInteger temp = mat[row][j];
                                        mat[row][j] = mat[i][j];
                                        mat[i][j] = temp;
                                }
                                return true;
                        }
                return false;
        }
        static String det()
        {
                if(n == 0) return "1";
                BigInteger res = BigInteger.ONE, accum = BigInteger.ONE;
                int sign = 1;
                for(int row = 0 ; row + 1 < n ; row++)
                {
                        if(mat[row][row].equals(BigInteger.ZERO))
                                if(!swap(row)) return BigInteger.ZERO.toString();
                                else sign *= -1;
                        for(int i = row+1 ; i < n ; i++)
                        {
                                BigInteger mult1 = mat[row][row].divide
                                        (mat[row][row].gcd(mat[i][row]));
                                BigInteger mult2 = mat[i][row].divide
                                        (mat[i][row].gcd(mat[row][row]));
                                accum = accum.multiply(mult1);
                                for(int j = row ; j < n ; j++)
                                        mat[i][j] = (mat[i][j].multiply(mult1)).subtract
                                                (mat[row][j].multiply(mult2));
                        }
                        res = res.multiply(mat[row][row]);
                        BigInteger G = res.gcd(accum);
                        res = res.divide(G);
                        accum = accum.divide(G);
                }
                res = res.multiply(mat[n-1][n-1]);
                return ((BigInteger.valueOf(sign)).multiply(res)).divide(accum).toString();
        }
```

KMP

```
#include <iostream>
#include <vector>
using namespace std;

//----------------------------
//Returns a vector containing the zero based index of
//  the start of each match of the string K in S.
//  Matches may overlap
//----------------------------
vector<int> KMP(string S, string K)
{
        vector<int> T(K.size() + 1, -1);
        for(int i = 1; i <= K.size(); i++)
        {
                int pos = T[i - 1];
                while(pos != -1 && K[pos] != K[i - 1]) pos = T[pos];
                T[i] = pos + 1;
        }

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

        return matches;
}
```

Nice Facts

The grey-code of a number **n** is (n ^ (n >> 1)) [2]

Hmm, what about the inverse?

The graph can not be bicolored or bipartite matched if and only if it has an odd cycle.

The graph is not planar if and only if it contains K5 or K3,3 as subgraphs of the graph resulting from repeating this operations till the max node degree is > 2:

Deleting any node of degree one with its edge.

Replacing any node of degree 2 by a direct edge (Also deleting node and edges).

Every finite, simple, planar graph has a node of degree less than 6.

If the graph is planar, connected with n nodes, m edges, then it divides the plane into f subplanes such that: f = m – n + 2

Every planar graph on nine nodes has a nonplanar complement.

Number of edges (m) in a planar graph with (n) nodes:

Theorem 1. If $n \geq 3$ then $m \leq 3n$ 6

Theorem 2. If $n > 3$ and there are no cycles of length 3, then $m \leq 2n$ 4

A connected graph has an Eulerian trail (Euler's path) if and only if it has at most two nodes of odd degree. (Two odd degree nodes or no odd degree node) If it has two odd degree nodes, then the path begins from an odd degree node and ends on the other one. Otherwise it begins from any node and ends in the same beginning node.

Area of a triangle:

A = abs((x1-x2)*(y1-y3)-(x1-x3)*(y1-y2)); //Cross Multiplication.

A = sqrt (p*(p-a)*(p-b)*(p-c)); // a, b, c are side lengths and p = (a + b + c) / 2

Inradius of a triangle:

r = 2 * Area of the triangle / Perimeter = 4*Circumradius*sin(A/2)*sin(B/2)*sin(C/2)

Yet another Euler's Formula: The distance between the circumcenter and the incenter, d is: d^2 = R(R-2r), R is the circumradius, r is the inradius

Area of an integer coordinate polygon (Lattice Polygon) P:

A = Number of interior points in P + ½ Number of boundary points of P - 1

Geometry Library 2.0

```
#include <vector>
#include <set>
#include <algorithm>
#include <cmath>
#include <iostream>
using namespace std;

typedef long long ll; //Type of coordinates

struct pt
{
        ll x, y;
        pt () {} pt (ll xx, ll yy): x (xx), y (yy) {}
};

inline pt operator+ (const pt & a, const pt & b)
{
        return pt (a.x+b.x, a.y+b.y);
}

inline pt operator- (const pt & a, const pt & b)
{
        return pt (a.x-b.x, a.y-b.y);
}

inline ll operator* (const pt & a, const pt & b)
{
        return a.x*b.x + a.y*b.y;
}
```

```
inline ll cross (const pt & a, const pt & b)
{
        return a.x*b.y - a.y*b.x;
}

inline bool operator< (const pt & a, const pt & b)
{
        return (a.y == b.y) ? (a.x < b.x) : (a.y < b.y);
}

inline ll norm2 (const pt & a) //norm squared
{
        return a.x*a.x + a.y*a.y;
}

inline double norm (const pt & a)
{
        return sqrt (a.x*a.x + a.y*a.y);
}
```

```
int ccw (const pt & a, const pt & b, const pt & c)
{
        // It just assumes a direction
        ll r = cross (b-a, c-b);
        return (r == 0)? 0: (r > 0)? 1:-1;
}

// The area, doubled
ll area2 (const vector<pt> & p)
{
        int N = p.size();
        if (N < 3) return 0;
        ll ret = 0;
        pt p0 = p[0];
        N--;
        for (int i = 1 ; i < N ; ++i)
                ret += cross(p[i]-p0, p[i+1]-p0);
        return abs(ret);
}

double circ (const vector<pt> & p)
{
        if (p.size() < 2) return 0.0;
        double ret = 0.0;
        for (int i = 1 ; i < p.size() ; ++i)
                ret += norm(p[i]-p[i-1]);
        ret += norm (p.back()-p.front());
        return ret;
}
```

```
bool ccwGS (const pt & a, const pt & b)
{
        ll d = cross (a, b);
        return (d == 0)? norm2(a) < norm2(b) : (d > 0);
}

vector <pt> grahamScan (vector <pt> p)
{
        if (p.size() < 2) return p;
        // Making the list unique and sorted//
        set <pt> s (p.begin(), p.end());
        p = vector <pt> (s.begin(), s.end());
    //////////////////////////////////////

        pt pivot = p[0];
        for (int i = 0 ; i < p.size() ; ++i)
                { p[i].x -= pivot.x; p[i].y -= pivot.y; }
        sort (p.begin()+1, p.end(), ccwGS);

        vector <pt> ret;
        #define N ret.size()

        ret.push_back (p[0]); ret.push_back (p[1]);
        for (int i = 2 ; i < p.size() ; ++i)
        {
                while (N > 1 && ccw(ret[N-2], ret[N-1], p[i]) != 1)
                        ret.pop_back();
                ret.push_back (p[i]);
        }

        for (int i = 0 ; i < ret.size() ; ++i)
                { ret[i].x += pivot.x; ret[i].y += pivot.y; }

        return ret;
        #undef N
}

// Doesn't handle colinear points
bool is_intersecting (pt p1, pnt p2, pnt r1, pnt r2)
{
        return cross (r2-p1, p2-p1) * cross (p2-p1, r1-p1) >= 0 &&
                cross (p2-r1, r2-r1) * cross (r2-r1, p1-r1) >= 0;
}
```

```
struct fpt
{
        ll x, y, d;
        fpt () {}
        fpt (ll xx, ll yy, ll dd)
        {
                ll g = __gcd(xx, yy);
                d = dd * g;
                x = xx / g;
                y = yy / g;
        }

        double get_x() {return double(x) / double(d);}
        double get_y() {return double(y) / double(d);}
};

bool operator< (const fpt & a, const fpt & b)
{
        if (a.x*b.d == b.x*a.d) return a.y*b.d < b.y*a.d;
        return a.x*b.d < b.x*a.d;
}

fpt intersection_point (const pnt & p1, const pnt & p2, const pnt & p3, const pnt & p4)
{
        int d = (p4.y-p3.y)*(p2.x-p1.x)-(p2.y-p1.y)*(p4.x-p3.x);
        int n = (p3.x-p1.x)*(p4.y-p3.y)-(p4.x-p3.x)*(p3.y-p1.y);
        int x = (p2.x-p1.x)*n, y = (p2.y-p1.y)*n;
        x += p1.x*d; y += p1.y*d;
        fpt ret (x, y, d);
        return ret;
}

double dis (const fpt & a, const fpt & b)
{
        return sqrt(double((b.d*a.x - a.d*b.x) * (b.d*a.x - a.d*b.x) +
                (b.d*a.y - a.d*b.y) * (b.d*a.y - a.d*b.y))) / abs(a.d * b.d);
}
```

String algorithms

Finding number of distinct substrings in O(n^2 lg n)

```
// Tested on distinct substrings..

#include <iostream>
#include <cstring>
#include <algorithm>
#include <stack>

using namespace std;

const int MAXL = 1 << 16;
const int MAXLG = 16;
int L;
char str[MAXL];
int pos[MAXL], lcp[MAXL];

/////////////////////////////////
```

```
struct ENTRY
{
        pair<int, int> nr;
        int p;
};

ENTRY M[MAXL], M_temp[MAXL];
int P[MAXLG][MAXL];
int Link[MAXL], bucket[256][257];
int depth;

inline bool cmp2(const ENTRY& a, const ENTRY& b)
{
        return a.nr < b.nr;
}

void make_suffix_array_nlog2n()
{
        for (int i = 0; i < L; i++)
                P[0][i] = str[i];

        depth = 1;
        for (int cnt = 1; (cnt >> 1) < L; depth++, cnt <<= 1)
        {
                for (int i = 0; i < L; i++)
                {
                        M[i].nr = make_pair(P[depth - 1][i], (i + cnt < L) ? P[depth - 1][i + cnt] : -1);
                        M[i].p = i;
                }

                sort(M, M + L, cmp2);

                for (int i = 0; i < L; i++)
                        P[depth][M[i].p] = (i && M[i].nr == M[i - 1].nr) ? P[depth][M[i - 1].p] : i;
        }
        for(int i = 0; i < L; i++)
                pos[P[depth-1][i]] = i;
}

void make_suffix_array_nlogn()
{
        for (int i = 0; i < L; i++)
                P[0][i] = str[i];

        depth = 1;
        for (int cnt = 1; (cnt >> 1) < L; depth++, cnt <<= 1)
        {
                for (int i = 0; i < L; i++)
                {
                        M[i].nr = make_pair(P[depth - 1][i], (i + cnt < L) ? P[depth - 1][i + cnt] : -1);
                        M[i].p = i;
                }

                memset(bucket, -1, sizeof(bucket));
                memset(Link, -1, sizeof(int) * L);
                for (int i = 0; i < L; i++)
                {
                        Link[i] = bucket[M[i].nr.first][M[i].nr.second + 1];
                        bucket[M[i].nr.first][M[i].nr.second + 1] = i;
                }
                int k = 0;
                for (int i = 0; i < 256; i++)
                        for (int j = 0; j < 257; j++)
                        {
                                int curr = bucket[i][j];
                                while(curr != -1)
                                {
                                        M_temp[k++] = M[curr];
                                        curr = Link[curr];
                                }
                        }
                for (int i = 0; i < L; i++)
                        M[i] = M_temp[i];

                for (int i = 0; i < L; i++)
                        P[depth][M[i].p] = (i && M[i].nr == M[i - 1].nr) ? P[depth][M[i - 1].p] : i;
        }
        for(int i = 0; i < L; i++)
                pos[P[depth-1][i]] = i;
}

int getlcp(int i, int j)
{
        if (i == j) return L - i;

        int ret = 0;
    for (int k = depth - 1; k >= 0 && i < L && j < L; k--)
        if (P[k][i] == P[k][j])
        {
                i += 1 << k;
                j += 1 << k;
                ret += 1 << k;
        }
    return ret;
}

/////////////////////////////////

inline bool cmp1(const int& i, const int& j)
{
        return strcmp(str + i, str + j) < 0;
}

void make_suffix_array_n2logn()
{
        for(int i = 0; i < L; i++)
                pos[i] = i;

        sort(pos, pos + L, cmp1);
}

/////////////////////////////////

void make_lcp_table_n2()
{
        memset(lcp, 0, sizeof(int) * L);
        for(int i = 1; i < L; i++)
                for(int k = 0; str[pos[i - 1] + k] == str[pos[i] + k]; k++, lcp[i]++);
}

void make_lcp_table_nlogn()
{
        memset(lcp, 0, sizeof(int) * L);
        for(int i = 1; i < L; i++)
                lcp[i] = getlcp(pos[i - 1], pos[i]);
}

int main()
{
        int t;
        scanf("%d", &t);
        while(t--)
        {
                scanf("%s", str);
                L = strlen(str);

                //make_suffix_array_nlog2n();
                make_suffix_array_nlogn();
                make_lcp_table_nlogn();

                int count = 0;
                for(int i = 0; i < L; i++)
                        count += L - pos[i] - lcp[i];

                printf("%d\n", count);
        }
        return 0;
}
```

Easy pattern matching

```
#include <iostream>
#include <vector>
#include <sstream>

using namespace std;

// Calculates ALL occurrences of pattern in text including overlapping ones!
// The concept behind this algorithm is terribly simple and can be deduced during contest.
// Z[i] represents the length of the longest common prefix of S[i..length(S)] and S.
// The Z table is very useful, and in fact the failure function of KMP and several preprocessing tables
// can be deduced from it immediately.
// Character delim has to be a character that does not occur in text or pattern.
// |pattern| = n, |text| = m, number of matches = z
// Time complexity = O(n + m)
// Space complexity = O(n + z)
vector<int> find_matches(const string& text, const string& pattern, bool only_first_match=false, char delim='$')
{
	vector<int> matches;
	int n = pattern.length();
	int m = text.length();
	int L = n + m + 2;
	vector<int> Z(n + 2);
	int l = 0, r = 0;
	// Build Z table
	#define STR(k)  (((k)<n+1)?pattern[(k)-1]:(((k)==n+1)?delim:text[(k)-n-2]))
	for(int k = 2; k < L; k++)
	{
		int Zk = 0;
		if(k > r)
		{
			for(int i = 1; i < L ; i++, Zk++)
				if(STR(i) != STR(k + i - 1))
					break;
			r = k - Zk - 1;
			l = k;
		}
		else
		{
			int kk = k - l + 1;
			int beta = r - k + 1;
			if(Z[kk] < beta)
				Zk = Z[kk];
			else
			{
				int q = 0, i = beta + 1;
				for(q = r + 1; q < L; q++, i++)
					if(STR(q) != STR(i))
						break;
				Zk = q - k;
				r = q - 1;
				l = k;
			}
		}
		// Record match
		if(k > n + 1 && Zk == n)
		{
			matches.push_back(k - n - 2);
			if(only_first_match)
				break;
		}
		if(k < n + 2)
			Z[k] = Zk;
	}
	return matches;
}
```

```
int main()
{
	/*int L = 1000000;
	stringstream ss;
	for(int i = 0; i < L; i++)
		ss << 'a';
	string text = ss.str();
	string patt = "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa";
	cout << "Done generating string, start matching..." << endl;
	*/
	string text = "ababababa";
	string patt = "aba";
	vector<int> matches = find_matches(text, patt);
	for(int i = 0; i < (int)matches.size(); i++)
		cout << matches[i] << endl;
	return 0;
}
```

Nice Trie

```
typedef int TYPE;
```

```
TYPE powerr(TYPE b, int e)
{
        if(e == 0) return 1;
        TYPE x = powerr(b, e >> 1);
        if(e & 1) return x * x * b;
        return x * x;
}

struct Trie
{
        vector<TYPE> cnt;
        vector<TYPE> link[2];
        int L;

        Trie(int mL) : L(mL)
        {
                cnt.push_back(0);
                link[0].push_back(0);
                link[1].push_back(0);
        }

        TYPE insert_and_count(const string& str, int i = 0, int v = 0)
        {
                TYPE val = cnt[v];
                if(i == (int)str.length())
                {
                        cnt[v] = powerr(2, (L - (int)str.length()));
                        //cerr << "Basis on " << v << " " << cnt[v] << " and b4 " << val << endl;
                        return val;
                }
                if(!link[str[i] - '0'][v])
                {
                        int n = cnt.size();
                        cnt.push_back(0);
                        link[0].push_back(0);
                        link[1].push_back(0);
                        link[str[i] - '0'][v] = n;
                }
                TYPE c = insert_and_count(str, i + 1, link[str[i] - '0'][v]);
                TYPE n = cnt[link[str[i] - '0'][v]];
                TYPE o = 0;
                if(link[1 - (str[i] - '0')][v])
                        o = cnt[link[1 - (str[i] - '0')][v]];
                cnt[v] = n + o;
                //cerr << "Recurse on " << v << " " << cnt[v] << " and b4 " << val << endl;
                return c;
        }
};
```

Abdenego's library

Dinic's algorithm

```
/**
 *   //////////////////
 *   // MAXIMUM FLOW //
 *   //////////////////
 *
 * This file is part of my library of algorithms found here:
 *      http://shygypsy.com/tools/
 * LICENSE:
 *      http://shygypsy.com/tools/LICENSE.html
 * Copyright (c) 2006
 * Contact author:
 *      abednego at gmail.c0m
 **/

/****************
 * Maximum flow * (Dinic's on an adjacency list + matrix)
 ****************
 * Takes a weighted directed graph of edge capacities as an adjacency
 * matrix 'cap' and returns the maximum flow from s to t.
 *
 * PARAMETERS:
 *      - cap (global): adjacency matrix where cap[u][v] is the capacity
 *          of the edge u->v. cap[u][v] is 0 for non-existent edges.
 *      - n: the number of vertices ([0, n-1] are considered as vertices).
 *      - s: source vertex.
 *      - t: sink.
 * RETURNS:
 *      - the flow
 *      - prev contains the minimum cut. If prev[v] == -1, then v is not
 *          reachable from s; otherwise, it is reachable.
 * RUNNING TIME:
 *      - O(n^3)
 * FIELD TESTING:
 *      - Valladolid 10330: Power Transmission (Gives WA, but it's probably my graph building that's wrong)
 *      - Valladolid 563:   Crimewave
 *      - Valladolid 753:   A Plug for UNIX
 *      - Valladolid 10511: Councilling
 *      - Valladolid 820:   Internet Bandwidth
 *      - Valladolid 10779: Collector's Problem
 * #include <string.h>
 **/

#include <string.h>
#include <stdio.h>

// the maximum number of vertices
#define NN 1024

// adjacency matrix (fill this up)
// If you fill adj[][] yourself, make sure to include both u->v and v->u.
int cap[NN][NN], deg[NN], adj[NN][NN];

// BFS stuff
int q[NN], prev[NN];
```

```
int dinic( int n, int s, int t )
{
    int flow = 0;

    while( true )
    {
        // find an augmenting path
        memset( prev, -1, sizeof( prev ) );
        int qf = 0, qb = 0;
        prev[q[qb++] = s] = -2;
        while( qb > qf && prev[t] == -1 )
            for( int u = q[qf++], i = 0, v; i < deg[u]; i++ )
                if( prev[v = adj[u][i]] == -1 && cap[u][v] )
                    prev[q[qb++] = v] = u;

        // see if we're done
        if( prev[t] == -1 ) break;

        // try finding more paths
        for( int z = 0; z < n; z++ ) if( cap[z][t] && prev[z] != -1 )
        {
            int bot = cap[z][t];
            for( int v = z, u = prev[v]; u >= 0; v = u, u = prev[v] )
                bot <?= cap[u][v];
            if( !bot ) continue;

            cap[z][t] -= bot;
            cap[t][z] += bot;
            for( int v = z, u = prev[v]; u >= 0; v = u, u = prev[v] )
            {
                cap[u][v] -= bot;
                cap[v][u] += bot;
            }
            flow += bot;
        }
    }

    return flow;
}

//----------------- EXAMPLE USAGE -----------------
int main()
{
    // read a graph into cap[][]
    memset( cap, 0, sizeof( cap ) );
    int n, s, t, m;
    scanf( " %d %d %d %d", &n, &s, &t, &m );
    while( m-- )
    {
        int u, v, c; scanf( " %d %d %d", &u, &v, &c );
        cap[u][v] = c;
    }

    // init the adjacency list adj[][] from cap[][]
    memset( deg, 0, sizeof( deg ) );
    for( int u = 0; u < n; u++ )
        for( int v = 0; v < n; v++ ) if( cap[u][v] || cap[v][u] )
            adj[u][deg[u]++] = v;

    printf( "%d\n", dinic( n, s, t ) );
    return 0;
}
```

Minimum cost max flow

```
/**
 *   //////////////////////
 *   // MIN COST MAX FLOW //
 *   //////////////////////
 *
 *   Authors: Frank Chu, Igor Naverniouk
 **/

/*********************
 * Min cost max flow * (Edmonds-Karp relabelling + fast heap Dijkstra)
 *********************
 * Takes a directed graph where each edge has a capacity ('cap') and a
 * cost per unit of flow ('cost') and returns a maximum flow network
 * of minimal cost ('fcost') from s to t. USE mcmf3.cpp FOR DENSE GRAPHS!
 *
 * PARAMETERS:
 *      - cap (global): adjacency matrix where cap[u][v] is the capacity
 *          of the edge u->v. cap[u][v] is 0 for non-existent edges.
 *      - cost (global): a matrix where cost[u][v] is the cost per unit
 *          of flow along the edge u->v. If cap[u][v] == 0, cost[u][v] is
 *          ignored. ALL COSTS MUST BE NON-NEGATIVE!
 *      - n: the number of vertices ([0, n-1] are considered as vertices).
 *      - s: source vertex.
 *      - t: sink.
 * RETURNS:
 *      - the flow
 *      - the total cost through 'fcost'
 *      - fnet contains the flow network. Careful: both fnet[u][v] and
 *          fnet[v][u] could be positive. Take the difference.
 * COMPLEXITY:
 *      - Worst case: O(m*log(m)*flow  <?  n*m*log(m)*fcost)
 * FIELD TESTING:
 *      - Valladolid 10594: Data Flow
 * REFERENCE:
 *      Edmonds, J., Karp, R.  "Theoretical Improvements in Algorithmic
 *          Efficieincy for Network Flow Problems".
 *      This is a slight improvement of Frank Chu's implementation.
 **/

#include <iostream>
using namespace std;

// the maximum number of vertices + 1
#define NN 1024

// adjacency matrix (fill this up)
int cap[NN][NN];

// cost per unit of flow matrix (fill this up)
int cost[NN][NN];

// flow network and adjacency list
int fnet[NN][NN], adj[NN][NN], deg[NN];

// Dijkstra's predecessor, depth and priority queue
int par[NN], d[NN], q[NN], inq[NN], qs;

// Labelling function
int pi[NN];
```

```
#define CLR(a, x) memset( a, x, sizeof( a ) )
#define Inf (INT_MAX/2)
```

```
#define BUBL { \
    t = q[i]; q[i] = q[j]; q[j] = t; \
    t = inq[q[i]]; inq[q[i]] = inq[q[j]]; inq[q[j]] = t; }

// Dijkstra's using non-negative edge weights (cost + potential)
#define Pot(u,v) (d[u] + pi[u] - pi[v])
bool dijkstra( int n, int s, int t )
{
    CLR( d, 0x3F );
    CLR( par, -1 );
    CLR( inq, -1 );
    //for( int i = 0; i < n; i++ ) d[i] = Inf, par[i] = -1;
    d[s] = qs = 0;
    inq[q[qs++] = s] = 0;
    par[s] = n;

    while( qs )
    {
        // get the minimum from q and bubble down
        int u = q[0]; inq[u] = -1;
        q[0] = q[--qs];
        if( qs ) inq[q[0]] = 0;
        for( int i = 0, j = 2*i + 1, t; j < qs; i = j, j = 2*i + 1 )
        {
            if( j + 1 < qs && d[q[j + 1]] < d[q[j]] ) j++;
            if( d[q[j]] >= d[q[i]] ) break;
            BUBL;
        }

        // relax edge (u,i) or (i,u) for all i;
        for( int k = 0, v = adj[u][k]; k < deg[u]; v = adj[u][++k] )
        {
            // try undoing edge v->u
            if( fnet[v][u] && d[v] > Pot(u,v) - cost[v][u] )
                d[v] = Pot(u,v) - cost[v][par[v] = u];

            // try using edge u->v
            if( fnet[u][v] < cap[u][v] && d[v] > Pot(u,v) + cost[u][v] )
                d[v] = Pot(u,v) + cost[par[v] = u][v];

            if( par[v] == u )
            {
                // bubble up or decrease key
                if( inq[v] < 0 ) { inq[q[qs] = v] = qs; qs++; }
                for( int i = inq[v], j = ( i - 1 )/2, t;
                     d[q[i]] < d[q[j]]; i = j, j = ( i - 1 )/2 )
                     BUBL;
            }
        }
    }

    for( int i = 0; i < n; i++ ) if( pi[i] < Inf ) pi[i] += d[i];

    return par[t] >= 0;
}
#undef Pot
```

```
int mcmf4( int n, int s, int t, int &fcost )
{
    // build the adjacency list
    CLR( deg, 0 );
    for( int i = 0; i < n; i++ )
    for( int j = 0; j < n; j++ )
        if( cap[i][j] || cap[j][i] ) adj[i][deg[i]++] = j;

    CLR( fnet, 0 );
    CLR( pi, 0 );
    int flow = fcost = 0;

    // repeatedly, find a cheapest path from s to t
    while( dijkstra( n, s, t ) )
    {
        // get the bottleneck capacity
        int bot = INT_MAX;
        for( int v = t, u = par[v]; v != s; u = par[v = u] )
            bot <?= fnet[v][u] ? fnet[v][u] : ( cap[u][v] - fnet[u][v] );

        // update the flow network
        for( int v = t, u = par[v]; v != s; u = par[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;
}

//----------------- EXAMPLE USAGE -----------------
#include <iostream>
#include <stdio.h>
using namespace std;
int main()
{
  int numV;
  //  while ( cin >> numV && numV ) {
  cin >> numV;
    memset( cap, 0, sizeof( cap ) );
    int m, a, b, c, cp;
    int s, t;
    cin >> m;
    cin >> s >> t;
    // fill up cap with existing capacities.
    // if the edge u->v has capacity 6, set cap[u][v] = 6.
    // for each cap[u][v] > 0, set cost[u][v] to  the
    // cost per unit of flow along the edge i->v
    for (int i=0; i<m; i++) {
      cin >> a >> b >> cp >> c;
      cost[a][b] = c; // cost[b][a] = c;
      cap[a][b] = cp; // cap[b][a] = cp;
    }
    int fcost;
    int flow = mcmf3( numV, s, t, fcost );
    cout << "flow: " << flow << endl;
    cout << "cost: " << fcost << endl;

    return 0;
}
```

Minimum cut

```
/**
 *  //////////////////////////////
 *  // Stoer-Wagner Minimum Cut //
 *  //////////////////////////////
 *
 * MAIN FUNCTION: minCut( n )
 *      Takes an undirected, weighted graph and returns the weight
 *      of the minimum cut in it. A cut is a set of edges that,
 *      when removed, disconnects the graph. A minimum cut is a
 *      cut of minimum total weight.
 * ALGORITHM:
 *      This is a O(n^3) implementation of the Stoer-Wagner
 *      deterministic algorithm (no randomization is required).
 * FIELD TESTING:
 *      - UVa 10989: Bomb, Divide and Conquer
 *
 * LAST MODIFIED:
 *      January 31, 2006
 *
 * This file is part of my library of algorithms found here:
 *      http://shygypsy.com/tools/
 * LICENSE:
 *      http://shygypsy.com/tools/LICENSE.html
 * Copyright (c) 2006
 **/

// Maximum number of vertices in the graph
#define NN 256

// Maximum edge weight (MAXW * NN * NN must fit into an int)
#define MAXW 1000

// Adjacency matrix and some internal arrays
int g[NN][NN], v[NN], w[NN], na[NN];
bool a[NN];


int minCut( int n )
{
    // init the remaining vertex set
    for( int i = 0; i < n; i++ ) v[i] = i;
```

```
// run Stoer-Wagner
    int best = MAXW * n * n;
    while( n > 1 )
    {
        // initialize the set A and vertex weights
        a[v[0]] = true;
        for( int i = 1; i < n; i++ )
        {
            a[v[i]] = false;
            na[i - 1] = i;
            w[i] = g[v[0]][v[i]];
        }

        // add the other vertices
        int prev = v[0];
        for( int i = 1; i < n; i++ )
        {
            // find the most tightly connected non-A vertex
            int zj = -1;
            for( int j = 1; j < n; j++ )
                if( !a[v[j]] && ( zj < 0 || w[j] > w[zj] ) )
                    zj = j;

            // add it to A
            a[v[zj]] = true;

            // last vertex?
            if( i == n - 1 )
            {
                // remember the cut weight
                best <?= w[zj];

                // merge prev and v[zj]
                for( int 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];

            // update the weights of its neighbours
            for( int j = 1; j < n; j++ ) if( !a[v[j]] )
                w[j] += g[v[zj]][v[j]];
        }
    }
    return best;
}

int main()
{
    // read the graph's adjacency matrix into g[][]
    // and set n to equal the number of vertices
    int n, answer = minCut( n );
    return 0;
}
```

Debugging in C++

C++:

```
#define DEBUG_FLAG 1
#if DEBUG_FLAG
#define DBG(z) cout << #z << ": " << (z) << endl;
#else
#define DBG(z)
#endif
```

To print

Don't forget to print:

- Articulation points algorithm
- Dominators algorithm
- Shygypsy's library

Notes

1. ↑ Many thanks to Ajay Somani (http://students.iiit.ac.in/%7Eajaysomani/) for providing me with the implementation:
2. ↑ http://forums.topcoder.com/?module=Thread&threadID=566309